A Fast Large-Integer Extended GCD Algorithm and Hardware Design for Verifiable Delay Functions and Modular Inversion

Kavya Sreedhar, Mark Horowitz and Christopher Torng

Stanford University, Stanford, USA
skavya@stanford.edu horowitz@ee.stanford.edu ctorng@stanford.edu

Abstract. The extended GCD (XGCD) calculation, which computes Bézout coefficients b_a, b_b such that $b_a * a_0 + b_b * b_0 = GCD(a_0, b_0)$, is a critical operation in many cryptographic applications. In particular, large-integer XGCD is computationally dominant for two applications of increasing interest: verifiable delay functions that square binary quadratic forms within a class group and constant-time modular inversion for elliptic curve cryptography. Most prior work has focused on fast software implementations. The few works investigating hardware acceleration build on variants of Euclid's division-based algorithm, following the approach used in optimized software. We show that adopting variants of Stein's subtraction-based algorithm instead leads to significantly faster hardware. We demonstrate this fact by performing a large-integer XGCD accelerator design space exploration to quantify the tradeoffs between Euclid- and Stein-based algorithms for various application requirements. This exploration leads us to an XGCD hardware accelerator that is flexible and efficient, supports fast average and constant-time evaluation, and is easily extensible for polynomial GCD. Our 16nm ASIC design calculates 1024-bit XGCD in 294ns (8X faster than the state-of-the-art ASIC) and constant-time 255-bit XGCD for inverses in the field of integers modulo the prime $2^{255} - 19$ in 87ns (31X faster than state-of-the-art software). We believe our chip is the first high-performance ASIC for the XGCD computation that is also capable of constant-time evaluation.

Keywords: Extended GCD · ASIC · Verifiable delay function · Class groups · Squaring binary quadratic forms · Constant-time · Modular inversion · Curve25519

1 Introduction

Computing the greatest common divisor (GCD) is a fundamental operation in number theory, with wide-ranging applications in cryptography [SRC20, NLRC10, RSA78, Mil85, Kob87]. Fast GCD algorithms repeatedly apply GCD-preserving transformations, primarily building from Stein's binary GCD algorithm [Ste67, Pur83, BK85, YZ86, Jeb93a, Por20] or Euclid's algorithm [Leh38, Col80, Jeb93b, Web95, Jeb95, Sor95, WTM05].

Both of these algorithms rely on the fact that the GCD of two numbers is the same as the GCD between their difference and the smaller number: $GCD(a, b) = GCD(|a - b|, \min(a, b))$. Stein's binary GCD algorithm [Ste67] directly uses this property when both a and b are odd but additionally removes factors of two to reduce the number of iterations: $GCD(a, b) = GCD(a/2, b)$ if a is even and $GCD(a, b) = GCD(a, b/2)$ if b is even. Euclid's algorithm relies on division to subtract as many multiples of the smaller input as possible: $GCD(a, b) = GCD(\min(a, b), \max(a, b) \mod \min(a, b))$.

Another family of subquadratic GCD algorithms are based on subquadratic multiplication and are asymptotically fast [Knu70, Sch71, Sch91, TY00, PW02, Möl08, SZ04].

These algorithms use a divide-and-conquer approach based on Lehmer's algorithm [Leh38] (which built from Euclid's algorithm) to recursively determine the quotients sequence.

Until recently, there had been very little advancement in fast GCD algorithms and in the extended GCD (XGCD) computation that also computes Bézout coefficients b_a, b_b satisfying the Bézout identity: $b_a * a_0 + b_b * b_0 = GCD(a_0, b_0)$. However, two recent developments suggest an increasing need for faster integer XGCD algorithms and implementations. The first is increased interest in squaring binary quadratic forms over class groups [Wes19] for verifiable delay functions (VDFs) [BBBF18], the computation for which XGCD is the primary bottleneck. The second is the realization that constant-time XGCD can be faster than Fermat's Little Theorem [DG11] for use in modular inversion [BY19, Por20].

These applications motivate rigorous exploration of fast XGCD hardware acceleration. However, only four relevant works currently exist in the literature. The first pair of works present 1024-bit XGCD ASIC designs based on Euclid's algorithm for squaring binary quadratic forms over a class group [ZST+20, ZTW21]. The other two works present FPGA designs for Euclid's algorithm [AHAJS16] and for the Bernstein-Yang algorithm [BY19] targeting constant-time modular inversion [DdPM+21]. Unfortunately, these prior works provide only point solutions to improve average performance (for squaring binary quadratic forms) or worst-case performance (for constant-time modular inversion). They also all build on Euclid's algorithm, citing its low iteration count and efficient software implementations.

In this work, we make key observations that suggest that Euclid's algorithm may not be the optimal target for hardware acceleration despite low iteration counts. In particular, execution time depends on the iteration count *and* on the time per iteration. Unlike in software, hardware designs have more influence over the time per iteration because they are not constrained by a processor's instruction set architecture and can instead implement fast and wide custom datapaths that do not correspond to any instruction. In addition, while individual applications may favor point solutions, one unified design that can efficiently support multiple applications is desirable since ASIC solutions can be expensive.

Leveraging these observations, we create an efficient parameterizable hardware architecture and conduct a detailed large-integer XGCD design space exploration that considers different application requirements and XGCD algorithms. In Section 2, we review the importance and requirements of the XGCD computation in our motivating applications. Then, in Section 3, we compare hardware execution times for Euclid- and Stein-based algorithms within the context of our application requirements. Despite the lower average number of iterations for Euclid-based algorithms, we find that using a redundant representation and carry-save adders for repeated addition significantly decreases the iteration time for Stein-based algorithms, resulting in faster average execution times. Both algorithm types have similar worst-case iteration counts, so this approach also results in faster worst-case execution times (used for constant-time XGCD). Thus, we contrast with prior XGCD hardware papers by building from Stein-based algorithms [YZ86]. In Section 4, we consider further algorithmic and hardware optimizations for higher performance. Section 5 optimizes the remaining operations required to build full accelerators with our XGCD design for our motivating applications. Section 6 evaluates our 16nm ASIC design.

Our work shows how we can compute a 1024-bit XGCD 8X faster than the state-of-the-art ASIC [ZST+20], which enables squaring binary quadratic forms 14X faster than optimized C++ on Apple's M1 processor in 5nm, and 255-bit XGCD 31X faster than the state-of-the-art software, directly translating into a 31X speedup for computing modular inverses for Curve25519. We believe this chip is the first high-performance ASIC for the XGCD computation that is also capable of constant-time evaluation.

Table 1: Summary of motivating cryptographic applications using large-integer XGCD.

	Squaring binary quadratic forms over class groups [Wes19]	Computing inverses mod $2^{255} - 19$ for Curve25519 [Ber06]
Constant-time	No	Yes
State-of-the-art algorithm	NUDUPL [JvdP02]	Optimized Stein's [Por20]
– Number of XGCDs	2	1
– XGCD % of execution time	91%	100%
– XGCD input bitwidth	1024+	255
– GCD = 1	Yes for 1st XGCD	Yes
– Requires minimal XGCD	Yes for 2nd XGCD	No
Other approaches		
– that use XGCD	[Lon19]	Bernstein-Yang [BY19]
– that do not use XGCD	N/A	Fermat's Little Theorem [DG11]

2 Applications

We focus on two cryptographic applications of XGCD that have drawn recent interest: squaring binary quadratic forms over class groups as a verifiable delay function and constant-time modular inversion for elliptic curve cryptography (Table 1). These applications represent two distinct spaces of application requirements (1024-bit fast average XGCD versus 255-bit constant-time XGCD), allowing us to showcase the flexibility of our design.

2.1 Verifiable Delay Functions

A verifiable delay function (VDF) is a cryptographic primitive that requires a fixed amount of sequential work to be evaluated but outputs a unique result that is still efficiently and publicly verifiable [BBBF18]. This fast verification but slow evaluation property is useful for adding delays in decentralized systems to avoid adversarial data manipulation, with applications detailed in [BBBF18]. In particular, VDFs have been considered a promising candidate as the core function for blockchain systems to disincentivize dishonest behavior: they have been integrated into the Chia Network's blockchain design, while the Ethereum Foundation and Protocol Labs anticipate that VDFs will also be crucial to their designs.

One proposed VDF construction is exponentiation in a group of unknown order such as an RSA group [Wes19, Pie18] or a class group [Wes19], which requires T sequential squarings performed in a group using a modulus N in order to compute $f(x) = x^{2^T}$ [BBF18]. The Chia Network chose to incorporate [Wes19]'s construction of squaring binary quadratic forms over a class group as a VDF in their blockchain design. We refer the reader to Buell's textbook [Bue89] for detail on binary quadratic forms. Since the repeated squaring in this construction continually doubles the output bitwidth, each squaring operation is followed by a reduction operation (Algorithm 5.4.2 of [CCC93]) to ensure that the output (and subsequent input) bitwidth do not exceed the bitwidth of the original input.

The Chia Network has hosted several competitions for fast software implementations for this repeated squaring computation. Both the Chia Network reference and the competition winner chose to implement the NUDUPL algorithm [JvdP02]. This algorithm not only computes the squaring operation, but it also partially reduces the output values to help with the reduction step to ensure values stay within a certain size.

Profiling the operations required for the NUDUPL algorithm with 1024-bit inputs shows that XGCD dominates, requiring 91% of the total execution time (Table 2). We averaged over one million trials of the Chia Network's reference C++ implementation on a 2020 MacBook Pro with the M1 chip, compiled with g++ and -O3 optimization, and used

Table 2: NUDUPL algorithm profiling on Apple's M1 processor with 1024-bit inputs.

Operation	% of execution time in 99.999% of squarings	% of execution time in few remaining squarings
XGCD	91	85
Modular Multiplication	5	8
Additions, Multiplications, Divisions	4	7

the standard C++ chrono library with nanosecond precision. The algorithm takes one of two branches in each squaring, depending on whether the size of intermediate variables need to be reduced. The branch taken significantly more often (99.999% of the time) computes two XGCDs: the first is the conventional XGCD while the second is a partial XGCD that terminates when the remainder in Euclid's algorithm is below a precomputed value instead of waiting until it is zero. The other branch only requires the first XGCD.

