FELIX (XGCD for FALCON): FPGA-based Scalable and Lightweight Accelerator for Large Integer Extended GCD

Sam Coulon, Tianyou Bao, and Jiafeng Xie

***Abstract*—The Extended Greatest Common Divisor (XGCD) computation is a critical component in various cryptographic applications and algorithms, including both pre- and post-quantum cryptosystems. In addition to computing the greatest common divisor (GCD) of two integers, the XGCD also produces Bézout coefficients b_a and b_b which satisfy $\mathrm{GCD}(a,b) = a \times b_a + b \times b_b$. In particular, computing the XGCD for large integers is of significant interest. Most recently, XGCD computation between 6,479-bit integers is required for solving N-th degree Truncated polynomial Ring Unit (NTRU) trapdoors in FALCON, a National Institute of Standards and Technology (NIST)-selected Post-Quantum digital signature scheme. To this point, existing literature has primarily focused on exploring software-based implementations for XGCD. The few existing high-performance hardware architectures require significant hardware resources and may not be desirable for practical usage, and the lightweight architectures suffer from poor performance. To fill the research gap, this work proposes a novel FPGA-based scalable and Lightweight accelerator for large Integer XGCD (FELIX). First, a new algorithm suitable for scalable and lightweight computation of XGCD is proposed. Next, a hardware accelerator (FELIX) is presented, including both constant- and variable-time versions. Finally, a thorough evaluation is carried out to showcase the efficiency of the proposed FELIX. In certain configurations, FELIX involves 81% less equivalent area-time product (eATP) than the state-of-the-art design for 1,024-bit integers, and achieves a 95% reduction in latency over the software for 6,479-bit integers (FALCON parameter set) with reasonable resource usage. Overall, the proposed FELIX is highly efficient, scalable, lightweight, and suitable for very large integer computation, making it the first such XGCD accelerator in the literature (to the best of our knowledge).**

***Index Terms*—FALCON, field-programmable gate array, hardware accelerator, large integer extended GCD, cryptographic applications.**

I. INTRODUCTION

RAPID progression in computer security has initiated a new round of cryptographic engineering research [1]–[3]. For instance, the Greatest Common Divisor (GCD) computation serves as one of the major components in many critical cryptographic applications and algorithms [4]–[6]. One application, the RSA cryptosystem requires verification that some large key parameters are coprime, which equates to confirming $\mathrm{GCD}(\cdot) = 1$ [7]. Accordingly, many related research activities have been carried out, including its implementation on both software and hardware platforms [5], [6]. The recent advance in the field, however, has gradually switched from software design to hardware implementation since the latter typically offers better performance [4], [6], [8].

Recently, a variant of the GCD algorithm, namely the large integer extended GCD (XGCD), has attracted significant attention from the research community [4], [6], [8]. Unlike the basic GCD function, XGCD also produces Bézout coefficients b_a and b_b, which satisfy $\mathrm{GCD}(a,b) = a \times b_a + b \times b_b$ [4]. The major cryptographic applications for XGCD are verifiable delay functions [9] and modular inversion [10]. More recently, the NIST-selected post-quantum digital signature scheme FALCON requires XGCD between 6,479-bit integers when solving NTRU trapdoors [11]. Accordingly, the research community has initiated to develop efficient implementations of XGCD on various platforms [4]–[6], [8], [10].

Prior Works and Existing Challenges. Overall, the research community has proposed four major methods to implement the above-mentioned large integer XGCD function [8], namely the extended Euclidean algorithm (EEA) [12], plus-minus (PM) algorithm [13], Two-Bit PM algorithm [14], and k-ary algorithm [15].

Meanwhile, related software and hardware implementations have been reported, which have mostly focused on high-performance implementations [4], [5], [8], [10]. The recent works in [4], [8] target high-performance applications. They have demonstrated the efficiency of hardware acceleration over software-based ones, and seem to represent the state-of-the-art in the literature. The work of [4] modified the Two-Bit PM algorithm to reduce the total number of necessary algorithm iterations. They further developed a high-speed architecture by performing single-cycle large integer arithmetic and reducing carry-propagation delay with CSAs (Carry-Save Adders). More recently, [8] adopted the k-ary algorithm, employing some strategies from Two-Bit PM and using redundant sign digit (RSD) representation to reduce carry-propagation for large integer arithmetic. On the other hand, [6] is the most recent (and only, to the best of our knowledge) existing lightweight XGCD accelerator. They utilize the BEEA algorithm and a sequential processing strategy to reuse hardware resources and avoid large integer arithmetic.

The major limitations of the existing works for XGCD hardware acceleration lie in the following three aspects. (i) The existing works like [4], [8] only consider the fast computation, and hence their applications are somewhat limited. For instance, design (1) in [4] requires 225,776 LUTs, 31,438 FFs,

This work is already accepted in IEEE TVLSI (with open access), see: https://ieeexplore.ieee.org/abstract/document/10593812

S. Coulon, T. Bao, and J. Xie are with the Department of Electrical and Computer Engineering, Villanova University, Villanova, PA, 19085 USA (e-mail: {scoulon,tbao,jiafeng.xie}@villanova.edu).

and 57,012 slices for 1,024-bit integers on Kintex-7 FPGA. Such large resource usage may not be ideal for FPGA-based practical applications for parameters as large as 6,479-bit integers, and especially when XGCD computation is just one component of a larger cryptographic scheme. (ii) The existing lightweight designs lack severely in speed. For example, the work of [6] presents a lightweight implementation, but the resulting performance is not sufficient for many applications. The fastest configuration of [6] for 2,048-bit integers still requires 1.68 ms for processing, and extrapolating from the given performance results, we estimate that it would take around 5.58 ms for 6,479-bit computation, which is 21% slower than our test of the FALCON software implementation [11]. (iii) Lastly, the existing works (including software ones) only reported the XGCD design for up to 1,024 or 2,048-bit integers, so an architecture for very large integers for applications like FALCON is missing in the literature.

Target Platform. This work targets the implementation of XGCD as one component of a theoretically larger scheme rather than a standalone chip. As such, FPGA was selected as the prototyping platform. FPGA-based platform provides many benefits for hardware accelerator design, including flexibility and relatively short development cycles. Application-specific integrated circuits (ASIC) could be selected in future work if a complete scheme was being implemented (FALCON, for example) with the goal of creating an end-product. Although [8] presented results only on the ASIC platform, [4], [6] presented implementation results on FPGA, so we primarily compare [4], [6] with our work. We also compare with the FALCON reference software code [11] for the evaluation of the XGCD computation over 6,479-bit integers.

Contributions. Following the above discussion, we aim to develop a novel FPGA-based scalable and lightweight accelerator for large integer XGCD (FELIX). In pursuit of this goal, we have made the following contributions.

- We have proposed a new algorithm for scalable XGCD computation based on the Two-Bit PM GCD algorithm. The proposed algorithm enables both constant- and variable-time processing options and features extensions based on our thorough analysis of potential speedup factors that are suitable for pipelined processing.
- We have proposed an efficient accelerator, FELIX, to perform the computation on the FPGA platform. The accelerator implements a pipelined, sequential processing strategy which achieves excellent balance between lightweight resource usage and efficient processing. We have given the architecture details and related design techniques for FELIX, including its different components, functions, and constant- and variable-time modes.
- We have lastly provided a thorough performance evaluation to demonstrate the efficiency of the proposed FELIX over the state-of-the-art designs. We also showcase the proposed FELIX's superior performance and practical resource usage when implementing XGCD for very large integers such as FALCON's 6,479-bit parameters.

To the authors' best knowledge, this is the first report in the literature of a scalable and lightweight XGCD accelerator with efficient processing, especially for computation over integers as large as 6,479 bits. We hope this work can stimulate many related research studies in the field.

The rest of the paper is arranged as follows. Section II introduces the preliminary knowledge. Section III introduces the algorithm background information. Section IV discusses the formulation of the proposed algorithm. Section V presents the proposed FELIX. Evaluation of the accelerator is provided in Section VI, including the extension to FALCON's parameter sets. The conclusion is finally given in Section VII.

II. PRELIMINARIES

A. XGCD

Definition. Given two integer inputs a and b, basic GCD computes the greatest common divisor, $\mathrm{GCD}(a, b)$. While the extended version, XGCD, computes $\mathrm{GCD}(a, b)$ and a pair of Bézout coefficients b_a and b_b satisfying the Bézout identity $\mathrm{GCD}(a, b) = a \times b_a + b \times b_b$.

