Low-latency Hardware Architecture for VDF Evaluation in Class Groups

Danyang Zhu, Jing Tian, Minghao Li, and Zhongfeng Wang, *Fellow, IEEE*

Abstract—The verifiable delay function (VDF), as a kind of cryptographic primitives, has recently been adopted quite often in decentralized systems. Highly correlated to the security of VDFs, the fastest implementation for VDF evaluation is generally desired to be publicly known. In this paper, for the first time, we propose a low-latency hardware implementation for the complete VDF evaluation in the class group by joint exploiting optimizations. On one side, we reduce the required computational cycles by decreasing the hardware-unfriendly divisions and increase the parallelism of computations by reducing the data dependency. On the other side, well-optimized low-latency architectures for large-number divisions, multiplications, and additions are developed, respectively, while those operations are generally very hard to be accelerated. Based on these basic operators, we devise the architecture for the complete VDF evaluation with possibly minimal pipeline stalls. Finally, the proposed design is coded and synthesized under the TSMC 28-nm CMOS technology. The experimental results show that our design can achieve a speedup of 3.6x compared to the optimal C++ implementation for the VDF evaluation over an advanced CPU. Moreover, compared to the state-of-the-art hardware implementation for the squaring, a key step of VDF, we achieve about 2x speedup.

Index Terms—Verifiable delay functions, class groups, extended GCD, low-latency, blockchain, ASIC.

1 Introduction

Decentralized systems have received extensive attention in both academic and industrial societies. It enables users to securely participate in a trustless system and reduces the risk of systemic failure. With the popularity of decentralized systems, there is a huge interest for timed-cryptographic primitives in blockchain area [1] [2]. In 2018, Boneh *et al*. proposed a promising scheme named verifiable delay function (VDF), which evaluates in a prescribed time and cannot be sped up by directly adding parallelism, but the result can be exponentially faster verified [3]. VDFs have numerous applications in decentralized systems. For example, they can be used to enhance the security of generating public verifiable random numbers [4], [5], [6] , which are guaranteed to be unbiased and unpredictable. VDFs are also effective as computational time-stamps and used in proof of replication [7]. Recently, the notable applications of VDFs are for designing resource-efficient blockchains, such as Chia blockchain [8], IOTA [9], and Ethereum 2.0 [10]. However, these applications are at a great potential risk. If an attacker can compute the VDF evaluation much faster than any honest user, the decentralized system can be broken. Therefore, for the security of VDF applications, the fastest implementation for VDF evaluation should be extensively studied and publicly known.

A VDF consists of a tuple of there algorithms : **Setup**, **Eval**, and **Verify**, where the evaluation is the most time-consuming and compute-intensive operation for any VDF construction. As the base module in practical applications, a stable VDF needs to satisfy two security requirements [3]:

- *Sequentiality*: Any honest participant can evaluate a function through a pre-defined t sequential steps and in no less than a prescribed time T, while an adversary has negligible chances to compute or guess the correct output in less than time T, even with many parallel processors.
- *Uniqueness*: For any input of **Eval**, exactly one output will be verified, and there is an easy way to verify the correctness of the function output.

The operation of repeated squaring over a group of unknown orders is considered to meet these requirements, *e.g.*, $a, a^2, a^4, ..., a^{2^t}$, and defined as a time-lock puzzle by Rivest, Shamir, and Wagner [11]. Two efficient VDF constructions, one proposed by Pietrzak [12] and the other by Wesolowski [13], similarly exploit the serial nature of this computation. Pietrzak's and Wesolowski's protocols are both based on exponentiation over a group of unknown order, such as the well-known RSA groups or class groups of binary quadratic forms [14]. The RSA group is a multiplicative group $\mathbb{Z}/N$ where $N = pq$ $(p, q \neq 2)$ are both unknown large primes. $\mathbb{Z}/N$ is considered as a group with an unknown order because the difficulty of calculating the group order is equivalent to the difficulty of factoring N. To guarantee that the group order is unknown, a trusted setup needs to be used to ensure that the factors chosen to generate the group are not revealed. Once the factors are obtained by adversaries, the sequentiality of the VDF will be attacked. In contrast, using class groups of binary quadratic forms omits the trusted setup [15], [16], because the order of this group is believed to be almost impossible to compute, where the discriminant of the form is a large negative prime. For this property, class groups also have been used for numerous decentralized protocols without trusted setup, including accumulators [17], timed commitments [18], and succinct

- *The authors are with the School of Electronic Science and Engineering, Nanjing University, Nanjing 210046, China. E-mail: zhudanyang10@foxmail.com, tianjing@nju.edu.cn, minghaoli@smail.nju.edu.cn, zfwang@nju.edu.cn*

non-interactive argument of knowledge [19].