In the first XGCD, the GCD will always be one, so the important operation is finding a pair of Bézout coefficients. We observe that the partial XGCD can be replaced by a full XGCD that does not terminate early but still returns valid results as long as the Bézout coefficients are one of the two *minimal pairs* possible. While multiple solutions can satisfy Bézout's identity for a pair of inputs, solutions are called minimal pairs only if the absolute values of the Bézout coefficients are less than the absolute values of the inputs divided by the GCD [MH94]. Euclid's algorithm always returns such a pair, while Stein's-based algorithms may need additional iterations or a final correction to produce minimal results.

Understanding the speedup that dedicated hardware can achieve for this squaring helps determine the security level needed (i.e., the number of squarings and the minimum input bitwidth required) to guarantee a certain amount of time has passed with the VDF evaluation. Thus, high performance is the primary objective for VDF solutions. In addition, since this work is sequential by definition and not much computation can be done in parallel, area and power consumption are lesser concerns. Finally, as the name suggests, VDFs have a verification step, so their inputs are not secret. Thus, there is no need for constant-time evaluation to protect against timing attacks and it is beneficial to minimize the average XGCD execution time even if constant-time execution does not improve.

To ensure our XGCD speedup translates well into overall squaring speedup, we accelerate the other large-integer operations required (one modular multiplication and various additions, multiplications, and divisions) to build the first hardware accelerator for the NUDUPL algorithm (Section 5). Finally, we note that there are varying reports on a reasonable input bitwidth for class-group-based VDF applications, ranging from 833 bits [HM00] to 3000+ bits [DGS20]. Since both the Chia Network and recent work [ZST+20, ZTW21] use 1024-bit inputs, we also evaluate 1024-bit XGCD. We note that our hardware design is parameterizable by bitwidth, and that fortunately, our use of redundant binary representation makes our iteration time relatively independent of bitwidth (Section 4.2). Thus, our design ensures high performance even as bitwidths change.

2.2 Modular Inversion

A modular inverse of an integer $x \pmod{y}$ is defined as the integer x^{-1} such that $x * x^{-1} = 1 \pmod{y}$. This computation is used in public-key cryptography, including RSA [RSA78] and elliptic curve cryptography (ECC) [Mil85, Kob87]. In both of these applications, some values must be kept a secret: in RSA, the secret key is generated by inverting the public key and in ECC, the value to be inverted is a secret while the modulus is publicly known. Thus, there is a need for constant-time solutions where the execution time does not depend on the secret values, protecting such systems from timing attacks. For ECC, computing the modular inverse must take the same execution time regardless of the input value.

One part of ECC consists of elliptic curves defined over a finite field of positive integers

modulo a prime number p. Curve25519 is one of the fastest and mostly commonly used elliptic curves defined with $p = 2^{255} - 19$ [Ber06]. Operations on points of the elliptic curve consist of field operations, the most time-consuming of which is modular inversion with modulus p. As a result, many ECC implementations use different coordinate systems for most of the computation to avoid inversions, resulting in one inversion at the end [Ras17].

There are two approaches to find the modular inverse when the modulus is prime. The first method is Fermat's Little Theorem (FLT) [DG11], which states that $x^{-1} = x^{p-2} \pmod{p}$. For Curve25519, this computation requires 254 squarings and 11 multiplications [Ber06, BY19]. The second method finds the XGCD between x, the value to be inverted, and p, the modulus, where the Bézout coefficient associated with x is the resulting inverse x^{-1}. This is a valid approach because $x * x^{-1} = 1 \pmod{p} \rightarrow x * x^{-1} - 1 = 0 \pmod{p} \rightarrow x * x^{-1} - 1$ is divisible by p, so $x * x^{-1} - 1 = y * p$ for some y or $x * x^{-1} - 1 = -z * p$ for $z = -y$. This can be rewritten as the Bézout Identity: $x * x^{-1} + z * p = 1$. Thus, finding the XGCD returns the Bézout coefficient x^{-1} as desired.

Until recently, FLT was used more often to find modular inverses because it was faster than XGCD for constant-time execution [XGW+17]. In 2019, the Bernstein-Yang algorithm, a subquadratic XGCD algorithm, showed that it is possible for constant-time XGCD to be faster than FLT [BY19]. In 2020, Pornin instead optimized Stein's algorithm to achieve faster results than the Bernstein-Yang algorithm for computing inverses mod $2^{255} - 19$ on recent 64-bit x86 CPUs [Por20]. Pornin's work is used for generating RSA key pairs in the BearSSL library [Por18] (a C implementation of the SSL/TLS protocol) and Falcon [FaPKL+] (a cryptographic signature algorithm part of the NIST Post-Quantum Cryptography Project [nis17]). In 2021, the Bernstein-Yang algorithm was incorporated into the MirageOS unikernel operating system [HAS21]. This implementation is up to 2.5X faster than FLT work but 3.8X slower than Pornin's Curve25519 results, so Pornin's work remains the state of the art software for computing inverses mod $2^{255} - 19$. On the hardware side, only one prior work considers constant-time XGCD. This paper implements the Bernstein-Yang algorithm on an FPGA and is faster than prior FLT-based designs for Curve25519 [DdPM+21]. We note that the FPGA execution time is slower than the software records [BY19, Por20] since the FPGA is run at a much lower frequency (207 MHz) compared to the frequency of the Intel processors (2.3 GHz) in the software papers.

These recent works show the fast adoption of XGCD-based modular inversion with the modulus $2^{255} - 19$, and all but Pornin's approach build from Euclid-based algorithms. Given this growing interest, a more in-depth design space exploration is required determine the more suitable XGCD algorithmic approach for fast constant-time execution in hardware.

3 Hardware Performance Analysis of XGCD Algorithms

As described in Section 2, most prior work has focused on optimized software. The few prior hardware papers all build from Euclid-based algorithms and represent point solutions in the XGCD hardware acceleration space, optimizing either for fast average-case or worst-case performance. In this section, we explore the broader design space over multiple axes: algorithm family (Euclid versus Stein), target platform (software versus hardware), and application requirements (fast average-case performance versus worst-case performance). We use the worst-case execution to implement constant-time XGCD (Section 4.6). We find that using a redundant representation with the two-bit PM algorithm [YZ86] in the Stein family is faster in hardware for fast average-case *and* worst-case execution.

3.1 Algorithm Family

XGCD algorithms use the same control flow as GCD algorithms and iteratively apply GCD-preserving reduction transformations. This section overviews the transformations

used by the three major GCD algorithm families and their suitability for our applications.

Stein-based Algorithms Stein's algorithm continually reduces its inputs by subtracting the numbers when both are odd or by dividing by two when a number is even [Ste67]. Building from Stein's algorithm, the Purdy algorithm also removes factors of two when a, b are even but replaces the subtraction transformation with $GCD(a, b) = GCD(\frac{a+b}{2}, \frac{a-b}{2})$ to avoid the comparison needed to determine $\min(a, b)$ [Pur83]. Note that $a \pm b$ will be even when a, b are odd. The Plus-Minus (PM) algorithm further avoids large-integer comparisons by approximating the binary logarithm of $a - b$ [BK85]. The two-bit PM algorithm avoids further comparisons by duplicating cases in the PM algorithm and removing two factors of two when possible in a single iteration [YZ86].

Euclid-based Algorithms Euclid's algorithm continually reduces its inputs by dividing the numbers and replacing the larger number with the remainder from division. Building from Euclid's algorithm, Lehmer's algorithm [Leh38] provides faster solutions for large integers by leveraging the fact that most quotients in Euclid's algorithm are small and the initial parts of the quotients only depend on the most significant bits (MSBs) of the large inputs. Other papers build on this work with efficient techniques to detect when approximate division based on the MSBs is correct, further reducing the number of large-integer divisions required [Col80, Jeb93b, Jeb95, Sor95].

Asymptotically Fast Subquadratic Algorithms Subquadratic XGCD algorithms are based on subquadratic multiplication and are thus asymptotically fast. The Knuth-Schönhage algorithm [Knu70, Sch71], one of the earliest subquadratic GCD algorithms, uses a divide-and-conquer approach based on Lehmer's algorithm to recursively determine the quotients sequence. Further work more clearly details and extends these ideas [Sch91, TY00, PW02, Möl08], including Bernstein and Yang's recent constant-time approach [BY19] and binary recursive approaches based on Stein's algorithm [Sor94, SZ04].

Algorithm Suitability For the size of the input bitwidths in our applications (1024 bits for squaring binary quadratic forms and 255 bits for computing inverses mod $2^{255} - 19$), profiling from existing literature already strongly suggests that Euclid's and Stein's algorithms are faster than subquadratic algorithms [Möl08]. Furthermore, in more recent work, Pornin's XGCD implementation [Por20], published after Bernstein-Yang's subquadratic XGCD algorithm [BY19], uses a quadratic approach (Stein's algorithm) to achieve higher performance: 2000+ fewer cycles on recent x86 CPUs compared to [BY19]. Thus, we focus on comparing the XGCD execution time in hardware for Euclid- and Stein-based algorithms to determine the more promising family for hardware acceleration.

3.2 Target Platform

In both software and hardware, **execution time** is the product of the number of iterations and the time per iteration. The number of iterations only depends on the XGCD algorithm and is independent of the target platform. Thus, optimizations to reduce the number of iterations directly translates from the software to the hardware context. The time per iteration, or **iteration time**, is the latency for the longest series of data-dependent operations that complete in an iteration which is also known as the **critical path**. Note that this may not be the sum of latencies for all operations since some operations can be performed in parallel. Software is constrained to using instructions predefined in the processor's instruction set architecture (ISA), and thus software iteration times correspond to the longest set of dependent instructions. On the other hand, hardware is not limited to a set of pre-defined instructions, enabling the implementation of fast and wide custom datapaths for an extremely short iteration time.

Key Insight – We make the key observation that the additional influence that hardware has on iteration time opens the opportunity to select different XGCD algorithms that require a greater number of iterations but have far simpler operations that hardware exploits to significantly reduce iteration time, resulting in faster overall execution times.

3.3 Hardware Design Performance Comparison

We determine the optimal XGCD algorithm for hardware acceleration for different application requirements by estimating and comparing the number of iterations (Section 3.3.1) and iteration time (Section 3.3.2) for both the Stein and Euclid algorithm families in the average case and in the worst case, which leads to our constant-time implementation.

The key difference between the average and worst-case evaluation is the algorithm termination condition, which only affects the number of iterations (not iteration time). In the average case, we run the algorithm until the GCD is found, while in the worst case, the algorithm is run for a fixed number of iterations, set to the worst-case number.

