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

Anonymous Submission

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 the computationally dominant operation for two applications of increasing interest: verifiable delay functions that perform squaring computations within class groups of binary quadratic forms and constant-time modular inversion for elliptic curve cryptography. Most prior work has focused on fast software implementations, and very few works have investigated hardware acceleration. All previous hardware literature has focused narrowly on variants of Euclid's algorithm following the approach used in optimized software. However, our work observes that XGCD hardware can perform significantly faster by adopting variants of Stein's binary GCD algorithm instead. In this work, we conduct a detailed large-integer XGCD design space exploration to quantify the tradeoffs between Euclid- and Stein-based algorithms for hardware acceleration. We then explore new algorithm and hardware optimizations resulting in an XGCD hardware accelerator that is flexible and efficient, supports fast and constant-time evaluation, and is easily extensible for polynomial GCD. Our ASIC in TSMC 16 nm calculates the XGCD of 1024-bit inputs in 294ns (36 to 211X speedup) and constant-time XGCD of 255-bit inputs for inverses in the field of integers modulo the prime $2^{255} - 19$ in 87ns (31 to 470X speedup) over the state of the art.

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]. GCD algorithms repeatedly apply GCD-preserving transformations, primarily building from Stein's binary GCD algorithm [Ste67, Pur83, BK85, YZ86, Jeb93a, Sor94, 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 also uses this subtraction property but subtracts as many multiples of the smaller input as possible: $GCD(a, b) = GCD(\min(a, b), \max(a, b) \bmod \min(a, b))$.

Among the work derived from Stein's algorithm, the Purdy algorithm [Pur83] 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 determining $\min(a, b)$. Note that $a \pm b$ will be even when a, b are odd. The Plus-Minus (PM) algorithm [BK85] further avoids large-integer comparisons by approximating the binary logarithm of the difference between the two numbers. The two-bit PM algorithm [YZ86] avoids more comparisons by duplicating cases in the PM algorithm and removing two factors of two in a single iteration.

Among the work derived 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 to present efficient techniques to detect when approximate division based on the MSBs is correct, further minimizing the number of large-integer divisions required [Col80, Jeb93b, Jeb95, Sor95].

Other developments in GCD algorithms focus on subquadratic algorithms that are asymptotically fast since they are based on subquadratic multiplication. The first subquadratic GCD algorithm was the Knuth-Schönhage algorithm [Knu70, Sch71]. This algorithm 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 more recent constant-time approach [BY19] and a binary recursive approach based on Stein's algorithm [SZ04].

Until recently, there has not been much other work on fast GCD algorithms or 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 XGCD algorithms and implementations:

1. Interest in squaring binary quadratic forms over class groups [Wes19] as a verifiable delay function (VDF) [BBBF18], computation for which XGCD is the bottleneck
2. Realization that constant-time implementations of XGCD can still be faster compared to Fermat's method for use in modular inversion [BY19, Por20]

We further explain these applications in Section 2. Since these developments, there have been four relevant papers in the XGCD hardware acceleration space that we are aware of. The first pair of works present a low-latency XGCD ASIC design for 1024-bit integers, targeting use for squaring binary quadratic forms over a class group [ZTW21], and an ASIC for the full squaring computation that utilizes a different XGCD implementation [ZST$^+$20]. The other two works present FPGA implementations for Euclid's algorithm for 1024-bit inputs [AHAJS16] and the recent Bernstein-Yang algorithm [BY19] for constant-time modular inversion [DdPM$^+$21]. All these papers provide point solutions to improve either average runtime (for squaring binary quadratic forms) or worst-case runtime (for constant-time modular inversion) by building from Euclid-based algorithms (citing its low number of cycles and efficient software implementations). With the exception of [AHAJS16], these papers all claim to be the first hardware paper for their target applications.

In this paper, we make three key observations. First, performance fundamentally depends on both cycle count and cycle time, so purely targeting algorithms that minimize cycle count, as prior work has done, can lead to non-optimal solutions. Second, for similar reasons, algorithmic decisions that increase efficiency in software may not translate to efficiency in hardware. Third, although individual applications may favor different points in an XGCD design space, a single unified pareto-optimal design that can efficiently support multiple applications is desirable given the cost of building an ASIC.

Leveraging these observations, we create an efficient parameterizable hardware architecture and conduct a detailed large-integer XGCD design space exploration to quantify tradeoffs between Euclid- and Stein-based algorithms. Despite the lower number of cycles for Euclid-based algorithms and the use of division approximations to improve cycle time, we find that using redundant representations for back-to-back addition dramatically reduces the cycle time for Stein-based algorithms, resulting in faster average runtimes (Section 3). Interestingly, this approach also works well for constant-time applications that rely on the worst-case runtime instead. We then consider the value of further algorithmic and hardware optimizations (Section 4) for minimizing average versus worst-case runtime.

Table 1: Summary of application case studies 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
SOTA algorithm	NUDUPL [JvdP02]	Optimized Stein's [Por20]
# of XGCDs in SOTA	2	1
XGCD % of SOTA runtime	91%	100%
XGCD input bitwidth	1024+	255
Requires minimal XGCD	Yes for 2nd XGCD	No
GCD = 1	Yes for 1st XGCD	Yes
Other XGCD approaches	[Lon19]	Bernstein-Yang [BY19]
Non-XGCD approaches	N/A	Fermat's Little Theorem
XGCD hardware work	[ZST+20, ZTW21]	[DdPM+21]

Our resulting 16nm ASIC is 8X faster than the state-of-the-art (SOTA) ASIC and 36X faster than optimized C++ run on Apple's 5nm M1 processor. These results enable squaring binary quadratic forms 14X faster than C++ on the M1 and computing modular inverses for Curve25519 31 to 470X faster compared to SOTA FPGA and software implementations.

2 Applications

In this paper, 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. We choose these applications to represent two distinct spaces of application requirements. For verifiable delay functions, the inputs are publicly known, and the goal is to reduce the average runtime for fast XGCD. For modular inversion, the input is secret, so the goal is reduce worst-case runtime (i.e., constant-time execution to protect against timing side channel attacks). We summarize these applications in Table 1.

2.1 Verifiable Delay Functions (VDFs)

A VDF is a cryptographic primitive that requires a fixed amount of sequential work to be evaluated despite available parallelism but is still efficiently verifiable [BBBF18]. As a trapdoor function with fast verification but slow evaluation, VDFs are useful for adding delays throughout 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 to compute $f(x) = x^{2^T}$ [BBF18]. The Chia Network chose to incorporate [Wes19]'s construction of squaring binary quadratic forms[1] over a class group as a VDF in their blockchain design. The company has hosted several competitions for fast software implementations for this computation. Both the Chia Network reference and the competition winner chose to implement the SOTA NUDUPL algorithm [JvdP02]. This algorithm not only computes this squaring, but also partially reduces the output values to help with a later reduction step that ensures values stay within a certain size.

[1]We refer the reader to Buell's textbook [Bue89] for detail on binary quadratic forms.

Table 2: NUDUPL algorithm software profiling with 1024-bit inputs.