VDFs have been widely investigated in the theoretical area (*e.g.*, [3], [13], [12], [20], [21], [22], [23], [24]), which greatly promotes the development of VDFs in decentralized applications. However, to make VDFs practical, sufficient research on implementation is required, especially developing specialized hardware. To date, a great deal of previous research related to VDF implementation focused on VDF in the RSA group. In 2019, VDF Alliance held a competition to reward winners for achieving obvious speed-up in operating a number of sequential squarings in the RSA group on FPGA platforms [25]. In the competition, the winner adopted the effective low-latency modular multiplication algorithm for a specific modulus proposed by $\ddot{O}zt\ddot{u}rk$ [26] and given the possible fastest implementation for modular squaring on the FPGA platforms [27]. Moreover, for the implementation on ASIC, Mert *et al.* fully analyzed and summarized the existing ultra-low latency ASIC algorithms of modular squaring for VDF evaluation in the RSA group [28].

However, in contrast to the significant progress made in designing low-latency hardware for VDF evaluation in the RSA group, to the best of our knowledge, there are very limited implementation results for VDF evaluation in the class group in the open literature, though the latter has more superiority than the former in security. Competition for fast software implementations for VDF in the class group has been held by Chia's Network and the optimized C++ implementations for low-latency VDF evaluation were obtained from this competition [29]. Later, an efficient hardware accelerator which speeds up the squaring in VDF evaluation by a factor of 2 compared to C++ implementation over a CPU has been proposed in [30]. So far, there is no complete implementation of VDF evaluation in the class group in the open literature, even though it has been used in many popular decentralized applications that do not support trusted setup [8] [18]. Therefore, there is an urgent need to give the complete low-latency implementation result for VDF evaluation in the class group and thus ensure the security and stability of related decentralized systems.

In this paper, for the first time, we propose a low-latency architecture for VDF evaluation by utilizing many algorithmic transformations and architectural optimizations to reduce the critical path delay and calculation cycles. Because the reduction and squaring need to be computed sequentially and cannot be parallelized, the only way to achieve a low-latency VDF evaluation design is speeding up the squaring and reduction within each iteration.

For squaring, the most time-consuming operation is the XGCD, which is extremely hard to speed up due to the complexity associated with large-number computations. To solve this problem, we study and modify a parallel XGCD algorithm [31] that uses small integers to replace large integers for the main calculation steps, which has been preliminary presented in our conference paper [32]. For the reduction, we improve a fast reduction algorithm [33] that greatly reduces the latency by using small numbers (such as 64-bit numbers) to approximate large numbers for calculations. For these two steps with a large discriminant, the basic arithmetic operations in them are made up of large-number divisions, multiplications, and additions, which are extremely hard to accelerate in hardware. To achieve a low-latency design for VDF evaluation, highly parallelized and pipelined architectures are devised.

The main contributions of this paper are summarized as follows:

- We introduce and optimize the squaring and reduction algorithms for VDF evaluation in the class group of binary quadratic forms to achieve ultra-low latency.
- Based on the proposed algorithms, we present a highly parallelized top-level architecture for the VDF evaluation design with possibly minimal pipeline stalls.
- A parallel XGCD algorithm is modified to be practical in hardware and the architecture for that is well designed for the low-latency implementation.
- We present extremely low-latency algorithms and implement them for large-number divisions, multiplications, and additions.
- We code the proposed design in System Verilog language and evaluate the complete design of our VDF evaluation architecture with TSMC 28-nm CMOS technology.