Some Applications. XGCD has been deployed in interesting applications such as verifiable delay functions (VDF) [16] and modular inversion (suitable for elliptic curve cryptosystem) [5]. For example, given an integer a and a modulus b, one can find the modular inverse $(a^{-1} \mod b)$ by solving $\mathrm{GCD}(a, b) = a \times b_a + b \times b_b$. If $\mathrm{GCD}(a, b) = 1$ (meaning a and b are coprime), then $(b_a \mod b)$ can be returned as the modular inverse. The challenge is to solve for $\mathrm{GCD}(a, b)$, b_a, and b_b when inputs a and b are very large.

B. XGCD in FALCON

Recently, XGCD has been found in the NTRUSolve function of FALCON, a critical operation that exists in its Key Generation phase [11]. FALCON's Key Generation phase samples polynomials f and g and solves the NTRU equation to find polynomials F and G, where a valid set of (f, g, F, G) constitutes a valid private key [11]. f and g are initially polynomials over ring $\mathbb{Z}[x]/(x^n + 1)$ where $n = 1,024$ with small coefficients (5-8 bits). Polynomials f and g are repeatedly cast onto smaller rings $\mathbb{Z}[x]/(x^{n/2}+1)$, ..., $\mathbb{Z}[x]/(x^{n/4}+1)$, ..., etc., until f and g are plain integers over ring $\mathbb{Z}$. With each iteration, polynomial degree halves and coefficient size approximately doubles, so the final f and g become very large integers. In the reference code [11], they are represented in RNS notation with 209, 31-bit primes. Given $209 \times 31 = 6,479$, we estimate this as the maximum f and g bitsize, though the reference states they average around 6,307 bits [11]. From here we must find F and G such that $f \times G - g \times F = q$. Substituting $u \times q$ and $v \times q$ for G and F respectively yields $f \times u \times q - g \times v \times q = q$, and factoring out q reduces to $f \times u - g \times v = 1$. This overall process equates to the computing of XGCD, where f and g are the 6,479-bit input integers, u and v are the resultant Bézout coefficients, and the expected result is $\mathrm{GCD}(f, g) = 1$. Note that further steps are needed to raise F and G back to ring $\mathbb{Z}[x]/(x^n + 1)$, but those steps are not the focus of this work.

C. Notations

We have used the following notations throughout the paper. N denotes the bit-width of the integer inputs; (a_0, b_0) and

(a, b, u, m, y, n) are N-bit variables used in the algorithm and accelerator descriptions. Q refers to the bit-width of the parsed sections of the above-mentioned variables. Therefore, a_0, b_0, a, b, u, m, y, n values are stored and processed as N/Q individual Q-bit sections. R_E and R_O are the power-of-two reduction factors used to configure the proposed XGCD algorithm. L is the pipeline latency (Table I). I is the average number of iterations (Table II). In Algorithms 1, 2, and 3, "//" indicates floor division, which is bit-shifting in hardware.

A valid set of (N, Q, R_E, R_O) constitutes a configuration for the architecture of the proposed FELIX, while a set of just (R_E, R_O) is sufficient to characterize the configuration of the underlying algorithm. As such, we denote configurations of the algorithm as (R_E, R_O) which is discussed in the next section. For a given N, smaller Q, R_E, and R_O results in lower resource usage and slower processing, while larger Q, R_E, and R_O results in greater resource usage and faster processing.

III. Algorithm Background

This section provides some background on GCD algorithms as a basis for our work. We start by detailing some information on the Stein family of GCD algorithms [17] and then discuss the one we choose to build from, Two-Bit PM [14].

A. Background and Algorithm Selection

Existing work suggests that Euclid's (division-based) [18] and Stein's (subtraction-based) [17] algorithm families perform favorably for large integer implementations of GCD. Given our objective of low resource usage and the relatively high cost of division operations in hardware, we chose Stein's algorithm family for our design. Stein's algorithm (also known as Binary GCD) [17] and its variants [19], [13], [14] operate on the basis that given two integers a and b, one of two conditions must always be true: (1) a or b is even, or (2) $a \pm b$ is even. These algorithms reduce their inputs by dividing the inputs directly when they are even or dividing the sum/difference of the inputs when they are both odd. Division operations are always executed by a power-of-two reduction factor, which equates to simple bit-shifting in hardware design.

Stein-based algorithms [17], [19], [13], [14] can be classified according to the maximum number of bits they can reduce per iteration when a or b is even and when a and b are both odd. We follow [4] and denote this as (R_E, R_O), where R_E is the maximum reduction factor when a or b is even and R_O the factor when a and b are both odd. The algorithms can further be classified by whether they operate in constant- or variable-time, where constant-time efficiency is based on the worst-case iteration time, and variable-time mode finishes when the inputs a or b have been reduced to 0 [14].

Purdy's algorithm divides by 2 when a or b is even and 2 when a and b are both odd (2,2). Purdy's algorithm is simple but suffers from quadratic worst-case time complexity [19]. The PM algorithm introduced swapping mechanisms and still has reduction factors (2,2) but finishes in at most $(3.012 \cdot N)$ iterations [13]. The Two-Bit PM algorithm extended the PM algorithm, exploiting the fact that when a and b are both odd, one of $(a + b)\%4 = 0$ or $(a - b)\%4 = 0$ is always true, and thus some PM operations can be combined [14]. Two-Bit PM divides by at most 4 when a or b is even, and divides by 4 when a and b are both odd, so the original Two-Bit PM has reduction factors (4,4) for both constant- and variable-time modes, with worst-case reduced to $(1.51 \cdot N + 1)$ iterations. When [4] modified Two-Bit PM, they noted that while the option to reduce by 4 (instead of 2) when a or b is even does improve average iteration time for variable-time mode, it does not improve worst-case iteration time. So we remove these operations for Two-Bit PM constant-time GCD, resulting in reduction factors (2,4). In summary, Two-Bit PM in variable-time mode has reduction factors (4,4) and completes when $a = 0$ or $b = 0$. Constant-time mode has reduction factors (2,4) and finishes in $(1.51 \cdot N + 1)$ iterations.

We select the Two-Bit PM (Algorithm 1) as the basis to develop our algorithm for implementation because of its excellent balance between simple operations (addition/subtraction/shifting) and reasonable worst-case speed. Further, those operations are easily converted to sequential processing, i.e., they can be executed bit by bit. Furthermore, the variable-time version of the algorithm (4,4) can easily be extended to accommodate optimizations for our pipelined sequential processing strategy, which is discussed in the next section. Lastly, the original Two-Bit PM source [14] also provides insight on extension from GCD to XGCD, which is relevant to this work.

Algorithm 1: Two-Bit PM GCD [14]

Input : a_0, b_0 (odd N-bit integers);
Output: $\mathrm{GCD}(a_0, b_0)$;
Initialization step
$a \leftarrow a_0$, $b \leftarrow b_0$, $g \leftarrow 0$;
Main step
while $!end$ **do**
  if $!constant\_time$ *and* $a\%4 == 0$ **then**
    $a \leftarrow a//4$; $g \leftarrow g - 2$;
  else if $!constant\_time$ *and* $b\%4 == 0$ **then**
    $b \leftarrow b//4$; $g \leftarrow g + 2$;
  else if $a\%2 == 0$ **then**
    $a \leftarrow a//2$; $g \leftarrow g - 1$;
  else if $b\%2 == 0$ **then**
    $b \leftarrow b//2$; $g \leftarrow g + 1$;
  else if $g \geq 0$ *and* $(b + a)\%4 == 0$ **then**
    $a \leftarrow (a + b)//4$; $g \leftarrow g - 1$;
  else if $g \geq 0$ *and* $(b - a)\%4 == 0$ **then**
    $a \leftarrow (a - b)//4$; $g \leftarrow g - 1$;
  else if $g < 0$ *and* $(b + a)\%4 == 0$ **then**
    $b \leftarrow (a + b)//4$; $g \leftarrow g + 1$;
  else
    $b \leftarrow (a - b)//4$; $g \leftarrow g + 1$;
  end
  if *constant_time* **then**
    $iterations \leftarrow iterations + 1$;
    $end \leftarrow (iterations \geq 1.51 * N + 1)$;
  else
    $end \leftarrow (a == 0)$ or $(b == 0)$;
  end
end
Final step
$\mathrm{GCD}(a_0, b_0) \leftarrow a + b$;
return $\mathrm{GCD}(a_0, b_0)$;

B. Two-Bit PM Algorithm for GCD

The procedure for Two-Bit PM GCD computation is shown in Algorithm 1. The algorithm initializes with $a \leftarrow a_0$ and $b \leftarrow b_0$. There is also a small variable g that tracks how many times a or b has been reduced and determines which should be reduced next. g increments or decrements when a or b is being

reduced, so the lesser-reduced between a and b can always be determined by the sign of g. The Main Step of Algorithm 1 implements the main iteration loop of Two-Bit PM. On each iteration, when a or b is even, they are reduced by a factor of 2 (or 4 in variable-time mode), and g is incremented or decremented accordingly. When a and b are both odd, either $(a+b)\%4 = 0$ or $(a-b)\%4 = 0$ must be true [14]. When this is the case, g indicates whether a or b should be reduced. This procedure loops for $(1.51 \cdot N + 1)$ iterations in constant-time mode or until $a = 0$ or $b = 0$ in variable-time mode, and returns $\mathrm{GCD}(a_0, b_0) \leftarrow a + b$.