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

Profiling the operations required for the NUDUPL algorithm with 1024-bit inputs shows that XGCD is computationally dominant, requiring 95% of the total runtime (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 the standard C++ chrono library with nanosecond precision. The algorithm takes one of two branches 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 critical result is finding any pair of Bézout coefficients. We observe that the partial XGCD can be replaced by a full XGCD that does not terminate early and still return 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, only minimal pairs guarantee that the absolute values of the coefficients are less than the absolute values of the inputs divided by the GCD. Euclid's algorithm always returns such a pair, while Stein's-based algorithms may need few additional iterations or a final correction to produce minimal results.

Understanding how much dedicated hardware can accelerate this algorithm helps determine the security level needed to guarantee a certain amount of time has passed with the sequential work in the VDF evaluation, informing the number of squarings required and the minimum input bitwidth. Thus, high performance is the primary objective for VDF solutions. Furthermore, VDFs require sequential work, so area and power consumption are lesser concerns since not much computation can be done in parallel. Finally, since there is a verification step, VDF inputs are not secret and timing attacks are not a security concern, so this application does not require constant-time evaluation. Thus, it is beneficial to minimize the average XGCD runtime even if the worst-case runtime does not improve.

Since XGCD dominates runtime, there is interest in high-performance solutions, and only few prior works consider XGCD hardware acceleration outside of a constant-time context [ZST+20, ZTW21, AHAJS16], we use this application as a case study to demonstrate the effectiveness of our XGCD design. To further ensure our XGCD speedup translates well into overall squaring speedup, we investigate accelerating the other operations required, which consist of one modular multiplication and various additions, multiplications, and divisions. We describe our hardware for these large-integer arithmetic operations and the resulting runtime for this algorithm Section 5. We find that our XGCD algorithm and design coupled with efficient large-integer multiplication and division algorithm implementations result in an efficient accelerator for this squaring. To our knowledge, our work is the first paper considering hardware acceleration for the NUDUPL algorithm.

Finally, we note that there are varying reports on a reasonable input bitwidth for applications using class-group-based VDFs, ranging from 833 bits [HM00] to 3000+ bits [DGS20]. Since the Chia Network uses 1024-bit inputs and recent prior work has done the same [ZST+20, ZTW21], we evaluate 1024-bit XGCD as well. Note that our hardware implementation is parameterizable and can easily generate the larger bitwidth hardware, and our use of redundant binary representation (Section 4.2) makes the cycle time of our hardware relatively independent of the input bitwidth, as shown in Section 6.

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 these applications, there are values that need to be kept a secret: in RSA, the secret key is generated by inverting the public key and in ECC, the value that needs 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, making such systems invulnerable to timing attacks. In the case of ECC, computing the modular inverse needs to 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, one of which is modular inversion with modulus p. This modular inversion is the most time-consuming field operation needed. As a result, many ECC implementations use different coordinate systems such as projective coordinates for most of the computation to avoid inversions, resulting in one inversion at the end [Ras17].

There are two common approaches to find the modular inverse when the modulus is prime. The first method is Fermat's little theorem (FLT) [DG11], which states that $x^{-1} = x^{p-2} \mod p$). 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 with the Bézout coefficient associated as x as 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 more often used to find modular inverses because it could more efficiently be implemented with a constant-time execution [XGW+17] compared to constant-time XGCD implementations. In 2019, the Bernstein-Yang algorithm showed that it is possible for constant-time XGCD to be faster than FLT by designing a subquadratic XGCD algorithm building from Lehmer's algorithm [BY19]. In 2020, Pornin instead optimized Stein's algorithm to build a software implementation faster than the Bernstein-Yang algorithm for computing inverses mod $2^{255} - 19$ on recent 64-bit x86 CPUs [Por20]. Pornin's implementation is used for the key pair generation of (1) RSA in the BearSSL library [Por18] (a C implementation of the SSL/TLS protocol) and (2) Falcon [FaPKL+] (a cryptographic signature algorithm part of the NIST Post-Quantum Cryptography Project [nis17]). In 2021, a software implementation of the Bernstein-Yang algorithm was shown to be up to 2.5X faster than FLT implementations in most cases and was incorporated in the MirageOS unikernel operating system [HAS21]. However, this work is 3.8X slower than Pornin's results for Curve25519 [HAS21], so Pornin's work remains the state of the art for computing inverses mod $2^{255} - 19$ in software.

The only hardware paper on constant-time XGCD that we are aware of 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 runtime is slower than the software records [BY19, Por20] since the FPGA design is run at a much lower frequency (207 MHz) compared to frequency of the Intel processors (2.3 GHz) in the software papers, and the FPGA is built in a 16nm node while the processors are built in 14nm.

These recent papers show the fast adoption of XGCD-based modular inversion, particularly with the modulus $2^{255} - 19$. We observe that all but Pornin's approach uses Euclid-based algorithms to compute the XGCD. Given the growing interest in constant-time XGCD implementations, we conduct a detailed design space exploration to determine which algorithmic approach is more suitable for high performance with hardware acceleration.

3 Performance Analysis of Extended GCD Algorithms

The existing works in Section 2 each represent point solutions in the space of XGCD hardware acceleration. In this section, we take a more holistic view of the design space over multiple axes, including the target platform (software versus hardware), the impact of choosing different algorithmic families (Euclid versus Steins), and the requirements of our two application spaces (minimizing average-case performance versus worst-case constant-time performance). We end by selecting the family of XGCD algorithms that offers the most potential for fast hardware acceleration.

3.1 Software versus Hardware Platform

In both software and hardware contexts, computation time is the product of cycle count and cycle time. This relationship remains true for both optimized software (with CPU cycle count and CPU clock frequency) and custom hardware implementations (with accelerator cycle count and accelerator clock frequency). However, software and hardware developers view these two quantities in different ways. From the perspective of a software developer, CPU clock frequency is fixed within a predetermined range and not under the developer's control. There is a limit to the fastest frequency at which a CPU can clock. As a result, fast software algorithms optimize for cycle count but largely ignore cycle time. On the other hand, hardware can be designed aggressively for extremely short cycle times. This opens the opportunity to select algorithms with higher cycle counts but far simpler operations, resulting in a faster overall computation time.

As described in Section 2, most prior work has focused on optimized software algorithms. To consider optimal XGCD in the hardware context, the remainder of this section evaluates XGCD algorithmic families in the context of both cycle count and cycle time.

3.2 Families of Extended GCD Algorithms

The two major families of XGCD algorithms that we consider include the variants of Euclid's algorithm and Stein's algorithm. While we also consider asymptotically fast sub-quadratic GCD algorithms described in Section 1, for the size of the input bitwidths in our applications (1024 bits for squaring binary quadratic forms, and 255 bits for computing inverses mod $2^{255} - 19$), existing literature with detailed profiling already strongly suggests that Euclid's and Stein's algorithms are faster [Möl08]. As a result, we exclude this family from our exploration. To quantify the tradeoffs between Euclid- and Stein-based algorithms, we compare the number of iterations required for both classes of algorithms in the average case and in the worst case (i.e., constant time). We will then estimate the cycle times of optimal hardware implementations to determine that which family of algorithms is more promising for faster overall computation time.