The hardware implementation results show that our implementation takes an average of 7.1 μs per iteration (squaring and reduction) for a 2048-bit discriminant. Our squaring achieves a speedup of 2x compared to the prior hardware implementation for VDF squaring in the class group [30], and the design for the complete VDF evaluation achieves a speedup of 3.6x compared to the software implementation over an Intel(R) Core(TM) i9-9900X @3.50GHz CPU.

The rest of this paper is organized as follows. In Section II, we introduce Wesolowski's VDF and operations of binary quadratic forms over a class group. The modified squaring algorithm, parallel XGCD algorithm, and fast reduction algorithm are detailed in Section III. In Section IV, we present the hardware architecture for the VDF evaluation. We implement our design and compare it to the prior hardware implementation and software implementation in Section V. Finally, this paper is concluded in Section VI.

2 Background

2.1 Overview of Wesolowski's VDF

To explain the role of evaluation in VDF, we first introduce the complete VDF construction. A VDF can be expressed as a triple of algorithms: setup, evaluation, and verification. The two most popular and efficient VDF constructions are Wesolowski's and Pietrzak's VDFs [20]. The setup and evaluation parts of these two constructions are the same. Therefore, we only introduce Wesolowski's VDF in this section as an example.

Wesolowski's VDF can be expressed as a triple of algorithms as follows:

- **Setup**$(\lambda, t) \rightarrow (pp)$
    1) The inputs are security parameter λ (typically 128, 192, or 256) and delay parameter t.
    2) The public parameter pp is set to be $pp := (\mathbb{G}, H, t)$. $\mathbb{G}$ is a finite Abelian group of unknown order - in this paper, $\mathbb{G}$ denotes the

class group of binary quadratic field. H is an efficiently computable hash function, which satisfies $H : \mathcal{X} \to \mathbb{G}$.

- **Eval**$(pp, x) \to (y, \pi)$
  1) The output y is computed by operating t squarings in $\mathbb{G}$, and $y \leftarrow H(x)^{2^t} \in \mathbb{G}$.
  2) The proof π is computed as described later.
- **Verify**$(pp, x, y, \pi) \to (state)$
  1) This is an operation to verify that y is indeed the correct output for x. If y is the correct output, then output $accept$, otherwise output $reject$.

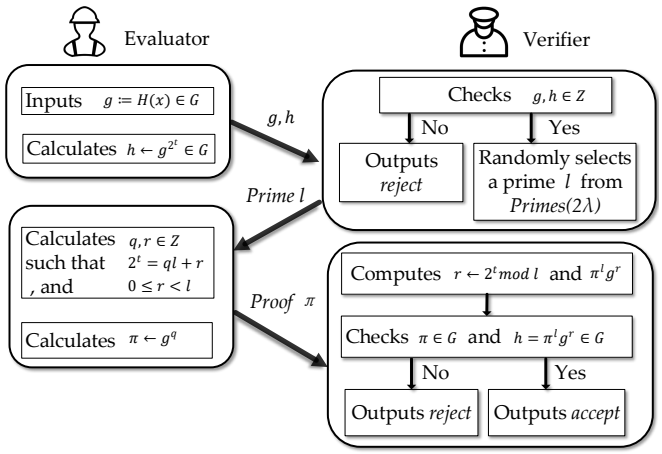


Fig. 1: Overview of evaluation and verification in Wesolowski's VDF.

Wesolowski proposed a succinct method for verifying $y = H(x)^{2^t}$ in [13] and an overview of Wesolowski's VDF is shown in Fig. 1. Fistly, a public variable $g := H(x)$ is given to the VDF evaluator as a base element, then $h := y = g^{2^t} \in \mathbb{G}$ is calculated and sent to the VDF verifier with g. After that, the verifier should check whether $g, h \in Z$. If yes, the verifier will randomly select a prime l from $Primes(2\lambda)$ and send it to the evaluator. $Primes(2\lambda)$ is a set containing the first $2^{2\lambda}$ primes, namely 2, 3, 5, 7, *etc*. If no, the verifier will output *reject* and end the verification. With the yes answer, the evaluator computes the proof $\pi = g^q \in \mathbb{G}$ where $q = \lfloor \frac{2^t}{l} \rfloor$ and sends it to the verifier. Finally, the verifier computes $r \leftarrow 2^t \bmod l$ which only takes $log_2 t$ multiplications in $\mathbb{Z}/l$. Meanwhile, the verifier also needs to compute $\pi^l g^r$ which consists of two small exponentiations in $\mathbb{G}$, and outputs *accept* if $\pi \in \mathbb{G}$ and $h = \pi^l g^r \in \mathbb{G}$.