3.3.1 Number of Iterations

Average Number of Iterations Required We generate iteration counts using uniform random 1024-bit inputs with our Python functional models. Since Euclid's algorithm divides every iteration while Stein's algorithm [Ste67] only reduces a factor of two, Euclid's algorithm requires 3.6X fewer iterations (598) compared to Stein's algorithm (2163) on average. The PM algorithm [BK85] requires even more iterations compared to Stein's algorithm since it can incorrectly approximate $a > b$ and reduce the smaller number instead of the larger one. The two-bit PM algorithm [YZ86] can reduce more bits each iteration than Stein's, requiring only twice the number of iterations (1195) required for Euclid's algorithm. Since Euclid-based algorithms have low iteration counts but require expensive divisions, prior work has focused on optimizing iteration time (and not iteration count). Thus, most optimized Euclid algorithms still require 598 iterations on average [Leh38, Sor95].

Worst-case Number of Iterations Required For Euclid's algorithm, the maximum number of iterations is $5\log(\min(a, b))$ [Mol97], when the inputs are Fibonacci numbers [Lam44]. In the Stein family, all algorithms reduce at least one bit per iteration, but only the two-bit PM algorithm can reduce two bits when a, b are odd. In addition, the approximations that the PM and two-bit PM algorithms make (to avoid comparisons that inefficiently result in reducing the smaller number instead of the bigger) do not occur often enough for their worst-case iterations to be higher than Stein's or Purdy's algorithm [BB87]. Thus, the two-bit PM algorithm requires the lowest worst-case number of iterations in the Stein family at $1.51 * \log_2(\min(a, b)) + 1$ iterations [YZ86]. For 255-inputs, this evaluates to 384 iterations, which is barely below the two-bit PM's 387. In general, these equations similarly closely track each other for the range of bitwidths we are interested in.

3.3.2 Iteration Time and Execution Time

Given the simplicity of operations in the two-bit PM algorithm compared to those in Euclid's algorithm, we expect the iteration time for the two-bit PM algorithm to be shorter. Then, for the worst-case execution time, the similarity in the worst-case number of iterations between the two algorithms in Section 3.3.1 is enough to conclude that the two-bit PM algorithm will likely yield faster constant-time implementations.

In the average case, the two-bit PM algorithm requires twice the number of iterations as Euclid's algorithm. Thus, the two-bit PM could be faster overall if its hardware critical path is less than half of the critical path of Euclid's algorithm. In this section, we compare the critical paths for optimal 1024-bit hardware designs and find this to be the case.

Carry-Save Adders (CSAs) Fast large-integer algorithms use carry-save adders to remove carry propagation delays in repeated additions [SKN08, RM19, Pur83, YZ86] for $O(1)$ instead of $O(\log(n))$ delays, where n is the input bitwidth. This is especially important in wide-word arithmetic with large bitwidths. CSAs take in three inputs and output two numbers, *carry* and *sum*, which represent the result in CSA form or redundant binary form, using two bits to represent one bit of the result. We denote this as 3:2 CSAs. Since the actual result is not directly stored, the value of the result is not known (and

thus cannot be compared to other values) but can be recovered with a normal addition with carry-propagation. We further describe CSAs in our hardware design in Section 4.2.

Since XGCD algorithms require repeated addition and/or subtraction, our hardware designs will rely on the performance savings gained using the CSA primitive. Thus, we report iteration times in the unit of CSA delays, which is also technology-agnostic. CSAs take in three numbers as input and output two numbers *carry* and *sum*. One of these additions corresponds to one CSA delay. In addition, multiplier arrays use CSAs to sum partial products quickly [FdCBM05, SKS09, RSPR11, JLL+15], so we can directly estimate multiplier delays in this unit. For non-CSA operations, we report latency in equivalent fractions of a CSA delay. In terms of basic logic gates, one CSA delay is approximately equal to the delay of two, two-input XOR gates in series.

Two-bit PM Algorithm The critical path for computing XGCD consists of the operations for adding odd values to reduce two bits in an iteration when a, b are odd and then shifting two bits to the left. Shifting is a fast operation in hardware, so we consider just the delay for the addition. This operation translates to adding five values in total: two values in CSA form and one constant (Section 4.1). Since CSAs take in three inputs, this operation requires three series CSAs (3 inputs go to first CSA; one addition input goes to the second; the last input goes into the third). We then multiply this delay by the average iteration count (1195) to estimate the average execution time as 3585 CSA delays.

Euclid's Algorithm The critical path for computing GCD with Euclid's algorithm consists of generating the quotient $q = \max(a, b)/\min(a, b)$ and then computing the remainder $\max(a, b) - q * \min(a, b)$ every iteration. The XGCD calculation also uses this quotient and then similarly multiplies and subtracts to generate the coefficient updates. These operations can be done in parallel with the GCD computation, so generating these additional results does not change the critical path. We denote a as the larger number and b as the smaller number to simply rewrite the critical path computation as $a - q * b$.

We first look at prior division algorithms to determine the latency for generating the quotient. Division algorithms are iterative, requiring repeated multiplication or subtraction. Multiplication-based division [Fly70, Gol64] will not be competitive because many slow large-integer multiplications are required. Each division would require hundreds of CSA delays, so this approach is not used in high-performance solutions. Thus, prior XGCD algorithms instead estimate the quotient by looking at only the most significant bits of the dividend and divisor [Leh38, Jeb93b, Jeb95, Sor95]. Using a lookup table (LUT) is the fastest way to accomplish this. Since only the top bits are used, this estimate can be incorrect. In that case, we need to execute the slow large-integer division. We find that even in the optimistic scenario where no large-integer divisions were required, the execution time for division-based XGCD is slower than the two-bit PM delay.

To calculate critical path delay, we need to consider the delay for generating the quotient (q), multiplying q and b ($q * b$), and subtracting to generate the remainder ($a - q * b$). For an optimistic estimate, we assume the look up to get the quotient estimate takes zero delay. The LUT takes pairs of c bits of a, b as input, denoted a_c, b_c, respectively. Since a, b are stored in CSA form, we require c-bit carry-propagate adds to get a_c, b_c. This requires $\lfloor log_2(c) \rfloor + 1$ CSA delays for a binary logarithmic adder tree structure, which evaluates to three CSA delays for $c = 5$ and four for $c = 9$. The LUT has 2^{2*c} entries, where each entry is c bits, so LUTs with $c \geq 10$ require over a million entries and are impractical. We precompute the LUT entries for $a_c/(b_c + 2)$ to guarantee that the quotient estimate is not greater than the true quotient when we use the most significant of a, b in CSA form. Then, the LUT entry can be shifted to the left to obtain the quotient estimate, denoted $q_{estimate}$.

We then need to compute $q_{estimate} * b$ as $q_{estimate} * b_{carry} + q_{estimate} * b_{sum}$ since b is in CSA form. Since these two multiplications can be computed in parallel, for a c-bit $q_{estimate}$, the delay for this operation is equal to the delay of generating c partial products and using CSAs to sum them together. For this addition, we need an adder tree with

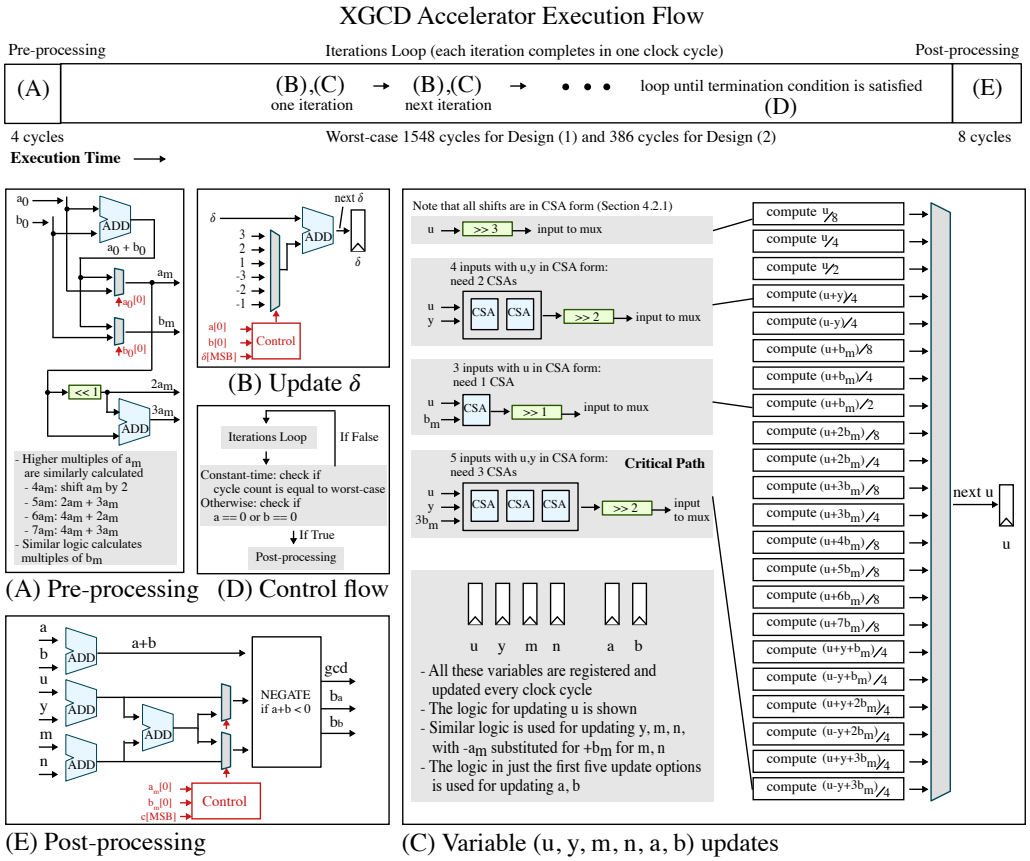

Figure 1: XGCD Execution Flow Diagram – Key components in the execution flow are broken out in detail. (A) Pre-processing step to generate inputs to iterations loop; (B) Update for δ, which approximates $a > b$ when a, b are odd; (C) Variable updates for a, b, u, y, m, n registers in the iterations loop illustrating the wide parallel datapath with late select (the logic for key pipes are shown in more detail); (D) State diagram for control flow including termination condition; (E) Post-processing step to generate XGCD outputs.

roughly $\lfloor \log_{3/2}(c) \rfloor$ series 3:2 CSAs. For $c = 5$, we require three CSA delays and for $c = 9$, we require four. Finally, to compute $a - q_{estimate} * b_{carry} - q_{estimate} * b_{sum}$, we need an adder tree with six inputs, which has a delay of three CSAs.