3.3 Comparing the Number of Iterations

Average Number of Iterations Required We generate uniform random 1024-bit inputs for various GCD algorithms with our functional models in Python and find that, as expected, Euclid's algorithm requires the fewest average number of iterations: 598. Since Stein's algorithm is reduces only a factor of two each cycle, it requires 3.6X the number of iterations. The PM algorithm requires even more iterations compared to Stein's algorithm since it can incorrectly approximate whether a or b is larger. The two-bit PM algorithm reduces more bits each iteration and requires only twice the number of iterations required for Euclid's algorithm. Since Euclid-based algorithms require fewer iterations but more expensive operations compared to Stein-based algorithms, prior work has focused reducing the time per iteration (and not iteration count) for Euclid-based algorithms. Thus, we find that most optimized Euclid algorithms all still require 598 average cycles [Leh38, Sor95].

Worst-case Number of Iterations Required We next consider the number of iterations required for these algorithms in the worst-case. For Euclid's algorithm, the maximum number of iterations is $5\log(\min(a, b))$ [Mol97], when the inputs are Fibonacci numbers [Lam44]. For 1024-bits, this evaluates to 1541 iterations. From the Stein-based algorithms, we observe that the original Stein's, PM, Purdy, and two-bit PM algorithm are all able to reduce one bit when a or b is even, but only the two-bit PM algorithm is able to reduce two bits instead of just one 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 out of these Stein-based algorithms at $1.51 * \log_2(\min(a, b)) + 1$ iterations [YZ86]. Interestingly, this evaluates to 1547 iterations, which is very close to Euclid's 1541 iterations. In general, these equations closely track each other for the range of bitwidths we are interested in and the maximum number of iterations for the two-bit PM case is barely above that required for Euclid's algorithm. Given the simplicity of operations in the two-bit PM algorithm compared to those in Euclid's algorithm, the cycle time for the two-bit PM algorithm will likely be faster than that for Euclid's algorithm, as we estimate below. Since both require essentially the same worst-case number of cycles, the two-bit PM algorithm will likely yield faster runtimes for constant-time implementations and offers a faster starting point.

3.4 Comparing the Cycle Time

We use 1024-bit inputs for the below estimates. To report technology-agnostic delays, we report our cycle time estimates in carry-save adder (CSA) delays, with CSAs explained briefly below and in more detail in Section 4.2. In terms of basic logic gates, one CSA delay is roughly the delay of two XOR gates.

Two-bit PM Cycle Time Estimate In the average case, the two-bit PM algorithm [YZ86] (Listing 1) requires twice the number of cycles as Euclid-based algorithms. Thus, the two-bit PM algorithm could be faster if its hardware critical path, which sets the clock cycle time, is less than half of the critical path of the Euclid algorithm. We find this to be the case if the two-bit PM hardware uses CSAs to perform the 1000+ 1024-bit additions required. CSAs require a constant delay independent of bitwidth. These adders output numbers in redundant binary or CSA form, which uses two bits to represent one bit of the original value, as explained in Section 4.2. To compute XGCD with the two-bit PM algorithm (Section 4.1), the worst-case chain of operations requires two additions between two values in CSA form and one constant. CSAs take in three inputs, so we estimate this delay as the delay for three carry-save adds. We multiply this cycle time delay estimate by the average number of cycles to estimate the total runtime as 3588 CSA delays.

Euclid Cycle Time Estimate Euclid's algorithm applies the same sequence of operations to find the GCD and the two Bézout coefficients. These operations can be done in parallel, so we can estimate the cycle time for the algorithm by considering how we can optimally implement the operations required to compute remainder sequence $\max(a, b) \mod \min(a, b)$ each cycle to find the GCD. This sequence is determined by generating the quotient $q = \max(a, b)/\min(a, b)$ and then computing the remainder with $\max(a, b) - q * \min(a, b)$ every cycle. Since this algorithm requires a subtraction every iteration, it is advantageous to keep our variables in CSA form in this case as well to avoid the carry-propagation for large-integer subtraction each cycle (Section 4.2). Note that in CSA form, we cannot determine whether a or b is greater since their actual values are not directly stored, but we can perform this comparison once at the start and continue swapping the values each iteration. For simplicity, we denote a as the larger number and b

as the smaller number in this discussion, rewriting the core computation as $a - q * b$.

Prior work has introduced the idea of dividing just the most significant bits (MSBs) and computing the full large-integer division if the approximate division does not return correct results [Leh38, Jeb93b, Jeb95, Sor95]. Note that the number of iterations stays the same, but the number of iterations requiring a large-integer division is lower. We build upon this work to estimate the delay for an optimal hardware implementation for such algorithms. When considering only the c MSBs of the divisor and the dividend, an optimal way to implement division in hardware is to use a look up table (LUT) that takes as input pairs of c bits and outputs an approximate quotient. We require this quotient to be less than or equal to the true quotient, since subtracting a smaller factor will still result in a valid quotient and remainder sequence for Euclid's algorithm but just require more iterations. This is accomplished by storing the quotient for all these c-bit pair combinations, assuming that the remaining $1024 - c$ bits of the 1024-bit dividend are all zero while the remaining bits of the 1024-bit divisor are all one. In this way, every approximate quotient guarantees a valid quotient that can be used to generate a valid remainder every iteration.

We note that the size of the LUT inputs is $2 * c$, so the LUT will have $2^{(2*c)}$ entries. Thus, the LUT size grows exponentially, requiring over a million entries for $c > 10$, which is impractical. Thus, we consider the delays for when $c = 5$ and $c = 8$ to have reasonably-sized LUTs. Since the a, b are in CSA form, we require a c-bit carry-propagate addition to get the LUT inputs, equal to $\lfloor \log_2(c) \rfloor$ CSA delays with an efficient adder implementation. Note that in this approach, we can reduce at most c bits every cycle (aside from any additional lucky cancellation), so more than an average of 598 iterations are likely required since that count assumed that any number of bits could be reduced every iterations. If we keep the conservative estimate of 598 iterations for the algorithm (which will not reflect the advantage of using $c = 8$ compared to $c = 5$), the total delay for getting the inputs for the look up alone requires $\lfloor \log_2(5) \rfloor * 598 = 1196$ CSA delays when $c = 5$ or $\lfloor \log_2(8) \rfloor * 598 = 1794$ when $c = 8$. The CSA approach with the resulting higher cycle count is still preferable since the alternative approach requires 598 1024-bit carry-propagate subtractions, i.e., 5980 CSA delays. Note that the time for lookup would require additional delay since the LUT sizes are 1024 and 65536 for $c = 5$ and $c = 8$, respectively, but this delay is not included in these conservative estimates.

After this look up, we need to multiply the c-bit quotient, q_c, by the 1024-bit divisor to get $q_c * b$ for computing the remainder. Since b is in CSA form, we require two multiplications to compute this result. Thus, the delay for this operation is equal to the delay of generating $2 * c$ partial products and using CSAs to sum them together, with the final result stored in CSA form. We do not include the delay for generating the partial products in our conservative estimates. To sum the partial products: for $c = 5$, we need to sum ten numbers. With three-input CSAs, we first require $\lceil \frac{10}{3} \rceil = 4$ CSAs which output $4 * 2 = 8$ results. Similarly continuing to reduce the number of results until two results that store one number in redundant form are left requires $\lceil \frac{10}{3} \rceil + \lceil \frac{8}{3} \rceil + \lceil \frac{6}{3} \rceil + \lceil \frac{4}{3} \rceil = 4 + 3 + 2 + 1 = 10$ CSA delays. For $c = 8$, we similar find that adding the partial products requires 16 CSA delays. Multiplying these values by the conservative 598 estimate for the number of iterations gives 5980 and 9568 delays for this multiplication in the $c = 5$ and $c = 8$ cases, respectively, making this the most expensive operation required each iteration.