In general, the evaluator needs to perform t sequential squarings for $h \leftarrow g^{2^t}$ and compute a proof $\pi \leftarrow g^q$. The computation of proof π can be sped up by using parallism and storing the intermediate data in the calculation of g^{2^t}, *e.g.*, g^2, g^4, g^8, *etc* [20]. The guaranteed time T is for the sequential operation $h \leftarrow g^{2^t}$ and the total time of evaluation is $(1 + \epsilon)$ T, where ϵ is small and corresponds to the number of processors. When the evaluator uses s processors, the total evaluation time is around $(1 + \frac{1}{20s})$ T. As a result, the calculation time of VDF evaluation is mainly determined by the number of squarings t and the runtime for one squaring. Since t is a public parameter for all evaluators, we are motivated to propose a high-speed implementation of squaring for VDF evaluation publicly.

2.2 Binary Quadratic Forms

The operations of a VDF are based on groups of unknown orders which are usually be an RSA group or a class group of binary quadratic forms. Using class groups of binary quadratic forms is more suitable for decentralized systems since it omits a trusted setup compared to using RSA groups. However, the operations in the class groups of binary quadratic forms are more complex than the same operations in RSA groups. We will succinctly introduce several concepts that are related to the binary quadratic forms that VDF construction uses. For a more detailed information of binary quadratic forms, please refer to [16] and [34].

1) $Form$
   A binary quadratic form is defined as:
   $$f(x, y) = ax^2 + bxy + cy^2, \tag{1}$$
   where $a, b, c \in \mathbb{R}$ and $a, b, c \neq 0$. When $a, b, c \in \mathbb{Z}$, this form is also called integral binary quadratic form which is used in Chia's VDF. Moreover, $f(x, y)$ is written as $f = (a, b, c)$ and called a $form$.
2) $Discriminant$
   The discriminant of a form f is $\Delta(f) = b^2 - 4ac$, where the discriminant Δ is a negative prime and $|\Delta| \equiv 3 \bmod 4$. In particular, $|\Delta|$ is sufficiently large, e.g., width of 2048 bits, making the order of the class group effectively unknown. If and only if $\Delta < 0$ and $a > 0$, the form f is positive definite.
3) $Normal$
   A form $f = (a, b, c)$ is called normal if $-a < b \le a$.
4) $Reduced$
   A positive definite form $f = (a, b, c)$ is called reduced, if it is normal and $a \le c$, and when $a = c$ then $b \ge 0$.
5) $Composition$
   Consider two binary quadratic forms that have the same discriminant:
   $$f_1 = ax_1^2 + bx_1y_1 + cy_1^2 \tag{2}$$
   and
   $$f_2 = \alpha x_2^2 + \beta x_2y_2 + \gamma y_2^2, \tag{3}$$
   where (x_1, y_1) and (x_2, y_2) are independent sets of variables. The f_1 and f_2 can also be written as $f_1 = (a, b, c)$ and $f_2 = (\alpha, \beta, \gamma)$, respectively. The composition is to find a form f_3 such that
   $$f_1 f_2 = AX^2 + BXY + CY^2 = f_3. \tag{4}$$
   In particular, when $f_1 = f_2$, the composition of $f_1 f_2$ is the squaring of f_1 or f_2.

2.3 VDF Evaluation in the Class Group

The VDF evaluation is mainly a serail calculation for $h := g^{2^t} \in \mathbb{G}$ which includes squaring and reduction operations. In this section, we will introduce these two algorithms separately.

2.3.1 Squaring Algorithm in the Class Group

Let a form $f = (a, b, c)$, the squaring is to calculate

$$f^2 = f' = (A, B, C), \tag{5}$$

and the squaring algorithm uses set (a, b, c) to obtain the value of (A, B, C). The detailed steps of the squaring algorithm are shown in Alg. 1, and the proof of this algorithm can be found in [34].