Adding the CSAs delays to generate $q_{estimate}$, compute $q_{estimate} * b$, and subtract to get the remainder requires at least $3 + 3 + 3 = 9$ CSA delays for five-bit $q_{estimate}$ and $4 + 4 + 3 = 11$ for nine-bit $q_{estimate}$. Thus, the iteration time estimate for division-based XGCD is 3X shorter than the two-bit PM critical path delay estimate. Then, even without considering the latency for large-integer division corrections, the two-bit PM execution will be 1.5X faster, so division-based XGCD is not competitive for hardware deisgns.

4 Fast XGCD Algorithm and Hardware Design Space

Based on our analysis in Section 3, we build from the two-bit PM algorithm [YZ86]. Since this algorithm was originally not completely specified, especially regarding iterative updates for odd Bézout coefficient values, we first present a *complete extended two-bit PM algorithm*. We then present algorithmic optimizations (handling carry propagation for the termination condition and increasing the number of bits reduced per iteration) and

Table 3: Update possibilities for Bézout coefficient variables u, m when a is shifted by two.

Divisibility of u, m	u_{update}	m_{update}
u, m divisible by 4	$u/4$	$m/4$
u, m divisible by 2 but not 4	$(u + 2 * b_m)/4$	$(m - 2 * a_m)/4$
$u + b_m, m - a_m$ divisible by 4	$(u + b_m)/4$	$(m - a_m)/4$
$u + b_m, m - a_m$ divisible by 2 but not 4	$(u + 3 * b_m)/4$	$(m - 3 * a_m)/4$

hardware design choices (using carry-save adders and minimizing control overhead) that together enable high-performance XGCD hardware acceleration. Finally, we show how our design easily supports constant-time evaluation and polynomial XGCD.

Figure 1 shows the execution flow of our hardware with the sequence of operations on the critical path. Note that in our hardware design, we execute one iteration each clock cycle. Thus the operations on the iteration critical path (Section 3.2) must finish in a clock cycle time, and the minimum clock cycle time is roughly equal to this critical path.

4.1 Complete Extended GCD with Two-Bit PM

Listing 1 shows our XGCD algorithm. This section explains the subset of the algorithm that corresponds to the two-bit PM algorithm and our extensions to compute the XGCD. Section 4.3 explains rest of the updates included and their suitability for constant-time versus fast average-case evaluation.

Existing GCD algorithms can be extended to calculate the Bézout coefficients by introducing four intermediate variables u, m, y, n such that $u * a_m + m * b_m = a$ and $y * a_m + n * b_m = b$ hold true for inputs a_m, b_m in every iteration. We denote these relations together as Equation 1. At the start of the iterations loop, $a = a_m$ and $b = b_m$, so the initial values must be $u = 1, m = 0$ and $y = 0, n = 1$.

During the iterations loop, the two-bit PM algorithm updates either a or b. When a is not updated, there is no need to update u, m since Equation 1 automatically holds. The same is true for b and y, n. When a or b is divided by two or four, we need to divide u, m or y, n by the same factor to maintain the relations in Equation 1. However, the divisibility of the coefficients may not match that of a and b, respectively. Then, if odd values are shifted, the truncated relations may not preserve these relations.

To address this problem, we consider the shift-by-one case first. If a is even and thus divided by two, we need to ensure that $u_{update} * a_m + m_{update} * b_m = \frac{a}{2}$. If the previous u, m are even, the update is straightforward: $u_{update} = \frac{u}{2}, m_{update} = \frac{m}{2}$. If the previous u, m are odd, we *add* b_m to u and *subtract* a_m from m as similarly done to extend the PM algorithm to compute XGCD [BK85]. Since b_m, a_m are odd by construction and the sum of two odd numbers is even, $u + b_m, m - a_m$ will be even. Then, we can reduce one bit by computing $u_{update} = \frac{u+b_m}{2}, m_{update} = \frac{m-a_m}{2}$. This update preserves the relation in Equation 1, since we have added and subtracted $\frac{a_m * b_m}{2}$ from the result.

For the shift-by-two case, we apply our updates rules for the shift-by-one case twice to satisfy $u_{update} * a_m + m_{update} * b_m = \frac{a}{4}$. Then, the worst-case update rule for the shift-by-two case is when m is not divisible by two and $m - a_m$ is not divisible by four, resulting in $m_{update} = \frac{\frac{m-a_m}{2} - a_m}{2}$. To reduce this update delay, we rewrite this update as $\frac{m-3*a_m}{4}$, noting that rounding due to the truncation when shifting is preserved. Since a_m is known at the start of the computation, $3 * a_m$ is a constant that we can precompute in a pre-processing step.[1] We similarly rewrite the other u, m, as shown in Table 3. Thus, the worst-case update delay is half the original form and similar to the shift-by-one case.

[1] We compute $3 * a_m$ as $2 * a_m + a_m$, so it requires one cheap left shift and one addition.

```
# inputs:
# a_m, b_m (int) -- numbers to find the gcd for, with factors of two removed
# constant_time (bool) -- whether this should use our constant-time algorithm
# bitwidth (int) -- the maximum bitwidth of the inputs

# outputs:
# gcd (int) -- gcd(a_m, b_m)
# u, l (int) -- Bezout coefficients u, l such that u * a_m + l * b_m = gcd(a_m, b_m)

def xgcd(a_m, b_m, constant_time, bitwidth):
    # Step 1: Pre-processing

    # ensure a_m, b_m are odd
    if (a_0%2 == 0): a_m = a_0 + b_0; b_m = b_0
    elif (b_0%2 == 0): a_m = a_0; b_m = a_0 + b_0
    else: a_m = a_0; b_m = b_0

    if constant_time: iterations = 0

    # δ ≈ log_2(a) − log_2(b) ≈ a − b to approximate if a > b
    a = a_m; b = b_m; u = 1; m = 0; y = 0; n = 1; δ = 0; end_loop = False

    # Step 2: Iteration loop

    def xgcd_update(num_bits_reduced, u, m, b_m, a_m):
        for i in range(num_bits_reduced):
            if u%2 == 1: u = (u + b_m)//2; m = (m − a_m)//2
            else: u = u//2; m = m//2
        return (u, m)

    while (not end_loop):
        if (not constant_time and (a%8 == 0)):
            a = a//8; δ = δ − 3; (u, m) = xgcd_update(3, u, m, b_m, a_m)
        elif (not constant_time and (a%4 == 0)):
            a = a//4; δ = δ − 2; (u, m) = xgcd_update(2, u, m, b_m, a_m)
        elif (a%2 == 0):
            a = a//2; δ = δ − 1; (u, m) = xgcd_update(1, u, m, b_m, a_m)
        elif (not constant_time and (b%8 == 0)):
            b = b//8; δ = δ + 3; (y, n) = xgcd_update(3, y, n, b_m, a_m)
        elif (not constant_time and (b%4 == 0)):
            b = b//4; δ = δ + 2; (y, n) = xgcd_update(2, y, n, b_m, a_m)
        elif (b%2 == 0):
            b = b//2; δ = δ + 1; (y, n) = xgcd_update(1, y, n, b_m, a_m)
        elif ((δ ≥ 0) and ((b + a)%4 == 0)):
            a = (a + b)//4; δ = δ − 1; (u, m) = xgcd_update(2, u + y, m + n, b_m, a_m)
        elif ((δ ≥ 0) and ((b − a)%4 == 0)):
            a = (a − b)//4; δ = δ − 1; (u, m) = xgcd_update(2, u − y, m − n, b_m, a_m)
        elif ((δ < 0) and ((b + a)%4 == 0)):
            b = (a + b)//4; δ = δ + 1; (y, n) = xgcd_update(2, u + y, m + n, b_m, a_m)
        else:
            a = (a − b)//4; δ = δ + 1; (y, n) = xgcd_update(2, u − y, m − n, b_m, a_m)

        # termination condition
        if constant_time:
            iterations = iterations + 1
            end_loop = (iterations >= 1.51 * bitwidth + 1)
        else: end_loop = (a == 0 or b == 0)

    # Step 3: Post-processing

    gcd = a + b; u = u + y; m = m + n

    # account for making a_m, b_m odd before iteration loop
    if (a_0%2 == 0): m = u + m
    elif (b_0%2 == 0): u = u + m

    if gcd < 0: gcd = −gcd; u = −u; m = −m

    return gcd, u, m
```

Listing 1: Our XGCD algorithm (building from the two-bit PM GCD algorithm [YZ86])

While half of the updates in the two-bit PM algorithm directly reduce bits from a or b (when a or b is even), the other half reduce two bits from $a \pm b$ (when a, b are odd). To preserve the relations in Equation 1 in these other updates, we apply the shift-by-two strategy on $u \pm y$, $m \pm n$ instead of individually on u, y, m, n (i.e., substitute u with $u \pm y$ and m with $m \pm n$ in Table 3). Thus, the critical path in this algorithm has two subtractions and one right shift to compute $\frac{m-n-3*a_m}{4}$ as an update for m, which requires one extra subtraction compared to the worst-case updates for m when a or b is even.

Having considered the operations in the iterations loop to find the XGCD, we consider necessary pre- and post-processing steps. Typically, GCD algorithms assume that the inputs have no common factors of two since such factors can be easily removed by shifting before the iterations loop and then shifted back in at the end. Then, at most one input may be even. In this case, we apply a pre-processing step that replaces the even input with the sum of the inputs to ensure inputs to the iterations loop are odd, as done in prior work [YZ86]. After the iterations loop, we add the equations in Equation 1 to get $(u + y) * a_m + (m + n) * b_m = a + b = GCD(a_m, b_m)$ since $a + b$ will ultimately be the GCD. Then, we calculate the Bézout coefficients as $b_a = u + y$ and $b_b = m + n$. If we applied the pre-processing step for odd inputs, we need to replace the even input's Bézout coefficient with the sum of the coefficients at the end to satisfy Equation 1. If the gcd computed is negative (since we can update the smaller number if our $a > b$ approximation is incorrect), we also negate our gcd and coefficient results in the post-processing step.

4.2 Carry-Save Adders

As previewed in Section 3.3.2, carry-save adder (CSA) designs improve the delay of back-to-back additions by removing carry propagation in all intermediate computations and storing the resulting sum in CSA form or redundant binary form. Since CSAs have constant delays, an added benefit is that the clock frequency for such designs will not be sensitive to the input bitwidth. As a result, CSAs have been used for many applications, including cryptography [MMM03, SKN08, RM19]. Prior work has also suggested using CSAs for GCD algorithms [Pur83, YZ86], but we find that using CSAs in practice in an XGCD hardware design surfaces challenges that have not been previously addressed.