To compute the remainder, we need to subtract $a - q_c * b$. Since $q_c * b, a$ are in CSA form, we require two three-input CSAs. Then, each iteration requires two additional CSA delays, resulting in a total of 1196 more CSA delays. Finally, after all the iterations are over, we need to compute one 1024-bit carry-propagate add to convert the final results out of CSA form, requiring $log_2(1024) = 10$ CSA delays.

Thus, our delay estimate for an optimal implementation of Euclid-based algorithms is $1196 + 5980 + 1196 + 10 = 8382$ CSA delays with $c = 5$, which is 2.3X higher than our two-bit PM delay estimate. Furthermore, we note that our delay estimate for Euclid-based algorithms is conservative. In addition to the reasons noted as we derived the estimates, we note that an efficient implementation would require a c-bit leading ones detector to determine where the most significant bits start in the 1024-bit variables every iteration, which will contribute some additional delay every cycle. Furthermore, if more than c bits are able to be cancelled based on the lower bits of the numbers, then this detector would not accurately find the MSB and require more computation and/or cycles to adjust for that. Note that due to our conservative cycle count estimate, the delay for $c = 8$ is higher and we compare the delay of the $c = 5$ case since it is the lower estimate.

Finally, we note that alternatively using fast large-integer division algorithms [Fly70, Gol64] would require about 15 iterations per division in Euclid's algorithm, where each iteration requires an 1024-bit by 1024-bit multiplication. This results in $15 * 598 = 8970$ expensive multiplications, which is already considerably more expensive than the two-bit PM estimate and the more optimal implementation approach discussed above.

3.5 Summary

We have made the following key observations. First, software algorithms tend to optimize performance for cycle count (as most literature has done), but when designing hardware, it is more efficient to switch to an algorithm that optimizes *both* cycle time and cycle count. We note that for both our motivating applications, the existing ASIC and FPGA papers choose to adapt existing high-performing software algorithms that focus on variants of Euclid's algorithm [ZST+20, ZTW21, DdPM+21]. Second, we conclude that for a hardware design context, using redundant representations with the two-bit PM algorithm is more promising compared to accelerating variants of Euclid's algorithm. Third, we find that this decision can be expected to improve both average-case and worst-case performance.

4 Fast XGCD Algorithm and Hardware Design Space

Our performance analysis in Section 3 motivates a deeper evaluation into Stein-based algorithms in the context of hardware acceleration. This section presents and evaluates an algorithm and hardware design space within this context. Before we begin, we first note that the fastest existing algorithm in our analysis was the two-bit PM algorithm [YZ86] (see Listing 1). However, this algorithm was originally not completely specified and did not address corner cases that require updating odd intermediate Bézout coefficients. We therefore begin by contributing a *complete extended two-bit PM algorithm.* With this modified algorithm, we then present a series of both algorithmic modifications (handling carry propagation for the termination condition and increasing the number of bits reduced per cycle) as well as hardware design choices (using carry-save adders and minimizing control overhead) that together enable high-performance XGCD hardware acceleration with an extremely short cycle time. Finally, we show how our XGCD design easily supports constant-time evaluation and polynomial XGCD.

Notation-wise in this section, we note that GCD algorithms usually consider the case when the inputs are odd for simplicity, since common factors of two can be easily removed by shifting before the main computation loop. If one input is even after this reduction, we add the two inputs so that both inputs are odd, as in [YZ86]. Let a_0 and b_0 represent the true initial inputs and let a_m and b_m represent the odd inputs after shifting.

4.1 Complete Extended GCD with Two-Bit PM

Typically, existing GCD algorithms can be extended to calculate the Bézout coefficients by introducing four intermediate variables u, m, y, n such that the relations in Equation 1

```
1. [initialization] $\delta := 0$ and $a := A$ and $b := B$
2. [iterations] case 1 (EVEN($a/2$) and EVEN($a$)): $a := a/4$ and $\delta := \delta - 2$
                case 2 (not EVEN($a/2$) and EVEN($a$)): $a := a/2$ and $\delta := \delta - 1$
                case 3 (EVEN($b/2$) and EVEN($b$)): $b := b/4$ and $\delta := \delta + 2$
                case 4 (not EVEN($b/2$) and EVEN($b$)): $b := b/2$ and $\delta := \delta + 1$
                case 5 ($\delta \geq 0$ and EVEN($\frac{b+a}{2}$)): $a := \frac{b+a}{4}$ and $\delta := \delta - 1$
                case 6 ($\delta \geq 0$ and EVEN($\frac{b-a}{2}$)): $a := \frac{b-a}{4}$ and $\delta := \delta - 1$
                case 7 ($\delta < 0$ and EVEN($\frac{b+a}{2}$)): $b := \frac{b+a}{4}$ and $\delta := \delta + 1$
                case 8 ($\delta < 0$ and EVEN($\frac{b-a}{2}$)): $b := \frac{b-a}{4}$ and $\delta := \delta + 1$
3. [termination condition] return GCD= $a + b$ if $a = 0$ or $b = 0$, or repeat step 2
```

Listing 1: Two-bit PM algorithm [YZ86]. $\delta \approx \log_2(a) - \log_2(b) \approx a - b$ to check if $a > b$.

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	$(u + 3 * b_m)/4$	$(m - 3 * a_m)/4$

hold true for inputs a, b in every iteration. At the start of the main computation loop, $a = a_m$ and $b = b_m$, so the initial values must be $u = 1, m = 0$ and $y = 0, n = 1$.

$$\begin{aligned} u * a_m + m * b_m &= a \\ y * a_m + n * b_m &= b \end{aligned} \tag{1}$$

The two-bit PM algorithm updates either a or b each iteration. When a is not updated, there is no need to update u, m since Equation 1 automatically holds. The same goes for b and y, n. When a or b is divided by 2^n, we need to divide u, m or y, n by the same factor to maintain the relations in Equation 1. However, the divisibility of u, m and y, n are not guaranteed to match the divisibility of a and b, respectively, and if odd values are shifted, the truncated results will not preserve these relations. We need to address this problem.

We consider the shift-by-one case first. Assume, 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 are still able to 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 can 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 beforehand.[2] We similarly rewrite the other u, m updates as shown in Table 3. Thus, the worst-case update delay is half the original form and similar to the shift-by-one case.

While cases one through four of the two-bit PM algorithm (Listing 1) apply updates directly with a or b, the last four cases apply updates on $a \pm b$ and then reduce two bits.

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

To preserve the relations in Equation 1 in cases five through eight, we apply a similar strategy to that used in the shift-by-two case on $u \pm y$, $m \pm n$ instead of individually on u, y, m, n. Thus, in these cases, we can 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 the update for m in case six and n in case eight ($\frac{m-n-3*a_m}{4}$), requiring an extra subtraction compared to the worst-case operations in the first four cases.