IV. FELIX: The Proposed Algorithm

In this section, we start by extending the Two-Bit PM GCD procedure (Algorithm 1) to Two-Bit PM XGCD to additionally calculate Bézout coefficients b_a and b_b. Then some further extensions to the algorithm are proposed to optimize processing, finally resulting in the proposed algorithm (Algorithm 2).

A. Proposed New Two-Bit PM Algorithm for XGCD

The original Two-Bit PM source [14] provides steps for the extension from GCD to XGCD computation. Our version is shown in Algorithm 2, where operations highlighted in blue are the only operations used for constant-time mode and where (R_E,R_O) is set to (2,4). Note that these are the same constant-time operations found in Algorithm 1. For variable-time mode, all operations in Algorithm 2 are available depending on the (R_E,R_O) configuration. We will discuss extension from GCD to XGCD, and the next subsection will discuss extensions for larger (R_E,R_O) configurations.

For extension to XGCD, we take Algorithm 1 and introduce Bézout variables u, m, y, and n, such that

$$\begin{aligned} a &= u \times a_0 + m \times b_0, \\ b &= y \times a_0 + n \times b_0, \end{aligned} \tag{1}$$

are always true, with $a \leftarrow a_0$, $b \leftarrow b_0$, $u \leftarrow 1$, $m \leftarrow 0$, $y \leftarrow 0$, and $n \leftarrow 1$ on startup. Similar to Algorithm 1, we proceed by reducing a or b directly when a or b are even, or reducing $a+b$ or $a-b$ when a and b are both odd. With each iteration we accordingly update u, m, y, and n to preserve the equality with a and b in (1). Thus, u and m must be updated when a is reduced, while y and n do not need to be updated because their equality with b in equation (1) still holds, and vice versa.

The procedure for updating u and m or y and n is shown in Algorithm 3 which is based on a similar procedure in [4]. Algorithm 3 is used in Algorithm 2 and called as $\mathrm{bez\_update}(\cdot)$. To understand its function, let us first consider the case where $a\%2 = 0$, so u and m need to be updated such that (1) still holds with $a \leftarrow a//2$. Remember that a_0 and b_0 are required to be odd. Assuming equation (1) currently holds, this means u and m must either be both even or both odd [14]. When u and m are even, we can reduce u and m directly so that $u \leftarrow u//2$ and $m \leftarrow m//2$. When u and m are odd, dividing by 2 will discard the lowest bit and break the equality. Instead, we add b_0 to u and subtract a_0 from m. Again, since a_0 and b_0 are odd, this produces an even result. Then the values can safely be reduced as $u \leftarrow (u+b_0)//2$ and $m \leftarrow (m-a_0)//2$.

Algorithm 2: Proposed new Two-Bit PM algorithm for large integer XGCD computation

Input : a_0, b_0 (Odd N-bit Integers);
Output: $GCD(a_0, b_0)$, b_a, b_b (N-bit Integers) such that $GCD(a_0, b_0) = a_0 \times b_a + b_0 \times b_b$;
Initialization step
1 $a \leftarrow a_0$, $b \leftarrow b_0$, $u \leftarrow 1$, $m \leftarrow 0$, $y \leftarrow 0$, $n \leftarrow 1$, $g \leftarrow 0$;
Main step
2 **while** $!end$ **do**
3 **if** $R_E \geq 32$ *and* $a\%32 == 0$ **then**
4 $a \leftarrow a//32$; $g \leftarrow g - 5$;
5 $u, m \leftarrow bez\_update(u, m, a_0, b_0, 5)$;
6 **else if** $R_E \geq 32$ *and* $b\%32 == 0$ **then**
7 $b \leftarrow b//32$; $g \leftarrow g + 5$;
8 $y, n \leftarrow bez\_update(y, n, a_0, b_0, 5)$;
9 ...*
10 **else if** $R_E \geq 2$ *and* $a\%2 == 0$ **then**
11 $a \leftarrow a//2$; $g \leftarrow g - 1$;
12 $u, m \leftarrow bez\_update(u, m, a_0, b_0, 1)$;
13 **else if** $R_E \geq 2$ *and* $b\%2 == 0$ **then**
14 $b \leftarrow b//2$; $g \leftarrow g + 1$;
15 $y, n \leftarrow bez\_update(y, n, a_0, b_0, 1)$;
16 **else if** $R_O \geq 32$ *and* $g \geq 0$ *and* $(b + a)\%32 == 0$ **then**
17 $a \leftarrow (a + b)//32$; $g \leftarrow g - 4$;
18 $u, m \leftarrow bez\_update(u + y, m + n, a_0, b_0, 5)$;
19 **else if** $R_O \geq 32$ *and* $g \geq 0$ *and* $(b - a)\%32 == 0$ **then**
20 $a \leftarrow (a - b)//32$; $g \leftarrow g - 4$;
21 $u, m \leftarrow bez\_update(u - y, m - n, a_0, b_0, 5)$;
22 **else if** $R_O \geq 32$ *and* $g < 0$ *and* $(b + a)\%32 == 0$ **then**
23 $b \leftarrow (a + b)//32$; $g \leftarrow g + 4$;
24 $y, n \leftarrow bez\_update(u + y, m + n, a_0, b_0, 5)$;
25 **else if** $R_O \geq 32$ *and* $g < 0$ *and* $(b - a)\%32 == 0$ **then**
26 $b \leftarrow (a - b)//32$; $g \leftarrow g + 4$;
27 $y, n \leftarrow bez\_update(u - y, m - n, a_0, b_0, 5)$;
28 ...**
29 **else if** $R_O \geq 4$ *and* $g \geq 0$ *and* $(b + a)\%4 == 0$ **then**
30 $a \leftarrow (a + b)//4$; $g \leftarrow g - 1$;
31 $u, m \leftarrow bez\_update(u + y, m + n, a_0, b_0, 2)$;
32 **else if** $R_O \geq 4$ *and* $g \geq 0$ *and* $(b - a)\%4 == 0$ **then**
33 $a \leftarrow (a - b)//4$; $g \leftarrow g - 1$;
34 $u, m \leftarrow bez\_update(u - y, m - n, a_0, b_0, 2)$;
35 **else if** $R_O \geq 4$ *and* $g < 0$ *and* $(b + a)\%4 == 0$ **then**
36 $b \leftarrow (a + b)//4$; $g \leftarrow g + 1$;
37 $y, n \leftarrow bez\_update(u + y, m + n, a_0, b_0, 2)$;
38 **else**
39 $b \leftarrow (a - b)//4$; $g \leftarrow g + 1$;
40 $y, n \leftarrow bez\_update(u - y, m - n, a_0, b_0, 2)$;
41 **end**
42 **if** *constant_time* **then**
43 $iterations \leftarrow iterations + 1$;
44 $end \leftarrow (iterations \geq 1.51 * N + 1)$;
45 **else**
46 $end \leftarrow (a == 0)$ or $(b == 0)$;
47 **end**
48 **end**
Final step
49 $GCD(a_0,b_0) \leftarrow a + b$, $b_a \leftarrow u + y$, $b_b \leftarrow m + n$;
50 **if** $GCD(a_0,b_0) < 0$ **then**
51 $GCD(a_0,b_0) \leftarrow -GCD(a_0,b_0)$, $b_a \leftarrow -b_a$, $b_b \leftarrow -b_b$;
52 **return** $GCD(a_0,b_0)$, b_a, b_b;

* (...) indicates repetitive else-if cases for $R_E \geq 16$, $R_E \geq 8$, $R_E \geq 4$ have been removed at Line 9;
** (...) indicates repetitive else-if cases for $R_O \geq 16$, $R_O \geq 8$ have been removed at Line 28;
* * * else-if conditions in blue are used for constant-time mode;

Let us now consider the case where a and b are both odd. Suppose $g \geq 0$ and $(a+b)\%4 = 0$ is met, so u and m need to be updated such that (1) still holds with $a \leftarrow (a+b)//4$. In the same way that b is added to a, y and n must be added to u and m, respectively. This can be observed in Algorithm 2 where $u \pm y$ and $m \pm n$ are sometimes fed into $\mathrm{bez\_update}(\cdot)$. From this point, the procedure is the same as when a or b are even and follows the operations of Algorithm 3.