We keep all our variables (a, b, u, y, m, n) in CSA form. All constants (a_m, b_m and their multiples) are not kept in CSA form since they do not change, which minimizes the number of adder inputs for the variable updates. Finally, the approximate difference of a and b is not stored in CSA form since we need to know its sign to determine which value to update when both a, b are odd. Fortunately, we use binary logarithms for this approximation, so this value is small and updates do not limit cycle time.

4.2.1 Addressing CSA Challenges

Remaining carry-propagate adds To ensure the carry-propagate adds required in our pre- and post-processing steps (see Section 4.1) do not limit our cycle time, we run these steps at one-quarter of the clock frequency of the system clock. This slower frequency allows these few cycles at the start and end to support the longer carry propagation while keeping the short CSA-defined cycle time as our system clock period.

Incorrect truncated results when shifting When *carry* and *sum* are shifted to the right by r bits (and inherently truncated) in CSA form, we need to efficiently add one to result when the shifted (now lost) bits are not all zeros and a carry-in would have been generated. Unfortunately, we cannot simply set the lowest bit of *carry* or *sum* to one, since these bits can both be one after a shift. Thus, we instead use a half adder to add the shifted *carry* and *sum* to get another (*carry*, *sum*) pair for the same number represented in CSA form, where the lowest bit of the *carry* output will be zero by design. Then, we can set that bit to one when needed and add one XOR gate delay to the critical path.

Table 4: Runtime and area in 16nm technology for 1024-bit XGCD ASIC designs that remove a maximum of different factors of two depending on whether a, b are even or odd.

Max factor of two reduction when *a or b* is *even*	Max factor of two reduction when *a and b* are *odd*	Average Number of Cycles	Cycle Time (ns)	XGCD execution time (ns)	ASIC area (mm^2)
2	2	2210	0.193	427	0.16
4	2	1845	0.218	402	0.21
8	2	1740	0.251	437	0.35
2	4	1450	0.234	339	0.22
4	4	1211	0.247	299	0.28
8	**4**	**1143**	**0.257**	**294**	**0.41**
2	8	1091	0.297	324	0.27
4	8	972	0.320	311	0.33
8	8	937	0.330	309	0.47

Arithmetic right shifts Like prior work [BK85], we approximate $a > b$ when a, b are odd. If the approximation is incorrect, we update the smaller number and our variables become negative. Thus, we need to preserve sign in CSA form (even though it is unknown) when shifting to the right. Earlier work determines that this can be accomplished using the two most significant bits of *carry* and *sum* before and after shifting [TPT06]. From this prior work, we determine balanced equations to preserve sign for a right shift in CSA form to minimize the critical path delay. Physical design tools are unable to automatically simplify chaining this sign-preservation logic when shifting by more bits since this logic is separated by the half adder discussed above. Thus, we further specialize this logic when shifting by two or three to make the worst-case delay equal to two OR gate delays.

4.3 Optimal Reduction of Bits Per Cycle

Stein-based algorithms reduce the number of iterations required by directly removing factors of two or four when possible. The higher factors of two that can be removed per iteration, the fewer iterations required. While removing these factors by shifting is cheap, this increases the number of update choices for u, y, m, n since they may not be divisible by powers of two. This added control logic for the XGCD can increase cycle time if deciding which branch to take becomes more expensive than computing the updates required.

Thus, we explore this hardware design space to quantify the tradeoff between the cycle time and the average and worst-case number of clock cycles required to determine the optimal reduction of bits per cycle. We separately vary the maximum reduction factor for updates when a or b is even and updates when a and b are odd (referred to as even and odd reduction factors) to see which parameterizations yield a net benefit for total execution time, the product of the number of clock cycles and the cycle time.

We update a, b with $\frac{a-b}{2}$ for an odd reduction factor of two since $a - b$ will be even when a, b are odd. Using the process in Section 4.1, we support an odd reduction factor of eight. Our efficient rewriting of the updates requires us to compute $k * b_m$ and $k * a_m$ for $k = 1$ to $k = 7$ for the three-bit reductions and does not increase the critical path delay compared to a design with an odd reduction factor of four. Note that the following (even, odd) maximum reduction factor pairs describe factors of two reduced in prior GCD algorithms: (2, 1) – Stein's, (2, 2) – Purdy's and PM, and (4, 4) – two-bit PM.

Table 4 shows the execution time and area for various reduction factor pairs and all optimizations in Section 4. The longest sequence of operations is updating m, n when a, b are odd. This is the critical path for all the designs but the (8, 8) design, where the critical path becomes the control logic for branching. Finally, since the execution time for designs

with an odd reduction factor of eight does not improve compared to a reduction factor of four, factors of 16 or higher would not be beneficial, due to the longer control logic delays.

We observe that the execution times for these designs are close, especially if the odd reduction factor is the same. Thus, while average number of cycles is reduced when more bits are reduced per cycle, the resulting increase in the cycle time mostly cancels that benefit. Choosing which design to use depends on the target application requirements, namely the number of XGCDs computed and whether XGCD must be constant-time.

These small execution time differences become important when this computation is repeated many times. Since most VDF applications require a large delay, we may need to compute over a million or billion squarings, each of which requires two XGCDs in the NUDUL algorithm. In this context, the five-nanosecond difference between the (4, 4) and (8, 4) becomes critical. **Since this application requires high performance, the fastest design – the (8, 4) design – would be most appropriate.**

For constant-time applications like modular inversion for Curve25519, we must consider the worst-case number of cycles to determine the optimal reduction factors. The two-bit PM algorithm – corresponding to the (4, 4) design – takes a maximum of $1.51n + 1$ cycles for n-bit inputs [YZ86] since every cycle, at least one bit is reduced if a or b is even and two bits are reduced when the values odd (since either $a + b$ or $a - b$ will be divisible by four). The two-bit PM algorithm [YZ86] notes that reducing three bits – corresponding to the (8, 8) design – will not reduce the worst-case number of cycles because there is no guarantee that $a + b$ or $a - b$ will be divisible by eight when a, b are odd (just as a, b may not be divisible by eight when they are even), so these branches may never be taken.

We observe that this logic also applies for the (2, 8), (4, 8), (2, 4), and (8, 4) designs since the only guaranteed transitions remain dividing by two when a, b are even and dividing $a + b$ or $a - b$ by four when a, b are odd. Thus, shifting by more than one bit when a, b are even or more than two bits when a, b are odd is beneficial for optimizing the average execution time but does not add any further value when optimizing the worst-case execution time. In addition, the average execution times for the designs with an odd reduction factor of two is higher than the worst-case execution time for the other designs, so these designs are not competitive in this context. **Thus, the optimal reduction factors for constant-time applications are (2, 4).**

4.4 Termination Condition Carry Propagation

Stein-based algorithms, including the two-bit PM, return the GCD when a or b is equal to zero. However, when a, b are in CSA form, their values are not known (Section 4.2). Obtaining their actual values for this check would require two carry-propagate adds in parallel to find $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$, which negates the benefit of using CSAs. As a minor concern, this operation also requires a large AND-gate tree to check whether all $2 * n$ bits are zero, where n is the input bitwidh. To improve the delay of this check, we investigated two possible approaches for a hardware design.

Sampling Our first approach computes our variable updates per iteration in a clock cycle time, while in parallel, $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$ are computed over x cycles. The true values of a, b are then sampled every x cycles. For 1024-bit inputs, $x = 4$ ensures that these additions do not limit the cycle time. With this approach, we can only check if the termination condition has been satisfied every four cycles and at most require four extra iterations to compute the XGCD. This is a very small overhead: 0.18% to 0.43% of the number of cycles for the (2, 2) to (8, 8) reduction factor designs (Section 4.3).

Approximating variables Our second approach repurposes $\alpha \approx \log_2(a), \beta \approx \log_2(b)$ from the PM algorithm [BK85] in a novel way. These variables are used to approximate $a > b$ in prior work and are updated every cycle by the minimum number of bits a, b will be reduced by. We instead use α, β to determine our termination condition: when α or β is equal to zero, we run one more iteration (to ensure either a or b is zero since α, β can

equal zero when a, b are one) and initiate the post-processing step. This approach avoids the carry-propagation for $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$ since the termination condition no longer checks a, b. In addition, we only check whether $\log_2(1024) * 2 = 20$ bits are zero for α, β instead of 2048 bits for a, b reducing the AND-gate tree delay. However, the α, β approximations can significantly diverge from the true values of $\log_2(a), \log_2(b)$. For example, when a updated with $a - b$, multiple bits could be reduced, but only one is subtracted from α. Thus, checking if α, β are equal to zero instead of a, b adds 150+ cycles for 1024-bit inputs (a 7 to 16% overhead for different reduction factor designs).

We experimented with correcting α, β to be the true values of $\log_2(a), \log_2(b)$ as often as possible. This requires computing 1024-bit carry-propagates adds to get a, b, the absolute values of a, b since they can be negative, and then $\log_2(|a|), \log_2(|b|)$. We can run these operations in parallel to the variable updates and as in the sampling approach, find that these operations can complete every four cycles without limiting our cycle time. While this approach is functional, the sampling approach without α, β requires less computation run in parallel to the iterations loop. Thus, our design uses the sampling approach.

4.5 Minimal Control Overhead

Having sped up the data path, we focus on minimizing the control path delay. We note that our control logic for detecting divisibility by factors of two is not as simple as checking if bits are zero since our variables are in CSA form. To minimize the delay for this logic, we duplicate computation to allow the control signals to arrive as late as possible (late selects) and precompute control signals each cycle for the next cycle's branching decisions.

Late selects Either a, u, m or b, y, n is updated every cycle. The update logic for these sets of values is identical but requires different inputs. Using the same hardware to perform these updates would thus add an extra two-to-one multiplexer before each input compared to dedicating separate hardware for each variable, doubling the number of branches to choose between. Thus, we intentionally duplicate modules to have separate update modules for a, b and u, y, m, n to allow control signals to arrive late in the cycle. We also apply this parallel computation and late select strategy wherever possible within all the update modules, avoiding redundant computation for updates that use the same inputs regardless of the updated variable (e.g. $a \pm b$ in both the update a and b modules).