Finally, at the end of the computation, 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 $b_b = m + n$.

4.2 Carry-Save Adders

Carry-save adder (CSA) designs improve the delay of back-to-back additions by removing carry propagation in all intermediate computations. This results in $O(1)$ delays rather than $O(\log(n))$, where n is the input bitwidth, making such savings important in wide-word arithmetic with large bitwidths. CSAs output the sum in CSA form, or redundant binary form, where two bits represent one bit of the result. In this form, a number x is represented by *carry* and *sum*, where $x = carry + sum$. Since CSAs have a constant delay, the cycle time for our designs will not be very sensitive to the input bitwidth.

Since our XGCD iterations require repeated additions to update a, b and u, m, y, n, we can benefit from relatively fast carry-save additions during the iterations until a time-consuming carry-propagate add at the end to convert out of CSA form. This approach reduces the number of carry-propagate additions from $O(n)$, where n is the number of iterations, to $O(1)$. During the iterations, we keep most variables in CSA form. However, we directly store and update δ, an approximation of $a - b$, out of CSA form since we need to check the sign of δ every cycle to determine which branch to take, and the sign of a number is not directly known in CSA form. Since this variable is small ($\log_2(1024) = 10$ bits with 1024-bit inputs), carry-propagate adds to update this value do not limit the cycle time. As a result, for constant-time designs, we also keep a cycle counter, which is $\log_2(1.51 * n + 1)$ bits (Section 4.6), out of CSA form, since it is similar in size to δ. Additionally, there is no benefit to keep constants a_m, b_m and their multiples $k * b_m, k * a_m$ for $k = 2$ to $k = 7$ (Section 4.1) in CSA form since they are not continually updated.

While prior work has suggested using CSAs for GCD algorithms [Pur83, YZ86], we found that using CSAs in practice surfaces challenges that have not been previously addressed. We describe our solutions to these challenges below.

Remaining carry-propagate adds Carry-propagate adds are still required to (1) add inputs a_0 and b_0 if one is even at the start to generate two odd inputs a_m, b_m for the initial values of a, b in the main computation loop, (2) precompute $k * b_m$ for $k = 2$ to $k = 7$ when k is odd (e.g., to compute $3 * b_m$ as $2 * b_m + b_m$), and (3) convert from CSA form to normal representation at the end for the final results. We need to ensure these additions do not limit our cycle time for the hardware to benefit from the delay savings of using CSAs each iteration. Since these necessary carry-propagate adds occur either before the iterations as in (1), (2), or afterwards as in (3), we run the initial and final computations at one-half and one-quarter of the clock frequency of the system clock, respectively. 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 in CSA form, there is a need to efficiently add one to get the correct result when a carry-in would have been generated by the shifted (now lost) bits. The inherent truncation associated with the shift loses this information, resulting in an incorrect answer compared to if we shifted (and truncated) the represented number out of CSA form. We note that we cannot simply set the lower bit of *carry* or *sum* to be one to correct the result, since the LSB for both can already be one after a shift. Thus, we instead use a half adder to add the shifted *carry* and *sum*. This operation results in another (*carry*, *sum*) pair for the same number represented in this split CSA form. Since the LSB of the *carry* output from a half adder will be zero by design, we set that bit to one whenever we need to add one back. In this way, we only add the delay of a single XOR gate for this correction.

Detecting when to apply this addition-by-one correction is cheap: an AND gate delay in the divide-by-two case and an OR gate delay in the divide-by-four and divide-by-8 cases. A carry-in to the next bit will be generated whenever the relevant LSBs of *carry* and *sum* are not all zeros. In the divide-by-two case, we use an AND gate to check when the LSBs are both odd. In the divide-by-four and divide-by-eight cases, we use an OR gate on the second and third LSBs of *carry* and *sum*, respectively. This gate correctly detects all the cases a carry-in is generated and the addition-by-one correction is needed. If the OR gate output is one but the number is not divisible by four or eight, then the calculated result is not used. As a result, there is no need for extra logic to detect the other bits.

Arithmetic right shifts Prior work approximates which of a, b is larger when both are odd [BK85]. If the approximation is incorrect, the smaller number is updated, making a, b (and u, y, m, n) negative. While the sign of a number cannot be directly determined in CSA form, it can be preserved. Earlier work determines the relation between the two most significant bits of *carry* and *sum* before and after shifting to the right [TPT06]. We use the truth table from that paper to determine balanced equations for shifting numbers in this form, with the worst-case delay equal to the delay for an XOR gate and an OR gate. While this logic can be applied sequentially for dividing by higher powers of two, this approach increases delay: physical design tools are unable to automatically simplify this chained logic since this shifting logic is separated by a half adder for the addition-by-one correction. As a result, we further specialize the logic when dividing by four and eight to make the worst-case delay is in these cases equal to the delay for two OR gates.

4.3 Termination Condition Carry Propagation

Stein-based algorithms, including the two-bit PM, check whether a or b is equal to zero to decide to exit the main computation loop. This check is expensive because checking if a or b is equal to zero requires knowing what the actual values of a and b are. However, since it is beneficial to store a and b in a redundant form when working with large integers (Section 4.2), obtaining their actual values would require two parallel carry-propagate adds: $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$. This computation would negate the benefit of storing these numbers in a redundant form since we need to convert out of this form every cycle with an expensive carry propagation. In addition, this operation requires a large AND-gate tree to check whether all $2 * n$ bits are zero every cycle, where n is the bitwidth of the inputs. To improve the delay of this check, we investigated the tradeoffs of two possible approaches and their suitability for software versus hardware implementations.

Sampling Our first approach spreads out the carry propagation for $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$ over x cycles and then samples the true values of a, b every x cycles. For 1024-bit inputs, we find that $x = 4$ ensures these additions do not limit our cycle time. We then run the XGCD iterations at our system clock frequency and run these expensive additions over four clock cycles in parallel. With this approach, we can only check if the termination condition has been satisfied every four cycles and in the worst case, require four extra iterations to compute the XGCD. This is a very small overhead, with a range of 0.18 to 0.43% of the number of cycles for the (2, 2) to (8, 8) reduction factor designs in Section 4.5. Since we can run this computation in parallel to our variable updates and it minimally adds to the number of cycles, we use this approach in our hardware designs.

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 updated every cycle by the minimum number of bits a and b will be reduced by. Prior work further does not store α, β but instead stores $\delta = \alpha - \beta$ directly to approximate which number is greater [BK85, YZ86]. We instead continue storing α, β to use these values for 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 final result computation. In this way, we completely avoid the carry-propagation for $a = a_{carry} + a_{sum}$ and $b = b_{carry} + b_{sum}$ since the termination condition no longer checks a, b. Also, since α, β approximate the binary logarithms of a, b, we only need to check if $\log_2(1024) * 2 = 20$ bits are zero instead of 2048 bits for 1024-bit inputs, reducing the AND-gate tree delay.