We now have obtained all necessary procedures to update a, b, u, m, y, and n such that (1) holds on each iteration. The algorithm iterates until it reaches $(1.51 \cdot N + 1)$ iterations in constant-time mode [14], or until $a = 0$ or $b = 0$ in variable-

Algorithm 3: Bézout coefficient update function (bez_update($\cdot$)) based on [4]

Input : uy, mn, a_0, b_0 (N-bit integers), $numshift$;
Output: uy, mn (N-bit integers);
Main step
for $i \leftarrow 0$ *to* $numshift - 1$ **do**
  if $uy\%2 == 0$ **then**
    $uy \leftarrow uy//2$;
    $mn \leftarrow mn//2$;
  else
    $uy \leftarrow (uy + b_0)//2$;
    $mn \leftarrow (mn - a_0)//2$;
  end
end
return uy, mn;

time mode. At this point, the GCD and Bézout coefficients can obtained as $\mathrm{GCD}(a_0, b_0) = a+b$, $b_a = u+y$, and $b_b = m+n$. There is also a final step to invert $\mathrm{GCD}(a_0, b_0)$, b_a, and b_b, when $\mathrm{GCD}(a_0, b_0) < 0$.

B. Extensions of the Proposed Algorithm

To this point, we have established a constant-time (2,4) and variable-time (4,4) Two-Bit PM XGCD algorithm based on [14]. We now explore options to extend the variable-time version of the Two-Bit PM algorithm for optimized processing. Recall that the variable-time algorithm terminates when either of the input integers has been reduced to 0, so average-case analysis is concerned with the average number of iterations required to reach this point. If the implementation only supports division by 4 (basic (4,4) version), then it takes many iterations to reduce a or b to 0. However, when a, b, or $a \pm b$ is divisible by a larger power of 2, the algorithm can be extended to reduce more bits in a single iteration, thus reducing the average number of iterations required [4].

The trade-off is that reducing more bits per iteration requires the loop in the bez_update($\cdot$) function (Algorithm 3) to run longer. As opposed to [4] who operate over whole N-bit integers and evaluate all possible outcomes of bez_update($\cdot$) in a single cycle, we have unrolled the loop and implemented it as a pipeline (Fig. 3), operating over Q-bit sections. This strategy is explained further in Section V-A, but for now the important note is that more iterations of the bez_update($\cdot$) loop equates to a longer, more resource-intensive pipeline. We observe that our sequential processing strategy already requires at least N/Q (the number of sequential processing sections) cycles per iteration, so extending the pipelines to any length lower than N/Q cycles imposes no latency penalty for the length of a single iteration. Overall, we have conducted a simple analysis of these possible extensions, and the results are listed in Tables I, II, and III, respectively.

Table I presents the pipeline latency L (in number of cycles) for the extensions of the proposed algorithm. For example, if we configure to allow reduction by 32 in a single iteration (32,32), the pipeline latency is 11 clock cycles. For reduction by 32, we need $\log_2(32)$ iterations of the bez_update($\cdot$) loop (2 cycles each), and an optional 1 cycle for adding or subtracting inputs to bez_update($\cdot$) when it is called. So L is calculated as $\mathrm{Max}(2 \times \log_2(R_E), 1 + 2 \times \log_2(R_O))$. The pipeline latency (and majority of pipeline resource usage) is determined by the greater of R_E and R_O, so there is no latency benefit (and only minimal resource usage benefit) to setting $R_E < R_O$ or $R_O < R_E$, so we maintain $R_E = R_O$. Table II presents the average number of iterations (I) required to reduce a or b to 0 for a given configuration (R_E,R_O), which was obtained by simulating Algorithm 2 with uniformly random inputs. Table III then uses this data to compute the average number of clock cycles required to reduce a or b to 0, where values are calculated as $[\mathrm{Max}(N/Q, L) + 1] \times I$ (L is the pipeline latency and I is the average number of iterations).

TABLE I: Processing Pipeline Latency (L) (Num. Cycles)

(R_E,R_O)	(2,4)*	(4,4)*	(8,8)	(16,16)	(32,32)
L	5	5	7	9	11

*: Basic Two-Bit PM without extensions for comparison [14].

TABLE II: Average Number of Iterations (I)

N	(4,4)*	(8,8)	(16,16)	(32,32)
1024	1198	924	815	765
2048	2396	1849	1606	1531
6479**	7580	5937	5157	4917

*: Basic Two-Bit PM without extensions for comparison [14].
**: For XGCD used in FALCON [11].

TABLE III: Average Number of Clock Cycles

N	Q	(4,4)*	(8,8)	(16,16)	(32,32)
1,024	8	154,542	119,196	105,135	98,685
1,024	16	77,870	60,060	52,975	49,725
1,024	32	39,534	30,492	26,895	25,245
1,024	64	20,366	15,708	13,855	13,005
1,024	128	10,782	8,316	8,150	9,180
2,048	16	309,084	238,521	207,174	197,499
2,048	32	155,740	120,185	104,390	99,515
2,048	64	79,068	61,017	52,998	50,523
2,048	128	40,732	31,433	27,302	26,027
2,048	256	21,564	16,641	16,060	18,372
6,479**	32	1,542,293	1,207,994	1,049,288	1,000,456
6,479**	64	774,937	606,965	527,223	502,686
6,479**	128	391,258	306,451	266,190	253,802
6,479**	256	199,419	156,194	135,673	129,359
6,479**	512	103,500	81,066	70,415	67,138

*: Basic Two-Bit PM without extensions for comparison [14].
**: For XGCD used in FALCON [11].

When selecting a configuration (N, Q, R_E, R_O), there are two speedup factors to consider: (1) average number of iterations and (2) cycles per iteration. The total average number of clock cycles for processing is a product of these two values. First, given N, the average number of iterations is purely determined by the reduction factor (R_E,R_O) configuration of the algorithm. Table II demonstrates that increasing the reduction factor decreases the average number of iterations required, though the percent reduction decreases with each successive extension. For example, extension from (4,4) to (8,8) decreases the average number of iterations by 23%, while further extensions to (16,16) and (32,32) yield only a further 12% and 5% decrease, respectively. For this reason, we chose not to explore further extensions such as (64,64). Second, given N, cycles per iteration is computed as $\mathrm{Max}(N/Q, L) + 1$. In other words, cycles per iteration is the

greater of the number of sequential processing sections (N/Q) and pipeline latency (L). If $N/Q < L$, then pipeline utilization will be sub-optimal. For example, in Table III, for $N = 1024$, $Q = 128$, it can be seen that average number of clock cycles actually increases from (16,16) to (32,32) since the pipeline length $L = 11$ is greater than $N/Q = 8$. For optimal cycles per iteration, it is best to make N/Q as small as possible (i.e. make Q as large as possible) without dipping below L.

From Table III, we conclude that extensions from (4,4) to (8,8) and (16,16) generally provide tangible reductions in average number of cycles, which should outweigh the increases in resource usage. However, further extension to (32,32) seems relatively ineffective, and will require trade-off analysis with the related increase in resource usage. On the other hand, doubling Q yields significant reductions in average number of cycles, since doubling Q halves the number of cycles per iteration if the updated N/Q is still greater than L. However, we expect that larger Q, such as $Q = 512$, will incur serious max frequency penalties due to carry-propagation delay for larger integer arithmetic. Overall, the various configurations and trade-offs are explored in Section VI.