Algorithm 1: Squaring Algorithm in the Class Group [34]

Input: $f = (a, b, c)$, where $a, b, c \in \mathbb{Z}$
Output: $f' = f^2 = (A, B, C)$, where $A, B, C \in \mathbb{Z}$

1 $(d, e, g) \leftarrow$ XGCD (*a*,*b*) ▷satisfy $bd + ae = g$
2 $\rho \leftarrow \lfloor \frac{c}{g} \rfloor$, $r \leftarrow c\%g$
3 **if** $r \neq 0$ **then**
4 | Return:no solution
5 **else**
6 | $\mu \leftarrow \rho d\%a$
7 **end**
8 $A \leftarrow a^2$, $B \leftarrow b - 2a\mu$, $C \leftarrow \mu^2 - \frac{b\mu - c}{a}$

First, the XGCD algorithm is applied to compute the Bézout coefficients d and e as well as the greatest common divisor (GCD) g of (a, b). Then, a division c/g is performed to compute the quotient ρ and the remainder r, and the algorithm continues when $r = 0$. After that, an auxiliary value μ is computed by the division $\rho d\%a$. Finally, $f' = (A, B, C)$ is obtained by one division, four multiplications, and three subtractions.

In practical applications, *e.g.*, Chia's VDF, the discriminant $\Delta(f) = b^2 - 4ac = -p$, where p is a prime number, resulting in $GCD(a, b) = g = 1$ in all cases. Therefore, the step 2-7 in Alg. 1 can be removed and μ is directly calculated by $\mu = cd\%a$.

2.3.2 Reduction Algorithm in the Class Group

Since $f(a, b, c)$ is repeatedly squared, the values of a, b, and c will keep growing until too large to be calculated. To avoid this situation, each squaring needs to be followed by a reduction operation. When $-a < b \leq a$, $a \leq c$, a positive definite form $f = (a, b, c)$ is called reduced. The reduction algorithm in the class group is shown in Alg. 2. First, a form $f(a, b, c)$ is normalized by the normalization algorithm which is shown in Alg. 3. The main operation in normalization is a large-number division for a parameter r, and $\eta(f)$ is a normalization operator that makes f normal. When f is normalized, another large-number division is applied for a reduced parameter $s \leftarrow \lfloor \frac{c+b}{2c} \rfloor$, and f is updated by s. After each updating, f needs to be tested to determine whether it is reduced. When f is not reduced, repeat steps 2-6. According to our test results, for the Chia's VDF with a 2048-bit discriminant, the loop needs to be repeated approximately 200 times.

Algorithm 2: Reduction Algorithm in the Class Group

Input: $f(a, b, c),\ \Delta < 0,\ a > 0$
Output: $f(a, b, c), -a < b \leq a,\ a \leq c$, and if $a = c$ then $b \geq 0$

1 Normalization(f)
2 **while** f *is not reduced* **do**
3 | $s \leftarrow \lfloor \frac{c+b}{2c} \rfloor$
4 | $\rho(f) \leftarrow (c, -b + 2sc, cs^2 - bs + a)$
5 | $f \leftarrow \rho(f)$
6 **end**
7 **Return** f

Algorithm 3: Normalization Algorithm in the Class Group

Input: $f(a, b, c),\ \Delta < 0,\ a > 0$
Output: $f(a, b, c), -a < b \leq a$

1 $r \leftarrow \lfloor \frac{a-b}{2a} \rfloor$
2 $\eta(f) \leftarrow (a, b + 2ra, ar^2 + br + c)$
3 $f \leftarrow \eta(f)$
4 **Return** f

3 Improved Algorithm for VDF Evaluation

For a reduced form f, we have

$$\begin{aligned} |\Delta| &= 4ac - b^2 && (\Delta < 0) \\ &\geq 4a(a) - a^2 && (-a < b \leq a, a \leq c) \\ &\geq 3a^2, \end{aligned} \tag{6}$$

and so,

$$a \leq \sqrt{\frac{|\Delta|}{3}}. \tag{7}$$

Supposing the discriminant Δ of the form f is a $2N$-bit large number (usually be 512 or 1024), the bit width of a is N according to Eq. (7). Moreover, since the form f is reduced, the bit width of b is also N and that of c is $2N$.