However, since α and β are approximations, we found that they significantly diverged from the true values of $\log_2(a)$ and $\log_2(b)$. In particular, when a or b is updated with $a - b$, multiple bits could be reduced, but only one bit is subtracted from α, β. As a result, using α, β instead of a, b for the termination condition adds 150+ cycles for 1024-bit inputs, which is a 7 to 16% overhead depending on the reduction factors used (Section 4.5).

We experimented with occasionally correcting the α, β approximations to be the true values of $\log_2(a), \log_2(b)$ for the a, b at a given point, which would require 1024-bit carry-propagate adds. Since these additions can be computed in parallel, it is beneficial to run this correction as often as possible – every four cycles as in the sampling approach. In this way, α, β closely track the actual logarithms and at most four extra cycles are required. However, this correction requires not only 1024-bit carry-propagate adds to convert a, b out of CSA form, but also requires computing the absolute values of a, b since they can be negative and then computing $\log_2(|a|), \log_2(|b|)$. Thus, for a hardware design, the sampling approach without α, β requires less computation, area, and energy.

4.4 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 (a, b), (u, y), and (m, n) 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. We instead 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 similarly apply this parallel computation and late select strategy wherever possible within all the update modules as well. We note that some computation uses the same inputs regardless of the updated variable (e.g. $u \pm y$ in both the update u and y modules), so we avoid redundantly computing such updates to save area and energy.

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 16nmn designs.

To reduce this delay, we compute the control signals to determine which branch to take for the next cycle in parallel with computing the updates for a, b, u, y, m, n. We do this instead of generating the control signals from the original values to determine which branch to take 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 update options for u in parallel. Then, we select the right control signal update based on the u update chosen. To similarly compute the divisibility of $a \pm b$, $u \pm y$, and $n \pm m$ during the previous cycle as inputs for the next cycle, we add a few XORs ($\approx$ 20ps each in 16nm), which is still smaller than the 0.15ns large buffer delay.

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 runtime (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

This optimization eliminated enough buffer delay on the control path such that the critical path is the data path for all the reduction factor pair designs we considered, except for the (8, 8) design. In this design, the control path is dominant and has larger delay than the data path due to the disproportional increase in the number of branches compared to the computation required for higher odd reduction factors.

4.5 Optimal Reduction of Bits Per Cycle

Stein-based algorithms reduce the number of iterations required by removing factors of two or four in an iteration. The higher factors of two that can be removed per iteration, the fewer iterations required on average. When computing only the GCD, removing more factors of two is cheap since that just requires shifting more bits to the right. However, for the XGCD, reducing more bits requires more branches to choose between depending on the divisibility of u, y, m, n. This added control logic 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 number of clock cycles required to determine the optimal reduction of bits per cycle. We separately vary the maximum reduction factor for the case of when a or b is even and the case of when a and b are odd (referred to as even and odd reduction factors) to see which parametrizations yield a net benefit for total runtime (cycles $*$ cycle time) in the average and worst-case scenarios and for our two motivating applications.

We implement an odd reduction factor of two as updating a or b with $\frac{a-b}{2}$ (their sum is never computed since we only either update a or b each cycle). We support a an odd reduction factor of eight such that the worst-case path requires two subtractions and a shift three bits to the right, with the procedure in Section 4.1. Our efficient rewriting of the updates (requiring us to compute $k * b_m$ and $k * a_m$ for $k = 1$ to $k = 7$ for the three-bit reductions) maintains the same number of additions/subtractions on our data path compared to the shift-by-two updates. Note that the following (even, odd) maximum reduction factor pairs describe factors of two reduced in prior GCD algorithms: (2, 1) – Stein's algorithm, (2, 2) – Purdy's and PM algorithms, and (4, 4) – two-bit PM algorithm.

Table 4 shows the runtime and area for designs with various reduction factor pairs. Note that these designs include the other optimizations described in this section. The worst-case data path is updating m, n when a and b are odd. Our efficient rewriting of this computation as explained in Section 4.1 ensures that all the branches when a and b are odd require the same amount of worst-case scenario computation. We observe that the critical path (cycle time) is the data path for all the designs but the (8, 8) design. In that case, the cycle time is dictated by the control path delay determining which branch to take instead. Finally, we note that since an odd reduction factor of eight does not result in any benefit in average cycle time compared to a reduction factor of four, designs with an odd reduction factor of 16 or higher would not be beneficial since the increase in the control logic delay and thus cycle time, outweighs the benefit of the lower number of cycles.

We observe that the XGCD runtimes for these designs are close to each other, especially within a given odd reduction factor group. These results show that while we can reduce the average number of cycles required by reducing more bits per cycle, the resulting increase in the cycle time mostly cancels that benefit. Choosing which design to use depends on the application requirements, primarily the number of XGCDs required, the importance of a small design, and the necessity of a constant-time design.

These small XGCD runtime differences become important when this computation is repeated many times. For squaring binary quadratic forms, the number of squarings required can be over a million or billion as most applications require a large delay. Each squaring requires two XGCDs in the NUDUPL algorithm, so a 5ns difference in XGCD runtime can result in a 10ms to 10s difference for this application. Since this application requires high performance, using the fastest design corresponding to the (8, 4) design would be most appropriate and preferable over the 5ns-slower (4, 4) design.

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], which evaluates to 1547 cycles for 1024-bit inputs. In the worst-case in every cycle, one bit is reduced if the values are even or one bit is reduced if the values are odd (since either $a+b$ or $a-b$ may generate a carry but will be divisible by four when both a, b are odd). However, if δ incorrectly approximates whether a is greater than b, than no bits will be reduced from the larger number and the smaller one will be updated.

The authors of the two-bit PM algorithm note that reducing three bits – corresponding to the (8, 8) design – will not reduce the maximum 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 and b may not be divisible by eight when they are even), so these additional branches may never be taken. We note that this logic extends for the (2, 8), (4, 8), (2, 4), and (8, 4) designs as well since the only guaranteed transitions remain either reducing one bit when a, b are even or dividing $a+b$ or $a-b$ by four when a, b are odd. Thus, shifting by three bits is beneficial for applications that require optimizing the average runtime but does not add any further value for applications that require optimizing the worst-case time. In addition, the two-bit PM algorithm with (4, 4) does not reduce the worst-case time compared to the smaller and more energy-efficient (2, 4) design. Finally, we note that the average runtimes for the designs with an odd reduction factor of two is higher than the worst-case runtime for all the other designs, so these designs are not competitive in this context. Thus, the optimal reduction factors for constant-time applications are (2, 4).

4.6 Extensions

Constant-time Constant-time implementations require the execution to take the same amount of time regardless of the input values. In other words, the execution runtime should always be equal to maximum runtime. Our XGCD algorithm can be padded to achieve a constant-time algorithm. For any of the reduction factor pairs from Section 4.5, running for any number of cycles beyond the nominal termination condition will still yield the correct answer. Since the inputs to the main computation loop are preprocessed to remove common factors of two, the GCD and valid Bézout coefficients will be found when one of a or b is equal to zero and the other is equal to an odd number. Then, the algorithm will continually detect one of these numbers to be even and keep dividing zero by a factor of two. Note that when these values are stored in a redundant representation as explained in Section 4.2, it is not clear when numbers become zero since they are not directly stored. Thus, the termination condition can instead keep track of the number of cycles and when that count is equal to the worst-case cycle count, the computation can end. As determined in Section 4.5, our optimal design for constant-time XGCD is with reduction factors (2, 4), which requires $1.51 * n + 1$ number of iterations for n-bit inputs.