V. FELIX: The Proposed Architecture

A. Overall Design Strategy

In keeping with our objective of scalability and lightweight resource usage, FELIX implements a sequential processing strategy where N-bit integer variables are parsed into Q-bit sections, and each operation found in Algorithms 2 & 3 (Addition/Subtraction/Shifting) is executed sequentially, section-by-section, from least to most significant. Large integer arithmetic, such as addition, incurs carry-propagation delay proportional to the bit-width of the operands. Our strategy of processing section-by-section reduces this delay by operating over smaller Q-bit sections instead of the entire N-bit integers. Then, carry bits between those sections are registered, rendering carry-propagation between sections negligible.

The necessary pipelined sub-components are AddSub(), Add(), Sub(), and Shift(s), where AddSub() is configurable for Q-bit addition or subtraction, Add() and Sub() implement only addition or only subtraction respectively, and Shift(s) shifts a Q-bit value right by s number of bits. The sub-components are assembled into processing pipelines that execute all possible permutations of operations in Algorithms 2 & 3. Overall, the proposed accelerator (FELIX) is shown in Fig. 1, which consists of four major components: (i) Variable Memory, (ii) PipeAB (a,b processing pipeline), (iii) PipeUY and PipeMN, and (iv) Control Unit. These components, along with some other important design details, are described below.

B. Data Flags

Attached to each Q-bit section are a series of flags that assist with dataflow throughout the accelerator, i.e., $valid$, $first$, $final$, sub, $target$, and $numshift$. The $valid$ flag indicates that the current section is valid data. The $first$ flag indicates that the current section contains the least significant bits of the data, while the $final$ flag indicates that the current section contains the most significant bits. The sub flag

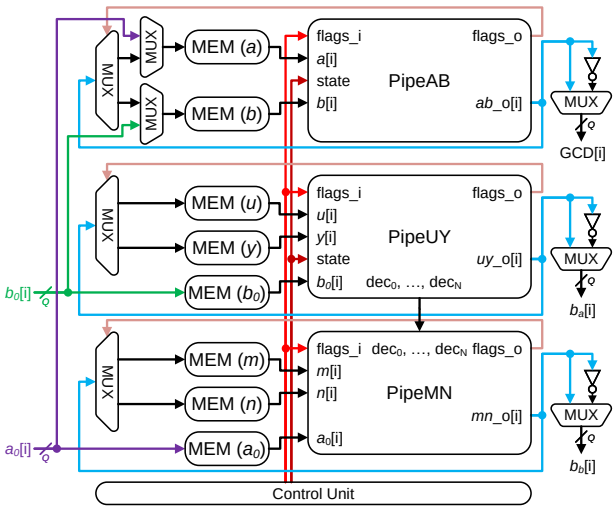

Fig. 1: Top Level of FELIX (MEM is Variable Memory).

configures the AddSub() blocks in the processing pipelines to perform subtraction. The $target$ flag indicates which variables are being processed (a, u, m or b, y, n). The $numshift$ flag indicates the reduction factor of the current operation, which configures the Shift($numshift$) block in PipeAB and enables and disables stages within PipeUY and PipeMN.

C. Constant- and Variable-Time Architectures

Overall, the architectures for constant- and variable-time processing are mostly the same. It can be seen in Algorithm 2 that, in general, the operations in blue (constant-time operations) require the same processing primitives (AddSub(), Add(), Sub(), Shift()) as the non-blue operations. Necessary components are mostly determined by Q and $\mathrm{Max}(R_E, R_O)$, so (2,4) and (4,4) configurations with the same Q require the same components, for example. The only difference is that the constant-time Control Unit maintains an iteration count for its termination condition, while the variable-time one contains logic to monitor when a or b has been reduced to 0.

D. Component Descriptions

Variable Memory. Input integers a_0 and b_0 and system variables a, b, u, m, y, n are all N-bit integers stored in RAM-like register structures of width Q and depth N/Q. With a stored in MEM_a, bits $a[N-1..N-Q]$ and $a[Q-1..0]$ are stored at indices $\mathrm{MEM}_a[N/Q-1]$ and $\mathrm{MEM}_a[0]$ respectively. Each structure has a dual port setup, with one read port and one write port. We tried using traditional BRAMs, but found that implementing with LUT/FF was much more efficient than BRAM when computing the equivalent number of slices (ENS) [20], which will be discussed later. Memory access is managed by the Control Unit. Since values always need to be processed in the order from section 0 to section $N/Q-1$, the access pattern is simply incrementing addresses from 0 to $N/Q-1$. To facilitate quicker next-state decisions, we also store an additional copy of the least significant bits of a and b in separate registers a_{lsb} and b_{lsb} (which are used to determine the divisibility of a, b, and $a \pm b$ within each iteration).

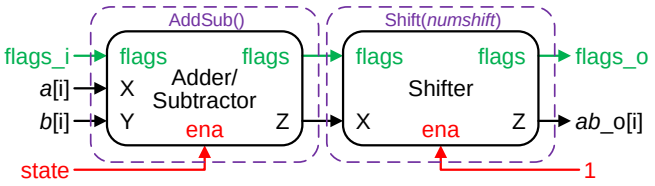

Fig. 2: PipeAB Component.

PipeAB Component. The PipeAB component is shown in Fig. 2 and implements the operations to reduce the a and b variables, which can be seen as the first assignment in each of the cases of the loop in Algorithm 2. Overall, this processing pipeline consists of one configurable AddSub() and one configurable Shift($numshift$). The sub flag received with input data configures the AddSub() to perform subtraction when necessary, and the $numshift$ flag configures the Shift($numshift$) block to shift by the correct number of bits. The maximum $numshift$ that must be supported is $\log_2(\text{Max}(R_E, R_O))$, which is the highest possible reduction factor set by the implementer. In cases where a or b is even, addition and subtraction are unnecessary, so the AddSub() block is inactive and acts as a pass-through register. Thus, the pipeline latency is 2 clock cycles in all cases.

PipeUY and PipeMN Components. The PipeUY and PipeMN components are shown in Fig. 3, which act as processing pipelines to implement the operations of the $\text{bez\_update}(\cdot)$ function (Algorithm 3). Note that for cases where a and b are both odd in Algorithm 2, when $\text{bez\_update}(\cdot)$ is called, the inputs u, y, m, and n need to be added or subtracted before they are fed to the function. So, similar to the PipeAB component, the first processing element in PipeUY and PipeMN is a configurable AddSub() block. The structure of the rest of the pipeline is dependent on (R_E,R_O) configuration. Let us consider a basic variable-time configuration (4,4) as a basis. The pipelines must support reduction by 4, which is 2 iterations of the $\text{bez\_update}(\cdot)$ loop. This is at most 2 Add() blocks for PipeUY or 2 Sub() blocks for PipeMN, as well as 2 Shift(1) blocks for both. With the initial AddSub(), this equates to a 5 clock cycle latency for each pipeline. For (8,8), the pipelines in Fig. 3 need to be extended with an additional Add() or Sub() and an additional Shift(1). This continues up to (32,32), which has an AddSub(), 5 total Add() or Sub(), and 5 total Shift(1) blocks to support a maximum of 5 iterations of the $\text{bez\_update}(\cdot)$ loop, totaling 11 clock cycle pipeline latency. These pipeline latencies can be seen in Table I.

Note that within the $\text{bez\_update}(\cdot)$ function, uy is checked on each iteration to see if it is even. This decision is implemented with the dec blocks in PipeUY in Fig. 3. When dec sees data with the $first$ flag (indicating these are the least significant bits), it checks the lowest bits of the z output of the preceding AddSub() or Add() block as well as the current overall $state$ operation that is being carried out. If the overall $state$ operation requires only reduction by 2, then the following stage(s) of Add()/Sub() and Shift(1) will be disabled and act as pass-through registers. If the overall $state$ operation enables another stage of Add()/Sub() and Shift(1), but z is odd, then both the proceeding Add()/Sub() and shift(1) components are enabled. If z is already even, Shift(1) is enabled but Add() or Sub() are disabled. When $state$ indicates that no more reduction is necessary, the remaining stages of Add() or Sub() and Shift(1) will be disabled, regardless of whether z is even or odd. This would indicate that the current operation is complete, and values should be passed through to the pipeline output.

Control Unit. The Control Unit manages the state machine and coordinates timing throughout the system. The primary function of the Control Unit is to manage memory access and send data to the correct processing pipelines (PipeAB, PipeUY, PipeMN). The Control Unit is also responsible for managing termination conditions. When running in constant-time mode, the Control Unit maintains an $iteration\_count$ and terminates the process when $iteration\_count$ exceeds $1.51 \cdot N + 1$. When in variable-time mode, the Control Unit monitors the output of PipeAB and terminates when the result is 0 (i.e. either a or b has been reduced to 0).