Precomputing control signals For 1024-bit inputs, our control signals determine the updates for 1028-bit numbers since several extra bits account for carry bits from repeated addition. As a result, these gates have very high fanout. For (4, 4) reduction factor pair design, we originally observed that the critical path was the control path and not the data path due to the 0.15ns of delay from several large buffers in 16nm designs.

To reduce this delay, we compute the control signals that determine which branch to take for the next cycle in parallel with the updates for a, b, u, y, m, n instead of generating the control signals from the generated values in the same cycle. For example, when we update u, we compute whether the updated u for the next cycle will be divisible by factors of two by computing the divisibility of all the possible u update options in parallel. Then, we select the control signal update based on the u update chosen. To similarly compute the divisibility of $a \pm b$, $u \pm y$, and $n \pm m$ for the next cycle, we add a few XORs ($\approx$ 20ps each in 16nm), which is significantly smaller than the 0.15ns large buffer delay. Due to this optimization, the critical path is the data path for our designs, except for the (8, 8) reduction factor pair. In this design, the disproportional increase in the number of branches compared to the increase in computation required makes the control path dominant.

4.6 Extensions

Constant-time To build a constant-time XGCD design, we pad our XGCD algorithm to ensure it always run for its worst-case execution time since running beyond the nominal

Table 5: Critical path operations and our implementations for the NUDUPL algorithm.

Operation (Op)	Implementation	Op Runtime (ns)	Count	Total Runtime (ns)
Addition	Designware Parallel-Prefix Adder	0.159	49	7.8
Multiplication	Toom-3	17.58	51	897
Inverter	Standard cell	0.01	3	0.03
XGCD	Our design with (8, 4) reduction factors	294	2	588
Total				1492

termination condition will still yield the correct answer. Since the inputs to the iterations loop are preprocessed to remove common factors of two, the XGCD will be found when either a or b is zero and the other is some odd value. Then, the algorithm will continually detect the zero value to be even and divide by a factor of two. Since all variables are in CSA form (Section 4.2), it is not clear when they become zero. Thus, we can keep track of the number of cycles and end the iterations loop when that count is equal to the worst-case cycle count, the computation can end. The (2, 4) reduction factor pair is most optimal for constant-time XGCD and requires $1.51 * n + 1$ iterations for n-bit inputs (Section 4.3).

Polynomial XGCD We can use the same algorithmic control flow to find the XGCD between two polynomials with integer coefficients by describing the polynomial equivalents for all the integer XGCD operations required, building from prior work [BK84, BY19]. The polynomial equivalent of reducing factors of two, the smallest prime, is reducing factors of x (x, x^2, x^3, etc.), i.e., reducing the polynomial degree. Then, evenness translates to polynomial divisibility by x. To ensure adding "odd" polynomials guarantees an "even" result (like with integers), we can multiply the polynomials by the other polynomial's constant term to enable cancelling the constant terms in addition [BY19]. For comparisons, we compare polynomial degrees. To find the XGCD, we maintain the relations in Equation 1 each iteration: a_m, b_m are now polynomials, where $a_m = a$ if a is not divisible by x or a/x if a is, and similarly for b_m with b. Then, we initialize $u = 1, m = 0$ and $y = 0, n = 1$, as with integer GCD, and apply the same updates (adds and divisions by x for shifts).

5 Complete Hardware Accelerators for Target Applications

Since XGCD already directly implements constant-time modular inversion (the first Bézout coefficient is the modular inverse), we focus on accelerating the remaining operations required for the NUDUPL algorithm (Section 2.1) in this section.

Our implementations are summarized in Table 5. Since additions on the critical path are in isolation, there is no benefit to using CSAs and we use efficient adders from our synthesis tool library. After the first XGCD, every execution requires a single modular multiplication. Since this operation is not repeated, conventional techniques like Montgomery multiplication (which converts to an intermediate representation) and Barrett Reduction (which precomputes constants) are expensive. Given that this operation minimally contributes to execution time, we use a straightforward approach and compute $x = y * z \pmod a$ as a multiplication ($m_1 = y * z$), division ($d = m_1/a$), another multiplication ($m_2 = d * a$), and a subtraction for ($m_1 - m_2$), all with 1024-bit inputs.

The Karatsuba algorithm [Kar63] is a well-known fast multiplication algorithm that recursively splits multiplication into three smaller ones and a few additions and shifts, until a base case input bitwidth. Implementing this algorithm in hardware has proven popular [EhyMZ+08, vzGS05, ROH17, NdMM03a, NdMM03b, ZST+20]. The Toom-Cook polynomial algorithm [Too63, CA69, BZ07] is more efficient and general. It splits numbers into k parts ($k = 3$ for Toom-3, which we implement) to use as polynomial coefficients,

finds the product of evaluating these polynomials at various points to determine the product polynomial, and then uses the product's coefficients to find the integer product. This algorithm is well-suited for our input size as it is faster than the asymptotically fast algorithms [Sch77] for numbers smaller than $2^{2^{15}}$. However, few *hardware designs* use Toom-Cook [GL18, DLG18] since it requires an expensive hardware operation, division by three. We reduce the complexity of this division by rewriting $\frac{x}{3}$ as $x - \frac{2x}{3}$ as shown in Listing 2. This allows us to calculate each bit of the output, $\frac{x}{3}$, starting from the least significant bit, successively generating the next bit of $\frac{2x}{3}$ to use in the subtraction.

$$
\begin{array}{rccccccc}
 & x & & x_{n-1} & x_{n-2} & \cdots & x_1 & x_0 \\
- & \frac{2x}{3} & & y_{n-1} & y_{n-2} & \cdots & y_1 & 0 \\
\hline
 & \frac{x}{3} & 0 & & y_{n-1} & \cdots & y_2 & y_1
\end{array}
$$

Listing 2: Rewriting $\frac{x}{3}$ as $x - \frac{2x}{3}$ to reduce the complexity of division by three in Toom-3.

The Newton-Raphson [Fly70] and Goldschmidt [Gol64] algorithms are often used for fast division. They initially estimate the reciprocal of the divisor and then iteratively multiply to refine this estimate. Few papers implement division in hardware [HSGJ10, ZST+20]. The NUDUPL critical path requires two divisions with 2048-bit dividends and 1024-bit divisors. Since these divisions constitute a small portion of total execution time, we directly implement the Newton-Raphson algorithm with Toom-3 multiplication. We include the additions and multiplications required for this division under those operations in Table 5.

6 Design Evaluation

We use a vertically integrated methodology spanning cycle-level performance modeling, VLSI-level modeling, and detailed physical design before taping out our chip. We wrote our RTL in Kratos 0.0.33 [Zha], a hardware design language capable of generating SystemVerilog, and built a parameterizable testbench with Fault 3.052 [THS+20] to verify our designs with our Python functional model for various parameterizations (bitwidth, reduction factors, constant time support) and randomly-generated inputs. We use mflowgen 0.3.1 [CTN+21] for our physical design flow, using Synopsys DC 2019.03 for synthesis and Cadence Innovus 19.10.000 for floorplan, power, place, clock tree synthesis, and route. We execute the full physical design flow to build our ASIC and report post-layout signoff numbers.

We evaluate two parameterizations of our ASIC design: (1) 1024-bit XGCD with (8, 4) reduction factors for squaring binary quadratic forms over class groups and (2) 255-bit constant-time XGCD with (2, 4) reduction factors for modular inversion for Curve25519. Both designs are in TSMC 16nm, using SVT, LVT, and ULVT libraries at 0.8V.

Timing Table 6 shows the critical path delay breakdown for the designs. We include technology-agnostic delays in units of inverter fanout-of-4 (FO4) delays [HHWH97], with the 16nm ULVT FO4 delay as 9ps. Since both these designs have an odd reduction factor of four, they have identical critical paths as the same worst-case sequence of operations limit the cycle time. As expected from Section 4.1, the critical path requires three CSAs to add the Bézout coefficient variables and a multiple of b_m or a_m when a, b, and these variables are odd. Then, we need to preserve sign when shifting in CSA form, with a half-adder to adjust incorrectly truncated results. Finally, we choose the updates for our variables using late selects and precompute control signals for the next cycle. Design (2) includes extra inverter and buffer delays compared to Design (1), which are not tied to specific logic and reflect stochastic decisions made by the physical design tools. Thus, our timing is as expected from our design-space exploration. In addition, our designs have similar cycle times despite supporting different input bitwidths due to our use of CSAs.

Area Our Design (1) ASIC is $0.26mm^2$, while our Design (2) ASIC is $0.04mm^2$. The modules to update the Bézout coefficient variables comprise the majority of area (68% and

Table 6: XGCD critical path breakdown for 1024-bit Design (1) and 255-bit Design (2).

Operation	**Design (1) Delay (ns)**	**Design (1) FO4 Inv Delay**	**Design (2) Delay (ns)**	**Design (2) FO4 Inv Delay**
Local clock gating	0.035	3.9	0.018	2
DFF clk to Q	0.040	4.4	0.045	5
Inverter	0	0	0.007	0.8
Add $u + y$: CSA 1	0.039	4.3	0.018	2
Add $u + y$: CSA 2	0.039	4.3	0.031	3.4
Buffer	0	0	0.013	1.4
Add $u + y + 2b_m$: CSA	0.034	3.8	0.030	3.3
Shift in CSA form	0.018	2	0.015	1.7
Late select multiplexers	0.018	2	0.018	2
Precomputing control	0.022	2.4	0.027	3
Total	0.257	28.6	0.220	24.4

Table 7: Comparison of our 1024-bit design to non-constant-time 1024-bit implementations in prior work. NR = not reported. * NR but assumed from other work.

XGCD Design	Technology Node / Platform	Area (mm^2)	Clock Frequency	Cycles	Time (ns)
GNU XGCD C++	Apple M1 | 5nm	-	-	-	10650
[AHAJS16]	Xilinx XC7K70T-2-FBG676 FPGA | 28nm	NR	39.94 MHz	598 *	151777
Our Design (1)	**Xilinx XC7K70T-2-FBG676 FPGA | 28nm**	**-**	**248 MHz**	**1143**	**4600**
[ZTW21]	TSMC 28nm ASIC	2.4	250 MHz	1623	6490
[ZST+20]	TSMC 28nm ASIC	9.9	500 MHz	3000	6000
Our Design (1)	**TSMC 16nm ASIC**	**0.41**	**3.89 GHz**	**1143**	**294**

61% in Designs (1) and (2), respectively), since they have the most update possibilities and we parallelize and duplicate computation for late selects (Section 4.5). We compute the $u \pm y$ and $m \pm n$ updates only in the update u, m modules since they are the same for (u, y) and (m, n). Thus, the u, m modules in Designs (1) and (2) are respectively 2X and 5X bigger than the y, n modules. Since the m, n updates require subtractions with a_m while u, y updates require additions with b_m, the m, n modules are slightly larger.