Polynomial XGCD We can use the same algorithmic control flow to find the GCD between two polynomials with integer coefficients by describing the polynomial equivalents for all the integer XGCD operations required, building from prior work [BK84, BY19]. Stein's-based integer GCD algorithms reduce factors of two, the smallest prime. The polynomial equivalent is reducing factors of x: $x, x^2, x^3, \ldots$. Accordingly, evenness translates to polynomial divisibility by x. To ensure adding/subtracting "odd" polynomials guarantees an even results (as is the case with integers), we can multiply the polynomials by the other polynomial's constant term to enable cancelling the constant terms in addition/subtraction to reduce the polynomial degree by at least one [BY19]. For comparisons, we compare the degrees between polynomials. This maps all the integer GCD operations to valid polynomial equivalents. To also find the polynomial XGCD, we maintain the same 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 initially set $u = 1, m = 0$ and $y = 0, n = 1$, as with integer GCD. The same updates can be applied by reducing factors of x instead of 2 and adding/subtracting polynomial coefficients.

5 Complete Hardware Accelerators for Target Applications

We build a complete hardware accelerator for each of our two cryptographic applications. For constant-time modular inversion, the XGCD hardware accelerator in fact directly implements modular inversion and requires no additional hardware since the first Bézout coefficient is the modular inverse result. This section will therefore describe our complete accelerator for the VDF squaring application, which includes the remaining operations on the critical path for the NUDUPL algorithm (see Section 2.1). Our implementations for these non-XGCD operations are summarized in Table 5.

We first note that the additions on the critical path are in isolation, so 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 Barret Reduction (which precomputes constants) are expensive and not effective. Given that this operation minimally contributes to runtime, we naively implement $x = y * z \pmod{a}$ as a multiplication ($m_1 = y * z$), division $d = m_1/a$, another multiplication $m_2 = d * a$, and a subtraction for $m_1 - m_2$, which is equal to x.

For fast large-integer multiplication, Karatsuba multiplication [Kar63] was one of the first algorithms introduced, requiring at most $n^{\log_2(3)} \approx n^{1.58}$ multiplications instead of the traditional schoolbook algorithm's n^2 limit, given n-bit inputs. Karatsuba breaks up one multiplication into three smaller ones and a few additions and shifts. This process can be recursively applied to a base case input bit length. Implementing this algorithm in hardware has proven popular [EhyMZ+08, vzGS05, ROH17, NdMM03a, NdMM03b, ZST+20].

The Toom-Cook algorithm [Too63, CA69] is a more efficient and general version of the Karatsuba algorithm that splits numbers into some k parts (k = 3 for Toom-3, which we implement). The commonly used Toom-Cook polynomial algorithm [BZ07] finds the product of evaluating polynomials formed with the split numbers as coefficients at various points to determine the product polynomial, and then uses the product's coefficients to get the answer to the original multiplication problem. This algorithm is well-suited for our input size since it is slower than schoolbook multiplication for smaller numbers and the fast fourier transform-based Schönhage-Strassen algorithm [Sch77] for numbers larger than

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

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

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

$2^{2^{15}}$. Few hardware designs use Toom-Cook [GL18, DLG18], however, since it requires a division by three, which is typically not hardware-friendly. We reduce the complexity of the division by rewriting $\frac{x}{3}$ as $x - \frac{2x}{3}$ as shown in Listing 2. This allows us to calculate bit by bit of the output, $\frac{x}{3}$, which generates the next bit for $\frac{2x}{3}$ to use in the subtraction.

Fast division algorithms include the Newton-Raphson [Fly70] and Goldschmidt [Gol64] iterative algorithms, which convert division operations into multiplication and then use fast multiplication methods. Both of these algorithms start with an initial estimate of the quotient and require around the same number of iterations for similarly-sized inputs [ESF05]. Few papers focus on accelerating division algorithms in hardware [HSGJ10, ZST+20]. Since there are only two divisions in our critical path and they constitute a small portion of total squaring runtime, we implement the Newton-Raphson algorithm with Toom-3 for multiplication. For 1024-bit inputs, this algorithm requires 15 multiplications and 15 additions. The additions and multiplications in Table 5 include the operations necessary for the divisions with Newton-Raphson, so there is no separate row for division.

6 ASIC Evaluation

Setup We use a vertically integrated methodology spanning cycle-level performance modeling, VLSI-level modeling, and detailed physical design before taping out our design. We designed 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 different input bitwidths, reduction factors, and levels of constant time support. We randomly generate inputs to verify our designs with our functional model in Python. We use mflowgen 0.3.1 [CTN+21] for our physical design workflow which relies on Synopsys DC 2019.03 for synthesis, Cadence Innovus 19.10.000 for floorplan, power, place, clock tree synthesis, and route. All our numbers in this paper are signoff numbers, reported after placing and routing our designs and executing the full physical design flow.

Unless otherwise specified, all C++ code is run on a 2020 MacBook Pro with the M1 chip, compiled with g++ and -O3 optimization, with random 1024-bit inputs. We use the standard C++ chrono library with nanosecond precision for timing results.

ASIC Designs We evaluate two XGCD ASIC designs generated with different parameters, one for each motivating application: (1) 1024-bit XGCD with reduction factor pair (8, 4) for squaring binary quadratic forms over class groups and (2) 255-bit constant-time XGCD with reduction factor pair (2, 4) for modular inversion for Curve25519. Both

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: XGCD chip area breakdown for 1024-bit Design (1) and 255-bit Design (2). Chip density is 70% for Design (1) and 79% for Design (2).

Module	**Design (1) Area (μm^2)**	**Design (2) Area (μm^2)**
Pre-iterations computation	25053	2844
Update a, b	25006	5255
Update u	55755	9958
Update y	30878	2019
Update m	56813	9932
Update n	32744	2032
Termination condition	4720	8
Post-iterations computation	17301	4274
JTAG	10577	2707
Miscellaneous	1553	475
Total	260400	39504

designs are in TSMC 16 nm, using SVT, LVT, and ULVT libraries at 0.8V. 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 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 our earlier analysis, the critical path requires three CSAs to compute the sum of the Bézout coefficient variables when a, b are odd and additionally add a multiple of b_m when these variables are odd. Then, we require some logic to preserve sign when shifting in CSA form, with a half-adder to correct incorrectly truncated results. Finally, we select the update to apply for our coefficient variables using late selects and precompute control signals for the next cycle. We note that 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, both our 1024-bit and 255-bit designs have similar cycle times, showing that our use of CSAs makes our cycle time relatively independent of input size.

Table 7 shows the area breakdown of our designs. Since the $u \pm y$ and $m \pm n$ updates for (u, y) and (m, n) are interchangeable and their divisibility by factors of two are always the same, we compute these values in only the update u and m modules to avoid redundant computation with no performance penalty. Thus, our update y, n modules are 2X and 5X smaller than the u, m modules for the 1024-bit and 255-bit designs, respectively. These

Table 8: Comparison of our fast 1024-bit and constant-time 255-bit XGCD designs to implementations in prior work. NR = not reported. * NR but assumed from other work.