3.1 Improved Squaring Algorithm in the Class Group

As shown in Alg. 1, the squaring algorithm includes the XGCD algorithm, divisions, multiplications, and subtractions of large numbers. The most time-consuming operation in squaring is the calculation of XGCD, and the second is the $3N$-bit division $\mu = cd\%a$. However, we notice that the division $\mu = cd\%a$ cannot start until the XGCD is finished, since d is the output of the XGCD algorithm.

To reduce the computation time for the squaring algorithm, we separate the $3N$-bit division $\mu = cd\%a$ into two $2N$-bit divisions $r = c\%a$ and $\mu = rd\%a$. The division $r = c\%a$ can be calculated simultaneously with XGCD since c and a are the initial input of squaring. In addition, since the divisors of $r = c\%a$ and $\mu = rd\%a$ are both a, the division $\mu = rd\%a$ follows XGCD and can be simplified as multiplication $\mu = rd \times \frac{1}{a}$ by saving the value $\frac{1}{a}$ in the calculation of $r = c\%a$. In summary, we modify the squaring algorithm from Alg. 1 to Alg. 4. The outputs B and C are also calculated in parallel, which further reduces the total calculation time of squaring.

Algorithm 4: Modified Squaring Algorithm for Low-latency Hardware Implementation

Input: $f = (a, b, c),\ where\ a, b, c\ \in \mathbb{Z}$
Output: $f' = f^2 = (A, B, C),\ where\ A, B, C\ \in \mathbb{Z}$
do in parallel
  $(d, e, 1) \leftarrow$ XGCD (*a*,*b*) ▷satisfy $bd + ae = 1$
  $x \leftarrow \frac{1}{a}$ ▷saving $\frac{1}{a}$
  $r \leftarrow c - \lfloor c \times x \rfloor \times a$
  $A \leftarrow a^2$
end
$\mu \leftarrow rd\%a$
do in parallel
  $B \leftarrow b - 2a\mu$
  $C \leftarrow \mu^2 - \frac{b\mu - c}{a}$
end

3.2 Improved XGCD Algorithm

The XGCD algorithm is used to calculate Bézout coefficients d and e which satisfy $bd + ae = GCD(a, b) = 1$ in Alg. 4. The commonly used XGCD algorithms are the binary XGCD algorithm and the extended Euclidean algorithm (EEA) [35]. For large numbers, the binary XGCD algorithm requires too many iterations to achieve low-latency. In addition, the original EEA includes extremely time-consuming operations of large numbers. To solve this problem, Lehmer proposed an algorithm based on EEA in [36], which uses the leading bits of large integers for the main calculation steps and greatly reduces the computational complexity. However, it is still difficult to speed up Lehmer's EEA by using parallelism because of the strong data dependency. Inspired by Lehmer's idea, Sidi [31] proposed a parallel XGCD algorithm to further accelerate this operation. In this paper, we modify Sidi's algorithm to be more practical in hardware and devise low-latency architecture for the parallel XGCD algorithm.

3.2.1 XGCD Reduction

Considering a pair of non-negative integers (a, b), where $a \geq b$, most of GCD algorithms use one or more transformations $(a, b) \rightarrow (a', b')$ that serially reduce the size of current pairs (a, b) until a pair $(a', 0)$ is eventually obtained. The last value a' is the GCD of (a, b). For XGCD computation that includes finding Bézout coefficients (x, y) such that $ax + by = GCD(a, b)$, a 2×2 matrix M is used, which is defined by:

$$(a, b) \rightarrow (a', b') = (a, b) \times M, \tag{8}$$

where $det(M) = \pm 1$. In this paper, distinguishing from "reductions" in the class groups, these transformations are called " XGCD reductions".