6.1 Design (1) Comparison

In Table 7, we compare Design (1) to prior software, FPGA, and ASIC designs.

Software We ran the GNU Multiple Precision Arithmetic Library (GMP) XGCD C++ function optimized for large integers on a 2020 MacBook Pro with the M1 chip in 5nm, compiled with g++ and -O3 optimization. We use the standard C++ chrono library with nanosecond precision to profile this code with random 1024-bit inputs. Our 16nm ASIC for Design (1) is 36X faster than the C++. This comparison shows building specialized hardware for the XGCD function for such high-performance applications is advantageous.

FPGA Design (1) is 33X faster and runs at a 6.2X faster clock frequency when synthesized on the same FPGA (Xilinx XC7K70T-2-FBG676) used in the prior 1024-bit FPGA work [AHAJS16] for a direct comparison. We achieve this speedup since this prior work implements Euclid's algorithm while we build from the two-bit PM. The authors in this prior work do mention using CSAs, though no further detail is provided. Note that the FPGA in [AHAJS16] is run at 39 MHz and in 28nm, while the M1 processor is in 5nm is run at 2.3 GHz, so the FPGA execution time is slower than software. Thus, our 16nm

Table 8: Comparison of our constant-time 255-bit XGCD design to constant-time 255-bit implementations in prior work. NR = not reported. * NR but assumed from other work.

XGCD Design	Technology Node / Platform	Area (mm^2)	Clock Frequency	Cycles	Time (ns)
[BY19]	Intel Kaby Lake | 14nm	-	2.3 GHz *	8543	3714
[Por20]	Intel Coffee Lake | 14nm	-	2.3 GHz	6253	2719
[DdPM+21]	Zynq UltraScale+ XCZU7EG FPGA | 16nm	NR	207 MHz	8466	40900
Our Design (2)	**Zynq UltraScale+ XCZU7EG FPGA | 16nm**	**-**	**447 MHz**	**386**	**863**
Our Design (2)	**TSMC 16nm ASIC**	**0.059**	**4.55 GHz**	**386**	**85**

ASIC for Design (1) is 211X faster than [DdPM+21] when technology-scaled to 16nm.

ASIC Design (1) is 8X faster and 12X smaller than the state-of-the-art 1024-bit XGCD ASIC [ZST+20], after technology-scaling their work to 16nm for fair comparison. We achieve this speedup since we reduce both the cycle count and cycle time compared to this prior work, which implements the GMP XGCD algorithm. This requires 3000 cycles on average, while Design (1) reduces the number of cycles required by over 60%. In addition, the GMP algorithm uses just most significant bits for some division steps in Euclid's algorithm, which we estimated will not translate into faster execution times compared to building from the two-bit PM algorithm [YZ86] instead (Section 3). Thus, our design uses faster operations (carry-save adds and shifts) and runs at a 7.8X faster clock frequency. Finally, we note that [ZST+20] provides synthesis results, which uses wire delay approximations. We instead execute the full physical design flow, which includes the final layout post place-and-route and has signoff-quality RC extraction for wire delay.

6.2 Design (2) Comparison

In Table 8, we compare Design (2) to prior software and FPGA designs. We are not aware of any prior ASICs and believe our work is the first ASIC for constant-time XGCD.

Software Design (2) achieves a 31X speedup over the state-of-the-art software implementation [Por20], which builds from Stein's algorithm as we do. Pornin focuses on equalizing the time for each iteration, which happens naturally in hardware with a standard clock frequency. Pornin also uses approximations of large values for faster computation, which we achieve in the hardware context by storing values in CSA form (Section 4.2) and sampling values for our termination condition (Section 4.4). The speedup achieved shows that building an ASIC for *constant-time* XGCD is also worth the designer time and effort.

Despite our lower cycle counts in Table 8, we do not expect our algorithm to be significantly faster than [Por20] and [BY19] in software. Cycles correspond to different operations in hardware and software. Our hardware design completes all the operations for an iteration in one clock cycle, resulting in a low cycle count equal to the iteration count. However, software designs are constrained by the processor's ISA, so the cycle count closely corresponds to the number of instructions. Thus, in software, each iteration of our algorithm would expand to many instructions emulating large-integer addition, resulting in larger cycle counts closer to [Por20]. There is also no CSA instruction in most processor ISAs, so the speedup from using CSAs would not translate to software.

FPGA Design (2) is 46X faster and runs at a 2X faster clock frequency when synthesized on the same FPGA (Zynq UltraScale+ XCZU7EG) used in the prior constant-time XGCD FPGA design [DdPM+21] for a direct comparison. We achieve this speedup since our critical path consists of fast carry-save adds and shifts while this prior work implements the Bernstein-Yang algorithm, which requires modifying fractions and efficiently dividing.

In addition, we use CSAs to eliminate the carry-propagation for large-integer additions in in the iterations loop while the prior work pipelines this delay. Finally, since we specify that the operations required in one XGCD iteration complete in one clock cycle while the prior work's cycle count is comparable to the Bernstein-Yang count in software [BY19], we reduce the cycle count by over 95%. Note that the FPGA in [DdPM+21] is run at 207 MHz, while the Intel processor in [Por20] is run at 2.3 GHz, so the FPGA execution time is slower than software. Thus, our 16nm ASIC for Design (2) is 470X faster than [DdPM+21].

6.3 Unified ASIC Design

We can also use the 1024-bit Design (1) for constant-time modular inversion by setting the constant time configuration to true and the bitwidth to 255. The bitwidth is not specified for functionality but to limit the worst-case number of cycles to the number required for 255-bit inputs rather than 1024-bit inputs. Our Design (1) ASIC is 27X faster than Pornin's work [Por20] and 412X faster than the prior FPGA design [DdPM+21] (26X faster with the 1024-bit Verilog synthesized on the same FPGA used in [DdPM+21]). Since ASICs can be expensive, achieving high performance with the same hardware unit for several applications varying in bitwidth (1024 versus 255 bits) and security requirements (constant-time versus not) is advantageous and a key benefit of our work.

6.4 Squaring Comparison

We ran over one million trials of the Chia Network's C++ to find that their NUDUPL implementation takes an average of $21us$ per squaring and partial reduction. If we accelerate only the two XGCD computations required by using our 1024-bit XGCD Design (1), we reduce the $0.91 * 21us = 19.11us$ XGCD execution time to $2 * 294ns = 588ns$ and speed up the full algorithm by 8.5X, which is close to 9.1X, the best speedup possible if accelerating only the XGCD. By accelerating the remaining operations on the critical path (Section 5), we can execute the full computation in $1.5us$, achieving a 14X speedup over the C++.

We believe our work is the first to accelerate the NUDUPL algorithm in hardware. One prior paper accelerates a different squaring algorithm for 1024-bit inputs in 28nm [ZST+20]. However, comparing our results to this prior paper is not a fair comparison because their work considers solely squaring without any reduction, while the NUDUPL algorithm requires more computation since it inherently squares and partially reduces the outputs. In terms of core components, our XGCD design is faster compared to this prior work (Section 6.1) and we implement the more efficient Toom-3 algorithm instead of the Karatsuba algorithm.

7 Conclusion

Fast XGCD implementations are increasingly important as the cryptography community investigates applications that are dominated by the XGCD, including verifiable delay functions based on squaring over class groups and modular inversion for elliptic curve cryptography. However, existing XGCD hardware literature is sparse, providing point solutions for specific applications and focusing narrowly on division-based algorithms building from Euclid. This work presents an algorithm and hardware design space exploration that reveals that building from Stein's subtraction-based algorithm results in faster hardware. Our hardware design uses a redundant representation to run at extremely high 4+GHz clock frequencies and applies further optimizations to achieve state-of-the-art performance for average and worst-case execution. Our 16nm ASIC is the first high-performance ASIC for XGCD that is also capable of constant-time evaluation, and is 8X faster than the state-of-the-art ASIC for 1024-bit XGCD and 31X faster than

the state-of-the-art software for constant-time 255-bit XGCD. These results significantly advance XGCD performance for cryptographic applications.

References

[AHAJS16] Qasem Abu Al-Haija, Monther Al-Ja'fari, and Mahmoud Smadi. A comparative study up to 1024 bit euclid's gcd algorithm fpga implementation and synthesizing. In *2016 5th International Conference on Electronic Devices, Systems and Applications (ICEDSA)*, pages 1–4. IEEE, 2016.

[BB87] Adam W Bojanczyk and Richard Peirce Brent. A systolic algorithm for extended gcd computation. *Computers & Mathematics with Applications*, 14(4):233–238, 1987.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 2018.

[BBF18] Dan Boneh, Benedikt Bünz, and Ben Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[BK84] Richard P. Brent and H. T. Kung. Systolic vlsi arrays for polynomial gcd computation. *IEEE Transactions on Computers*, C-33(8):731–736, 1984.

[BK85] R. P. Brent and H. T. Kung. A systolic algorithm for integer gcd computation. In *1985 IEEE 7th Symposium on Computer Arithmetic (ARITH)*, pages 118–125, 1985.

[Bue89] Duncan A Buell. *Binary quadratic forms: classical theory and modern computations*. Springer Science & Business Media, 1989.

[BY19] Daniel J Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 340–398, 2019.

[BZ07] Marco Bodrato and Alberto Zanoni. Integer and polynomial multiplication: Towards optimal toom-cook matrices. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 17–24, 2007.

[CA69] Stephen A Cook and Stål O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969.

[CCC93] Henri Cohen, Henry Cohen, and Henri Cohen. *A course in computational algebraic number theory*, volume 8. Springer-Verlag Berlin, 1993.

[Col80] GE Collins. Lecture notes on arithmetic algorithms. *University of Wisconsin*, 1980.

[CTN+21] Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen, Mark Horowitz, Priyanka Raina, and Christopher Torng. Enabling reusable physical design flows with modular flow generators. *arXiv preprint arXiv:2111.14535*, 2021.

[DdPM+21] Sanjay Deshpande, Santos Merino del Pozo, Victor Mateu, Marc Manzano, Najwa Aaraj, and Jakub 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)*, pages 122–129. IEEE, 2021.