E. Dataflow

To understand the dataflow of the proposed FELIX, we consider an example using the constant-time configuration (2,4). The necessary operations are highlighted in blue in Algorithm 2. The discussed dataflow process can then be extrapolated for the variable-time operations and overall processing.

Initialization Step. On startup, a_0 and b_0 are shifted into the system section-by-section, from the least significant bits to most significant bits, and stored in variable memory. Note that these values are stored twice, (i) as a_0 and b_0 that will be accessed but not modified by the $\text{bez\_update}(\cdot)$ function, and (ii) as a and b that will be accessed as well as modified. Other variables are also initialized per Initialization Step in Algorithm 2. Then we enter the processing loop, i.e., the **Main Step** of Algorithm 2. Each iteration proceeds as follows:

- First, the least significant section of a and b (stored in a_{lsb} and b_{lsb}) as well as g are read to determine which $state$ operation of Algorithm 2 is selected. For example, if a is even, then the first constant-time condition ($a\%2 = 0$) in Algorithm 2 is met. g is decremented as $g \leftarrow g - 1$ since a is being reduced by 1 bit.
- Next, flags are set according to the determined $state$. For this case, $sub \leftarrow 0$, $target \leftarrow 0$, and $numshift \leftarrow 1$.
- The Control Unit begins reading sections from a, u, m, a_0, and b_0. Values b, y, and n are not needed during this operation, so they are not read. From here, the data sections from a are fed to PipeAB, sections from u and a_0 are fed to PipeUY, and sections from m and b_0 are fed to PipeMN. The sub, $target$, and $numshift$ flags travel with the data, and the $valid$, $first$, and $final$ flags are also appended to the proper data sections.
- Data populates the pipelines in parallel. For this state, a just needs to be reduced by a factor of 2, so the AddSub() block in PipeAB is disabled, and data gets shifted by $numshift = 1$. For PipeUY and PipeMN, the AddSub() blocks are also disabled, so data passes through to the first Add() or Sub(). When this happens, the first dec unit observes the $first$ flag from the first section of u passing through. If u is even, then the first Add() and

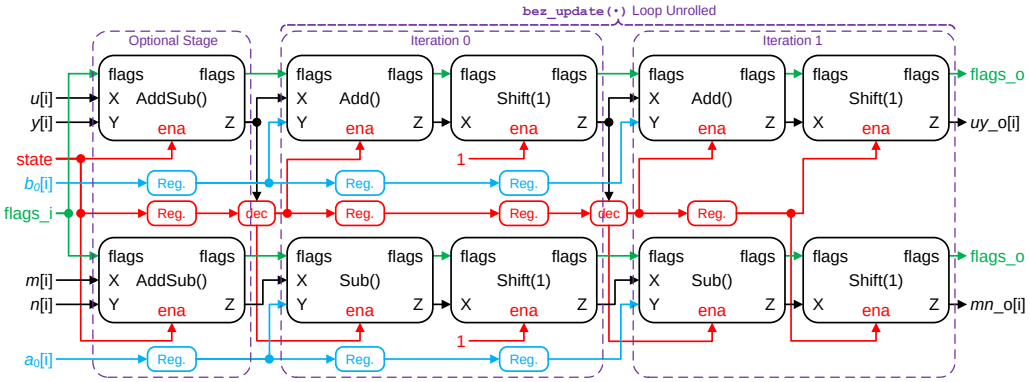

Fig. 3: The internal structure of the PipeUY/PipeMN component, based on (2,4) or (4,4) configuration.

Sub() blocks in PipeUY and PipeMN can be disabled. If u is odd, then Add() and Sub() are enabled. Then, they pass through the first Shift(1) block which shifts data by 1. At this point, the operation is complete, so the second dec block in PipeUY disables remaining stages of Add()/Sub() and Shift(1) in both PipeUY and PipeMN.

- Two clock cycles after data enters the processing pipelines, the first section of a exits PipeAB and is sent to be saved back in Variable Memory. The $target$ flag determines whether to save the resultant values in a or b (a since $target = 0$). After 3 more clock cycles, the first sections of u and m exit PipeUY and PipeMN (5 cycle total pipeline latency) and are also sent to be saved in Variable Memory via the same method.
- At this point, the first section of a data should have already returned to the Variable Memory, with its lowest bits updated in a_{lsb}. With a_{lsb}, b_{lsb}, and g all updated, the Control Unit can determine the next state ahead of time. The system continues reading a, u, m, a_0, and b_0 sections and sending them to the processing pipelines until the final section has been sent.
- The Control Unit then increments the iteration count in constant-time mode, or checks the output of PipeAB to see if the output is 0 in variable-time mode, and terminates if these conditions are true. This sequence of procedures repeats until the termination condition is met.

Final Step. When the termination condition is met, we enter the post-processing and data shift-out phase, the Final Step in Algorithm 2. Flags are set to perform the final addition and potential value inversion, with $sub \leftarrow 0$ and $numshift \leftarrow 0$. The Control Unit begins reading sections from a, b, u, y, m, n. Sections from a and b are fed to PipeAB, u and y are fed to PipeUY, and m and n are fed to PipeMN. The values are added together using the first AddSub() of each pipeline with the remaining pipeline stages disabled. The results $a+b$, $u+y$, and $m+n$ are returned to memory as they sequentially exit the pipelines. When the final result section of $a+b$ exits PipeAB, the data is analyzed to determine if $a+b$, $u+y$, and $m+n$ must be inverted. If $a+b$ is positive (i.e. MSB is a 0), then results are returned as $\mathrm{GCD}(a_0, b_0) \leftarrow a+b$, $b_a \leftarrow u+y$, and $b_b \leftarrow m+n$. If the $a+b$ is negative, then $\mathrm{GCD}(a_0, b_0)$ and Bézout results need to be inverted. Results are then returned as $\mathrm{GCD}(a_0, b_0) \leftarrow -(a+b)$, $b_a \leftarrow -(u+y)$, and $b_b \leftarrow -(m+n)$.

TABLE IV: ENS Computation Sheet

Type	Slice	DSP	BRAM(18K)	BRAM(36K)
#Slice	1	128	166	327
Weight	1.0	0.8	0.7	0.6
ENS	1	102.4	116.2	196.2

The listed data apply to AMD-Xilinx Series 7 FPGAs, as specified in [20].

VI. FELIX: Evaluation and Comparison

This section gives a detailed evaluation of the proposed FELIX on the FPGA platform. Results cover the implementation and comparison with a state-of-the-art high-performance XGCD accelerator, comparison with an existing lightweight XGCD design, and evaluation under the FALCON parameters.

A. Performance Metrics

We invoke many of the common FPGA performance metrics, such as Throughput, Throughput per Slice (TPS), and Area-Time Product (ATP). One shortcoming of a metric like ATP, which is generally calculated as (LUTs $\times$ Latency), is that it does not account for other resource usage, such as BRAM and DSP. So we additionally present results for ENS and Equivalent Area-Time Product (eATP). ENS is effective as a holistic resource-usage metric and is computed according to Table IV [20], where eATP=ENS$\times$Latency.