3.2.2 Parallel XGCD Algorithm

The parallel XGCD algorithm is shown in Alg. 5, where the inputs are a, b, and m. The parameter m is related to the degree of parallelism, *e.g.*, when $m = 3$, the parallelism of XGCD algorithm is $2^3 = 8$. For hardware implementations, m is a fixed number determined by software simulation results.

This algorithm contains three XGCD reduction algorithms for three different situations, where the input (a, b) is reduced by them. A matrix N needs to be initialized for calculating Bézout coefficients, and updated by matrix M after each XGCD reduction as shown in step 9 in Alg. 5.

Algorithm 5: Parallel XGCD Algorithm

Input: $a \geq b > 0, m$
Output: g, d, e ▷where $bd + ae = g = GCD(a, b)$
$N \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
while $b > 2^{20}$ **do**
  $(n, p) \leftarrow$the number of significant bits of (a, b)
  if $n - p < m - 1$ **then**
    $(M, a, b) \leftarrow PERA(a, b, m)$
  else
    $(M, a, b) \leftarrow \rho\text{-}Euclid(a, b, m)$
  end
  $N \leftarrow N \times M$
end
$(g, d', e') \leftarrow EEA(a', b')$
$(d, e) \leftarrow (d', e') \times N$

We use n and p to represent the number of significant bits of a and b, respectively, where $2^{n-1} \leq a < 2^n$ and $2^{p-1} \leq b < 2^p$. As shown in Alg. 5, when $b > 2^{20}$ and $n - p < m - 1$, a parallel XGCD reduction algorithm (PERA) [31] detailed in Alg. 6 is adopted.

Algorithm 6: Parallel XGCD Reduction Algorithm

Input: $a \geq b > 0, m, n, p$
Output: $M \leftarrow \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}, \begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$
$\lambda \leftarrow 2m + n - p + 2$
$a_1 \leftarrow \lfloor \frac{a}{2^{p-\lambda}} \rfloor, b_1 \leftarrow \lfloor \frac{b}{2^{p-\lambda}} \rfloor, d \leftarrow \frac{1}{b_1}$
for $i = 1 : 2^m$ **do**
  do in parallel
    $q_i \leftarrow i \times a_1 d, r_i \leftarrow i a_1 - q_i b_1, s_i \leftarrow b_1 - r_i$
    if $r_i < \frac{b_1}{2^m}$ **then**
      $X \leftarrow r_i, (x_2, y_2) \leftarrow (i, -q_i)$
    end
    if $s_i < \frac{b_1}{2^m}$ **then**
      $Y \leftarrow s_i, (x_1, y_1) \leftarrow (-i, q_i + 1)$
    end
  end
  if $X > Y$ **then**
    $(x_2, y_2) \leftarrow (x_1, y_1), R_2 \leftarrow a x_2 + b y_2$
  end
  if $(R_2 < 0)$ **then**
    $(x_2, y_2) \leftarrow (-x_2, -y_2), R_2 \leftarrow -R_2$
  end
end
$(x_1, y_1) \leftarrow$Bézout (x_2, y_2)
$R_1 \leftarrow a x_1 + b y_1$
if $(R_1 < 0)$ **then**
  $(x_1, y_1) \leftarrow (-x_1, -y_1), R_1 \leftarrow -R_1$
end

The PERA is used for the case where a and b are close to each other, which is also the most complex case for XGCD

reduction. The inputs of PERA are a, b, m, n, and p, where $n - p < m - 1$ and $2p > n + 2m + 2$, and the outputs are a matrix M where $|x_1y_2 - x_2y_1| = 1$ and a pair (R_1, R_2) which satisfy $GCD(R_1, R_2) = GCD(a, b)$.

The core idea of PERA is to replace large numbers with their leading bits and reduce the number of iterations by parallel computing. As shown in Alg. 6, first, a parameter λ is obtained for the calculation of small numbers a_1 and b_1. At the same time, the reciprocal of b_1 is calculated and used in step 5, which makes the parallel divisions be replaced by parallel multiplications. Step 3 to step 19 in Alg. 6 are computed in parallel and the degree of parallelism is equal to 2^m. After step 19 is finished, a set of parameters (x_2, y_2, R_2) is obtained that satisfies $R_2 = ax_2 + by_2$. Then another set (x_1, y_1, R_1) that satisfies $R_1 = ax_1 + by_1$ is calculated by the Bézout algorithm as shown in Alg. 7.