[DG11] Ulrich Daepp and Pamela Gorkin. Fermat's little theorem. In *Reading, Writing, and Proving*, pages 315–323. Springer, 2011.

[DGS20] Samuel Dobson, Steven D Galbraith, and Benjamin Smith. Trustless groups of unknown order with hyperelliptic curves. *IACR Cryptol. ePrint Arch.*, 2020:196, 2020.

[DLG18] Jinnan Ding, Shuguo Li, and Zhen Gu. High-speed ecc processor over nist prime fields applied with toom–cook multiplication. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(3):1003–1016, 2018.

[EhyMZ+08] wajih El hadj youssef, Mohsen Machhout, Medien Zeghid, Belgacem Bouallegue, and Rached Tourki. Efficient hardware architecture of recursive karatsuba-ofman multiplier. In *2008 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, pages 1–6, 2008.

[FaPKL+] Pierre-Alain Fouque, Jeffrey Hoffstein annd Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. https://falcon-sign.info/.

[FdCBM05] Mateus Fonseca, Eduardo da Costa, Sergio Bampi, and José Monteiro. Design of a radix-2m hybrid array multiplier using carry save adder format. In *Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 172–177, 2005.

[Fly70] Michael J Flynn. On division by functional iteration. *IEEE Transactions on Computers*, 100(8):702–706, 1970.

[GL18] Zhen Gu and Shuguo Li. A division-free toom–cook multiplication-based montgomery modular multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(8):1401–1405, 2018.

[Gol64] Robert E Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964.

[HAS21] Benjamin Salling Hvass, Diego F Aranha, and Bas Spitters. High-assurance field inversion for curve-based cryptography. *IACR Cryptol. ePrint Arch.*, 2021:549, 2021.

[HHWH97] David Harris, Ron Ho, Gu-Yeon Wei, and Mark Horowitz. The fanout-of-4 inverter delay metric. *Unveröffentlichtes Manuskript: http://odin. ac. hmc. edu/harris/research/FO4. pdf*, 1997.

[HM00] Safuat Hamdy and Bodo Möller. Security of cryptosystems based on class groups of imaginary quadratic orders. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 234–247. Springer, 2000.

[HSGJ10] Andreas Habegger, Andreas Stahel, Josef Goette, and Marcel Jacomet. An efficient hardware implementation for a reciprocal unit. In *2010 Fifth IEEE International Symposium on Electronic Design, Test Applications*, pages 183–187, 2010.

[Jeb93a] Tudor Jebelean. A generalization of the binary gcd algorithm. In *Proceedings of the 1993 international symposium on Symbolic and algebraic computation*, pages 111–116, 1993.

[Jeb93b] Tudor Jebelean. Improving the multiprecision euclidean algorithm. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, pages 45–58, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Jeb95] Tudor Jebelean. A double-digit lehmer-euclid algorithm for finding the gcd of long integers. *Journal of Symbolic Computation*, 19(1):145–157, 1995.

[JLL+15] Song Jia, Shigong Lyu, Xiayu Li, Li Liu, and Yandong He. Simplified carry save adder-based array multiplier scheme and circuits design. *International Journal of Circuit Theory and Applications*, 43(9):1226–1234, 2015.

[JvdP02] Michael J. Jacobson and Alfred J. van der Poorten. Computational aspects of nucomp. In Claus Fieker and David R. Kohel, editors, *Algorithmic Number Theory*, pages 120–133, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[Knu70] Donald E Knuth. The analysis of algorithms. In *Actes du congres international des Mathématiciens*, volume 3, 1970.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[Lam44] Gabriel Lamé. *Note sur la limite du nombre des divisions dans la recherche du plus grand commun diviseur entre deux nombres entiers*. 1844.

[Leh38] Derrick H Lehmer. Euclid's algorithm for large numbers. *The American Mathematical Monthly*, 45(4):227–233, 1938.

[Lon19] Lipa Long. Binary quadratic forms. https://github.com/Chia-Network/vdf-competition/blob/main/classgroups.pdf, 2019.

[MH94] Bohdan S Majewski and George Havas. The complexity of greatest common divisor computations. In *International Algorithmic Number Theory Symposium*, pages 184–193. Springer, 1994.

[Mil85] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[MMM03] Ciaran Mclvor, Maire McLoone, and John V McCanny. Fast montgomery modular multiplication and rsa cryptographic processor architectures. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 1, pages 379–384. IEEE, 2003.

[Mol97] Richard A Mollin. *Fundamental number theory with applications*. Crc Press, 1997.

[Möl08] Niels Möller. On schönhage's algorithm and subquadratic integer gcd computation. *Mathematics of Computation*, 77(261):589–607, 2008.

[NdMM03a] Nadia Nedjah and Luiza de Macedo Mourelle. Fast less recursive hardware for large number multiplication using karatsuba-ofman's algorithm. In *International Symposium on Computer and Information Sciences*, pages 43–50. Springer, 2003.

[NdMM03b] Nadia Nedjah and Luiza de Macedo Mourelle. A reconfigurable recursive and efficient hardware for karatsuba-ofman's multiplication algorithm. In *Proceedings of 2003 IEEE Conference on Control Applications, 2003. CCA 2003.*, volume 2, pages 1076–1081. IEEE, 2003.

[nis17] Post-quantum cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography, 2017.

[NLRC10] JAM Naranjo, JA López-Ramos, and LG Casado. Applications of the extended euclidean algorithm to privacy and secure communications. In *Proc. of 10th international conference on computational and mathematical methods in science and engineering*, pages 702–713, 2010.

[Pie18] Krzysztof Pietrzak. Simple verifiable delay functions. In *10th innovations in theoretical computer science conference (itcs 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[Por18] Thomas Pornin. Bearssl: a smaller ssl/tls library. https://bearssl.org/, 2018.

[Por20] Thomas Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint Archive, Report 2020/972, 2020. https://ia.cr/2020/972.

[Pur83] George B Purdy. A carry-free algorithm for finding the greatest common divisor of two integers. *Computers & Mathematics with Applications*, 9(2):311–316, 1983.

[PW02] Victor Y Pan and Xinmao Wang. Acceleration of euclidean algorithm and extensions. In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 207–213, 2002.

[Ras17] Bahram Rashidi. A survey on hardware implementations of elliptic curve cryptosystems. *arXiv preprint arXiv:1710.08336*, 2017.

[RM19] Debapriya Basu Roy and Debdeep Mukhopadhyay. High-speed implementation of ecc scalar multiplication in gf (p) for generic montgomery curves. *IEEE transactions on very large scale integration (VLSI) systems*, 27(7):1587–1600, 2019.

[ROH17] Ciara Rafferty, Máire O'Neill, and Neil Hanley. Evaluation of large integer multiplication methods on hardware. *IEEE Transactions on Computers*, 66(8):1369–1382, 2017.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RSPR11] N Ravi, Y Subbaiah, T Jayachandra Prasad, and T Subba Rao. A novel low power, low area array multiplier design for dsp applications. In *2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies*, pages 254–257. IEEE, 2011.

[Sch71] Arnold Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica*, 1(2):139–144, 1971.

[Sch77] Arnold Schönhage. Schnelle multiplikation von polynomen über körpern der charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977.

[Sch91] Arnold Schönhage. Fast reduction and composition of binary quadratic forms. In *Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, pages 128–133, 1991.

[SKN08] 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 *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 44–51. IEEE, 2008.

[SKS09] Raminder Preet Pal Singh, Parveen Kumar, and Balwinder Singh. Performance analysis of 32-bit array multiplier with a carry save adder and with a carry-look-ahead adder. *International Journal of Recent Trends in Engineering*, 2(6):83, 2009.

[Sor94] Jonathan Sorenson. Two fast gcd algorithms. *Journal of Algorithms*, 16(1):110–144, 1994.

[Sor95] Jonathan Sorenson. An analysis of lehmer's euclidean gcd algorithm. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, ISSAC '95, page 254–258, New York, NY, USA, 1995. Association for Computing Machinery.

[SRC20] M Siddhartha, Jelwin Rodriques, and BR Chandavarkar. Greatest common divisor and its applications in security: Case study. In *2020 International Conference on Interdisciplinary Cyber Physical Systems (ICPS)*, pages 51–57. IEEE, 2020.

[Ste67] Josef Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[SZ04] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *International Algorithmic Number Theory Symposium*, pages 411–425. Springer, 2004.

[THS+20] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, et al. fault: A python embedded domain-specific language for metaprogramming portable hardware verification components. In *International Conference on Computer Aided Verification*, pages 403–414. Springer, 2020.

[Too63] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[TPT06] A.F. Tenca, S. Park, and L.A. Tawalbeh. Carry-save representation is shift-unsafe: the problem and its solution. *IEEE Transactions on Computers*, 55(5):630–635, 2006.

[TY00] Klaus Thull and Chee Yap. A unified approach to fast gcd algorithms for polynomials and integers: Technical report from fachbereich mathematik, frie universitaet berlin. In *Fundamental problems in algorithmic algebra*, pages Chapter–2. Oxford University Press, 2000.

[vzGS05] Joachim von zur Gathen and Jamshid Shokrollahi. Efficient fpga-based karatsuba multipliers for polynomials over f2. In *International Workshop on Selected Areas in Cryptography*, pages 359–369. Springer, 2005.

[Web95] Kenneth Weber. The accelerated integer gcd algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 21(1):111–122, 1995.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.

[WTM05] Kenneth Weber, Vilmar Trevisan, and Luiz Felipe Martins. A modular integer gcd algorithm. *Journal of Algorithms*, 54(2):152–167, 2005.

[XGW+17] Sen Xu, Haihua Gu, Lingyun Wang, Zheng Guo, Junrong Liu, Xiangjun Lu, and Dawu Gu. Efficient and constant time modular inversions over prime fields. In *2017 13th International Conference on Computational Intelligence and Security (CIS)*, pages 524–528, 2017.

[YZ86] D. Y. 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*, SYMSAC '86, page 82–84, New York, NY, USA, 1986. Association for Computing Machinery.

[Zha] Keyi Zhang. Kratos: Debuggable Hardware Generator. https://github.com/Kuree/kratos.

[ZST+20] Danyang Zhu, Yifeng Song, Jing Tian, Zhongfeng Wang, and Haobo 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)*, pages 137–140, 2020.

[ZTW21] Danyang Zhu, Jing Tian, and Zhongfeng Wang. Low-latency architecture for the parallel extended gcd algorithm of large numbers. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.