B. Experimental Setup

To test the effectiveness of the proposed FELIX, we have implemented the design in various (N, Q, R_E, R_O) configurations on the FPGA platform. The experimental setup is as follows: (a) The accelerator code is written in VHDL with functionality verified in ModelSim. (b) We have identified parameter combinations (N, Q, R_E, R_O) that effectively demonstrate the performance of FELIX compared with existing literature. (c) We have implemented the design (in both constant- and variable-time modes) in Vivado 2020.2 on the Kintex-7 (XC7K410TFBG676-1) for $N = 1,024$ and on the Ultrascale+ (XCZU7EG-FFVF1517-3-E) for $N = 2,048$ and $N = 6,479$. Note that for a fair comparison, we do not include the ASIC- or software-based designs (such as [8], [10]) for direct comparison, though we have discussed them in Section

TABLE V: Comparison with High-Performance Variable-Time XGCD Accelerator Based on FPGA Implementation

Device	Method	N	Q	LUT/FF/Slice	Fmax (MHz)	CCs	Latency (μs)	Throughput (Mbps)	TPS[a]	ENS[b] (K)	ATP[c] (K)	eATP[d] (K)
Variable-Time Reference [4] ($R_E = 8, R_O = 4$)												
Kintex-7	Modified Two-Bit PM	1,024	-	225,776/31,438/57,012	204.00	1,147	5.62	182.12	3.19	57.01	1,269.44	320.55
This Work ($R_E = 8, R_O = 8$)												
Kintex-7	New Two-Bit PM	1,024	32	1,717/1,426/547	277.78	30,492	109.77	9.33	17.05	0.55	188.48	60.04
			64	3,419/2,727/1,065	277.01	15,708	56.71	18.06	16.96	1.07	193.88	60.39
			128	6,201/5,331/1,897	212.77	8,316	39.09	26.20	13.81	1.90	242.37	74.14
This Work ($R_E = 16, R_O = 16$)												
Kintex-7	New Two-Bit PM	1,024	32	2,029/1,701/657	275.48	26,895	97.63	10.49	15.96	0.66	198.09	64.14
			64	3,920/3,275/1,205	267.38	13,855	51.82	19.76	16.40	1.21	203.13	62.44
			128	7,120/6,388/2,313	209.64	7,335	34.99	29.27	12.65	2.31	249.11	80.93
This Work ($R_E = 32, R_O = 32$)												
Kintex-7	New Two-Bit PM	1,024	32	2,255/1,976/792	262.47	25,245	96.18	10.65	13.44	0.79	216.89	76.18
			64	4,283/3,805/1,356	259.74	13,005	50.07	20.45	15.08	1.36	214.45	67.89
			128	-/-/-	-	9,180	-	-	-	-	-	-

[a]: TPS=Throughput/Slice. [b]: BRAMs and DSPs are converted to ENS. [c]: ATP=#LUTs×Latency. [d]: eATP=#ENS×Latency.

TABLE VI: Comparison with Lightweight Constant-Time XGCD Accelerator Based on FPGA Implementation

Device	Method	N	Q	LUT/FF/CLB	BRAM (18K)	Fmax (MHz)	CCs	Latency (μs)	Throughput (Mbps)	TPS[a]	ENS[b] (K)	eATP[d] (K)
Constant-Time Reference [6]												
Ultrascale+	BEEA	2,048	16	-/275/136	1.5	335.57	-	13,592.00	0.15	1.11	0.31	4,217.60
			32	-/370/220	1.5	312.50	-	7,335.00	0.28	1.27	0.39	2,892.19
			64	-/390/291	6.0	263.16	-	4,399.00	0.47	1.60	0.99	4,347.09
			128	-/564/554	12.0	232.56	-	2,540.00	0.81	1.46	1.95	4,948.94
			256	-/932/933	22.5	182.82	-	1,680.00	1.22	1.307	3.55	5,959.80
This Work ($R_E = 2, R_O = 4$)												
Ultrascale+	New Two-Bit PM	2,048	16	1,187/665/237	0	588.24	399,059	678.40	3.02	12.74	0.24	160.78
			32	2,272/1,995/467	0	495.05	201,076	406.17	5.04	18.34	0.28	111.70
			64	2,775/2,209/520	0	450.45	102,085	226.63	9.04	17.38	0.52	117.85
			128	5,471/4,326/1,071	0	401.61	52,589	130.95	15.64	14.60	1.07	140.24
			256	11,955/8,432/1,939	0	348.43	27,841	79.90	25.63	13.22	1.94	154.93

[a]: TPS=Throughput/Slice (CLB). [b]: BRAMs and DSPs are converted to ENS. [d]: eATP=#ENS×Latency.

VI-F. The FPGA utilization results are post place-and-route, but were not implemented on a physical FPGA, so listed frequencies may be slightly higher than what is achievable in a real system. Lastly, FALCON reference code for XGCD (zint_bezout($\cdot$) from keygen.c [11]) was benchmarked on the CPU: (i) the microbenchmark support library from Google [21] is used as the benchmark library; (ii) a single core of AMD Ryzen Threadripper 3960X processor (@3.8 GHz) is used; (iii) the Ubuntu 20.04 LTS OS (virtual machine) was used for testing; (iv) g++ 9.4.0 was used to compile the code and disable the optimization flag.

C. Comparison with High-Performance Variable-Time XGCD

Comparison. Table V reports the FELIX implementation results for comparison with the only (to our knowledge) high-performance large integer XGCD implementation on FPGA [4]. The design from [4] is also based on a modified Two-Bit PM with ($R_E = 8, R_O = 4$). Since [4] operates in variable-time with $N = 1,024$, we implemented FELIX with ($N = 1,024$, $Q = 32/64/128$) with ($R_E = 8/16/32$, $R_O = 8/16/32$). We focus on higher performance configurations of FELIX, ignoring lower Q, R_E, R_O. Also, ($N = 1,024$, $Q = 128$, $R_E = 32$, $R_O = 32$) is not recorded because of inefficient pipeline utilization, which was discussed in the proposed algorithm section (Section IV-B).

Analysis. Since [4] targets high-performance applications, and FELIX instead targets low resource usage and scalability, it is unsurprising that [4] utilizes significantly more ENS, but is generally much faster. Still, FELIX configuration $Q = 32$ with ($R_E = 8, R_O = 8$) achieves an 81% reduction in eATP over [4]. Further, the fastest presented FELIX configuration $Q = 128$ with ($R_E = 16, R_O = 16$) achieves a 96% reduction in ENS, and is only 6.2× slower than [4]. As expected, increasing Q significantly reduces processing time, but at the cost of resource usage approximately doubling and a lower max frequency. The optimal selection based on eATP seems to be $Q = 32/64$. For (R_E,R_O) configuration, (8,8) gave the best eATP, extension to (16,16) yielded the fastest possible processing, but extenstion to (32,32) provided no real benefit.

D. Comparison with Lightweight Constant-Time XGCD

Comparison. Next, Table VI reports the FELIX implementation results for comparison with a lightweight constant-time XGCD architecture which follows a similar sequential processing strategy to FELIX [6]. The design from [6] is based on a variant of the binary EEA from [5]. Since [6] operates in constant-time with $N = 2,048$ and $Q = 16/32/64/128/256$, we also implemented FELIX with those same parameters with ($R_E = 2$, $R_O = 4$), which is our constant-time configuration. [6] does not indicate whether 18K or 36K BRAMs are used, so we conservatively assess them as 18K.

Analysis. It is easily seen that FELIX outperforms [6] in almost all metrics. Since [6] utilizes BRAMs, they do achieve lower resource usage in terms of FF/CLB (LUTs not reported).

TABLE VII: Comparison with FALCON XGCD Reference Implementation Performance

Device	Method	N	Q	LUT/FF/CLB	Fmax (MHz)	CCs	Latency (μs)	Throughput (Mbps)	TPS[a]	ENS[b] (K)	ATP[c] (K)	eATP[d] (K)
FALCON [11]												
CPU	Binary GCD	6,479	-	-/-/-	-	-	4,412.92	-	-	-	-	-
This Work Constant-Time ($R_E = 2, R_O = 4$)												
Ultrascale+	New Two-Bit PM	6,479	32	2,841/1,234/502	568.18	1,990,797	3,503.80	1.85	3.68	0.50	9,954.30	1,758.91
			64	3,938/2,261/747	448.43	1,000,291	2,230.65	2.90	3.89	0.75	8,784.29	1,666.29
			128	6,082/4,365/1,079	398.41	505,038	1,267.64	5.11	4.74	1.08	7,709.81	1,367.79
			256	11,523/8,465/2,014	378.79	257,411	679.56	9.53	4.73	2.01	7,830.63	1,368.64
			512	25,507/16,748/4,100	308.64	133,598	432.86	14.97	3.65	4.10	11,040.86	1,774.71
This Work Variable Time ($R_E = 4, R_O = 4$)												
Ultrascale+	New Two-Bit PM	6,479	32	2,919/1,212/564	444.44	1,542,293	3,470.16	1.87	3.31	0.56	10,129.40	1,957.17
			64	4,018/2,248/717	432.90	774,937	1,790.10	3.62	5.05	0.72	7,192.64	1,283.50
			128	6,218/4,352/1,048	390.63	391,258	1,001.62	6.47	6.17	1.05	6,228.08	1,049.70
			256	12,506/8,499/1,977	357.14	199,419	558.37	11.60	5.87	1.98	6,983.02	1,103.90
			512	25,741/16,750/4,193	315.46	103,500	328.09	19.75	4.71	4.19	8,445.46	1,375.70
This Work Variable Time ($R_E = 32, R_O = 32$)												
Ultrascale+	New Two-Bit PM	6,479	32	3,588/2,051/725	411.52	1,000,456	2,431.11	2.67	3.68	0.73	8,722.81	1,762.55
			64	5,373/3,866/994	390.63	502,686	1,286.88	5.03	5.07	0.99	6,914.39	1,279.16
			128	8,547/7,469/1,564	383.14	253,802	662.42	9.78	6.25	1.56	5,661.72	1,036.03
			256	18,341/14,737/3,290	332.23	129,359	389.37	16.64	5.06	3.29	7,141.47	1,281.03
			512	38,660/29,187/6,406	301.20	67,138	222.90	29.07	4.54	6.41	8,617.27	1,427.89

[a]: TPS=Throughput/Slice (CLB). [b]: BRAMs and DSPs are converted to ENS. [c]: ATP=#LUTs×Latency. [d]: eATP=#ENS×Latency.