XGCD Design	Technology / Processor	Area (mm^2)	Clock Frequency	Cycles	Time (ns)
1024-bit inputs					
[AHAJS16]	Xilinx XC7VH290T-2-HCG1155 FPGA | 28nm	NR	39.94 MHz	598 *	151777
GNU XGCD C++	Apple M1 | 5nm	-	-	-	10650
[ZTW21]	TSMC 28 nm	2.4	250 MHz	1623	6490
[ZST+20]	TSMC 28 nm	9.9	500 MHz	3000	6000
Our Design (1)	**TSMC 16 nm**	**0.41**	**3.89 GHz**	**1143**	**294**
255-bit inputs Constant-time					
[DdPM+21]	Zynq UltraScale+ XCZU7EG FPGA | 16nm	NR	207 MHz	8466	40900
[BY19]	Intel Kaby Lake | 14nm	-	2.3 GHz *	8543	3714
[Por20]	Intel Coffee Lake | 14nm	-	2.3 GHz	6253	2719
Our Design (2)	**TSMC 16 nm**	**0.059**	**4.55 GHz**	**396**	**87**

modules comprise the majority of the area since we duplicate computation for late selects and these updates have the most options to compute and select from.

XGCD Comparison We compare our designs to prior work in Table 8. Design (1) achieves a 211X speedup over prior FPGA designs [AHAJS16], a 36X speedup over the GNU XGCD implementation in C++ profiled on Apple's M1 processor in 5nm, and a 8X speedup over the state-of-the-art ASIC design [ZST+20] for 1024-bit inputs. We profile the time for the C++ implementation of XGCD in the GNU Multiple Precision Arithmetic Library (GMP), which has been optimized for large integers. For the ASIC and FPGA comparisons, we scale prior work runtimes in 28nm to 16nm for fairer comparisons.

The FPGA design implements Euclid's algorithm and mentions using CSAs [AHAJS16], though no further detail is provided. Our 40X shorter clock period, along with our decision to build from the two-bit PM algorithm instead, likely result in our significant speedup.

We achieve the speedup over the SOTA ASIC design due to two reasons: First, Zhu et al. implemented the GMP XGCD algorithm, which requires an average of 3000 cycles [ZST+20], while our Design (1) reduces the number of cycles required by over 60%. Second, the GMP algorithm uses the most significant bits of the numbers for some division steps in Euclid's algorithm, which we estimated will likely not translate into faster total runtimes compared to the two-bit PM algorithm (Section 3). Since our work uses simpler operations — adds, shifts, and comparisons — building upon the two-bit PM [YZ86] and Stein's algorithms [Ste67], we reduce both the cycle count and cycle time compared to Zhu et al.'s work. Finally, we note that Zhu et al. provide synthesis numbers, which uses approximations for wire delays and thus may not translate into measured ASIC runtimes. We instead build an ASIC for our design by executing the full physical design flow, which includes the final layout and does signoff-quality RC extraction for wire delay.

Design (2) achieves a 31X speedup and a 470X speedup over the state-of-the-art software [Por20] and FPGA design [DdPM+21], respectively. Pornin builds from Stein's algorithm as we do but focuses on (1) equalizing the time for each iteration of the algorithm, which naturally happens in a hardware design with a standard clock frequency and (2) using approximations of large values for faster computation [Por20], which we solve by approximating $\log_2(a) - \log_2(b)$ as in prior work [BK85, YZ86], storing our numbers in CSA form (Section 4.2), and sampling the values for our termination condition (Section 4.3).

On the hardware side, [DdPM+21] is the first (and only) constant-time XGCD paper we are aware of. These authors implement the integer version of the Bernstein-Yang XGCD algorithm [BY19] on an FPGA [DdPM+21]. Compared to their fastest design, we are able to achieve a much faster cycle time compared to this prior work for two reasons. First, their faster design pipelines all additions to achieve a cycle time of 4.8ns for 255-bit inputs, while we use carry-save adders to eliminate rather than pipeline the carry propagation in addition. Second, the Bernstein-Yang algorithm requires modifying fractions and implementing division efficiently, while our approach based on Stein's algorithm inherently requires simpler operations, translating into faster runtimes, as estimated in Section 3.

Finally, we note that we can use Design (1) for *both* applications by setting the constant time configuration to be true and the bitwidth to 255 in the 1024-bit design. The worst-case number of cycles for the algorithm is the same for both designs and even with a longer cycle time, the performance of Design (1) is still 9X faster than Pornin's work [Por20] and 139X faster than the prior FPGA design [DdPM+21]. Since ASICs can be costly, using the same hardware unit to obtain state-of-the-art performance for various applications with different bitwidths (1024 vs 255 bits) and requirements (constant-time vs not) is beneficial.

Squaring speedup We ran over one million trials of the Chia Network's reference C++ implementation 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 runtime to $2 * 294ns = 588ns$ and speed up the full algorithm by 8.5X, which is close to 9.1X, the best we could do by accelerating only the XGCD. To further speed up the full algorithm, we accelerate the remaining operations on the critical path (Section 5) to execute this computation in $1.5us$, achieving a 14X speedup over the C++.

To our knowledge, this paper is the first to consider hardware acceleration for the NUDUPL algorithm. One prior paper accelerates a less efficient squaring algorithm for squaring over class groups, with a runtime of $6.3us$ per squaring for 1024-bit inputs in 28nm [ZST+20]. However, we note that comparing our work to this prior paper is not a fair comparison because their work considers solely the squaring operation without any reduction afterwards, while the NUDUPL algorithm squares and additionally partially reduces the outputs. Despite handling more of the overall computation required, our work is 1.7X faster than this prior design due to the higher efficiency of our XGCD design (as compared to this prior work earlier) and our use of the more efficient Toom-3 algorithm instead of the Karatsuba algorithm for large-integer multiplication and division.

7 Conclusion

Fast XGCD implementations are increasingly important as the cryptography community investigates applications that are dominated by the XGCD operation: verifiable delay functions based on class groups and modular inversion for elliptic curve cryptography based on XGCD. However, existing literature is sparse on XGCD hardware, providing only point solutions for specific applications, and focusing narrowly on one specific family of algorithms (i.e., Euclid's algorithm). This work contributes an algorithm and hardware design space exploration that reveals that the best algorithm in an optimized software context is not the same as that in a hardware acceleration context. Unlike other hardware XGCD works, we build on variants of Stein's algorithm and greatly improve performance using redundant representations and various algorithm and hardware techniques. Our 16nm ASIC achieves 8X speedup over the state-of-the-art ASIC and 36X speedup over optimized C++ running on an Apple 5nm M1 processor. These results enable squaring binary quadratic forms 14X faster than C++ on the M1 and computing modular inverses for Curve25519 31 to 470X faster compared to FPGA and software implementations, significantly advancing XGCD performance for future 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.

[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.

[ESF05] Guy Even, Peter-M Seidel, and Warren E Ferguson. A parametric error analysis of goldschmidt's division algorithm. *Journal of Computer and System Sciences*, 70(1):118–139, 2005.

[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/.

[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.

[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.

[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.

[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.

[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.

[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.

[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.