However, when accounting for BRAMs, FELIX achieves significantly lower ENS, eATP, and Latency. In particular, the fastest implementation ($Q = 256$) achieves 45% lower ENS, 97% lower eATP, and 95% lower Latency than [6] with the same Q. Even when configuring for lowest possible resource usage, FELIX implementation for $Q = 16$ achieves 23% lower ENS than [6] with the same Q while still achieving a 95% reduction in Latency. The BEEA algorithm used in [6] requires some integer division, which seems to lengthen their time per iteration, compared to FELIX's use of simple operations in Two-Bit PM. Further, our strategy to implement memory with LUT/FF instead of BRAM seemed to yield better ENS usage, especially for storing wide Q-bit data.

E. Comparison with FALCON XGCD Reference Code

Comparison. Finally, Table VII reports the FELIX implementation results for comparison with the FALCON reference code [11] which is written in C and was tested on CPU. Since FALCON reference documents do not indicate whether XGCD should be computed in constant- or variable-time, we present our results for both. Further, since the FALCON XGCD is implemented on the CPU, we only compare the Latency.

Analysis. For the constant-time implementation ($R_E = 2$, $R_O = 4$), the fastest implementation of FELIX ($Q = 512$) achieves a 90% reduction in Latency over the reference implementation. Then, in terms of the lowest resource usage implementation ($Q = 32$), FELIX achieves ENS of just 0.50K while still producing a 21% reduction in Latency.

For the variable-time implementation, the proposed accelerator of $Q = 512$ with ($R_E = 32$, $R_O = 32$) yields the fastest overall processing and produces a 95% reduction in Latency over [11]. In terms of eATP, FELIX ($Q = 128$) with ($R_E = 32$, $R_O = 32$) leads to the lowest eATP of 1,036.03K, while still giving 85% lower Latency compared to the reference implementation of [11].

F. Some Additional Considerations with Other Works

Note that there also exists other GCD/XGCD implementations like [10], [22], [23], but we do not explicitly compare them here since they belong to either software/ASIC implementations and have also been discussed and outperformed by [4]. However, we do want to mention that [8] is an ASIC-based XGCD design and represents the fastest existing accelerator in the literature. Although the direct comparison with their ASIC-based design is difficult, from Table III of [8], one can observe that [8] involves around 50.5% less eATP than [4] for $N = 1,024$. Since our design can achieve at most an 81% reduction in eATP when compared with [4] on the FPGA platform, we conclude that FELIX indirectly outperforms [8].

G. Discussion and Future Work

From the above analysis, it can be seen that FELIX performs very competitively with [4], significantly outperforms [6], and effectively implements the XGCD found in FALCON [11]. Regarding [4], it is clear that their accelerator targets vastly different applications from FELIX, and the authors seem to indicate that ASIC is the primary target of their work. However, their implementation on FPGA is the most advanced such design in the literature and is thus helpful for comparison using metrics like ATP and eATP, where it is shown that our accelerator performs favorably. [6] is effective for comparison because it implements a sequential processing strategy similar to FELIX. Considering this was the only recent lightweight implementation we could find in the literature, the need for a lightweight and scalable XGCD with reasonable efficiency like FELIX becomes even more evident. Lastly, we believe we have sufficiently proven the viability of a reasonably lightweight XGCD accelerator for extremely large parameter sets like the one used in FALCON. As new cryptographic schemes are being developed, we hope our research provides a foundation for potentially even larger parameters such as $N = 16,384$.

VII. Conclusion

This paper presents an FPGA-based scalable and lightweight accelerator for XGCD computation (FELIX). We have developed a modified version of the Two-Bit PM algorithm for XGCD computation suitable for pipelined sequential processing, and an FPGA-based accelerator to efficiently implement its functionality. We found that our proposed accelerator achieved more efficient area-time complexity compared to the state-of-the-art designs for both high-performance and lightweight accelerators in the literature. Lastly, we proved the viability of an FPGA-based accelerator for extremely large parameter sets such as the one used in FALCON, which has not yet been shown in the literature (to our best knowledge). We hope this work can positively impact related research in the field.

References

[1] S. Ullah, J. Zheng, N. Din, M. T. Hussain, F. Ullah, and M. Yousaf, "Elliptic curve cryptography; applications, challenges, recent advances, and future trends: A comprehensive survey," *Computer Science Review*, vol. 47, p. 100530, 2023.

[2] J. Xie, W. Zhao, H. Lee, D. B. Roy, and X. Zhang, "Hardware circuits and systems design for post-quantum cryptography–a tutorial brief," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 3, pp. 1670–1676, 2024.

[3] B. J. Lucas, A. Alwan, M. Murzello, Y. Tu, P. He, A. J. Schwartz, D. Guevara, U. Guin, K. Juretus, and J. Xie, "Lightweight hardware implementation of binary Ring-LWE PQC accelerator," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 17–20, 2022.

[4] K. Sreedhar, M. Horowitz, and C. Torng, "A fast large-integer extended GCD algorithm and hardware design for verifiable delay functions and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 163–187, 2022.

[5] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR transactions on cryptographic hardware and embedded systems*, pp. 340–398, 2019.

[6] S. Deshpande, S. M. Del Pozo, V. Mateu, M. Manzano, N. Aaraj, and J. Szefer, "Modular inverse for integers using fast constant time GCD algorithm and its applications," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 122–129.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[8] L. Ou, D. Zhu, J. Tian, and Z. Wang, "Fast hardware implementation for extended GCD of large numbers in redundant representation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2023.

[9] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *Annual international cryptology conference*. Springer, 2018, pp. 757–788.

[10] T. Pornin, "Optimized binary GCD for modular inversion," *Cryptology ePrint Archive*, 2020.

[11] Falcon, https://falcon-sign.info/.

[12] D. H. Lehmer, "Euclid's algorithm for large numbers," *The American Mathematical Monthly*, vol. 45, no. 4, pp. 227–233, 1938. [Online]. Available: http://www.jstor.org/stable/2302607

[13] R. P. Brent and H. Rung, "A systolic algorithm for integer GCD computation," in *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*. IEEE, 1985, pp. 118–125.

[14] D. Y. Yun and C. N. Zhang, "A fast carry-free algorithm and hardware design for extended integer GCD computation," in *Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, 1986, pp. 82–84.

[15] J. Sorenson, "The k-ary GCD algorithm," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1990.

[16] Chia network consensus explained. Online Webpage, 2021. https://manuals. plus/chia/chia-network-consensus-explained.

[17] J. Stein, "Computational problems associated with Racah algebra," *Journal of Computational Physics*, vol. 1, no. 3, pp. 397–405, 1967.

[18] D. H. Lehmer, "Euclid's algorithm for large numbers," *The American Mathematical Monthly*, vol. 45, no. 4, pp. 227–233, 1938. [Online]. Available: http://www.jstor.org/stable/2302607

[19] G. B. Purdy, "A carry-free algorithm for finding the greatest common divisor of two integers," in *Comp. and Maths. with Appls.*, vol. 9, no. 2, 1983, pp. 331–316.

[20] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, 2019.

[21] Benchmark: A microbenchmark support library Google. https://google.github.io/benchmark/random_interleaving.html.

[22] D. Zhu, J. Tian, and Z. Wang, "Low-latency architecture for the parallel extended GCD algorithm of large numbers," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[23] D. Zhu, Y. Song, J. Tian, Z. Wang, and H. Yu, "An efficient accelerator of the squaring for the verifiable delay function over a class group," in *2020 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2020, pp. 137–140.