The input of Bézout algorithm are (x_2, y_2) where $GCD(x_2, y_2) = 1$, and the output are (x_1, y_1) where $x_1|y_2| + y_1|x_2| = 1$. This algorithm has similar parallel operations as steps 3-12 in Alg. 6 and the degree of parallelism is equal to 2^{m-1}.

Algorithm 7: Bézout Algorithm

Input: $|x| \leq |y|$ ▷where $GCD(x, y) = 1$
Output: u, v ▷where $u|y| + v|x| = 1$

for $i = 1 : 2^{m-1}$ **do**
  do in parallel
    $q_i \leftarrow \lfloor i \times |y| \times \frac{1}{|x|} \rfloor$, $r_i \leftarrow i|y| - q_i|x|$
    $s_i \leftarrow |x| - r_i$
    if $r_i == 1$ **then**
      $(u, v) \leftarrow (i, -q_i)$
    end
    if $s_i == 1$ **then**
      $(u, v) \leftarrow (-i, q_i + 1)$
    end
  end
end

When $b > 2^{20}$ and $n - p \geq m - 1$, an efficient XGCD reduction algorithm, named ρ-Euclid algorithm is used. As shown in Alg. 8, the ρ-Euclid algorithm also uses the leading bits of large numbers a and b for the division $q \leftarrow \lfloor \frac{a_1}{b_1} \rfloor$, which is also the main operation. This algorithm is simple but only valid when a is much larger than b.

After serial PERA or ρ-Euclid XGCD reductions, when b is reduced to smaller than 2^{20}, the traditional EEA described in Alg. 9 is employed. Since both a' and b' of EEA are small numbers, operations involved in this algorithm are simple. After several iterations, the output $g = GCD(a', b')$ is obtained, and it is also the final $GCD(a, b)$ of Alg. 5. In addition, d' and e' are calculated as $b'd' + a'e' = g$, and the final Bézout coefficients (b, e) of (a, b) are computed by $(d, e) \leftarrow (d', e') \times N$.

3.3 Modified Reduction Algorithm in the Class Group

As seen in Alg. 2, reduction includes complex calculation of large-number divisions and multiplications so that the reduction is extremely slow to compute. To solve this problem, the Chia company hosted competition in 2019, and the fast reduction algorithm was proposed by Akashnil Dutta [33]. According to Akashnil's test, the reduction operations in repeated squaring in the class group of binary quadratic forms are sped up by almost 5x by applying the fast reduction algorithm. The main idea of this algorithm is using the leading bits of large numbers for the reduction iterations detailed in Alg. 10. In this paper, we modify this fast reduction algorithm to implement our hardware accelerator.

Algorithm 8: ρ-Euclid Algorithm

Input: $a \geq b > 0$, m, n, p ▷where $n - p \geq m - 1$
Output: $M \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & q \end{pmatrix}$, $\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} b \\ R_\rho \end{pmatrix}$

$\lambda \leftarrow n - p + 2$
$a_1 \leftarrow \lfloor \frac{a}{2^{p-\lambda}} \rfloor$, $b_1 \leftarrow \lfloor \frac{b}{2^{p-\lambda}} \rfloor$
$q \leftarrow \lfloor \frac{a_1}{b_1} \rfloor$
$R_\rho \leftarrow a - qb$

Algorithm 9: Extended Euclidean Algorithm (EEA)

Input: $a' \geq b' > 0$
Output: g, d', e' ▷where $b'd' + a'e' = g$

$r_0 \leftarrow a'$, $r_1 \leftarrow b'$
$s_0 \leftarrow 1$, $s_1 \leftarrow 0$, $t_0 \leftarrow 0$, $t_1 \leftarrow 1$
$k \leftarrow 1$
while $r_{k+1} \neq 0$ **do**
  $q_k \leftarrow \lfloor \frac{r_{k-1}}{r_k} \rfloor$, $r_{k+1} \leftarrow r_{k-1} - q_k r_k$
  $s_{k+1} \leftarrow s_{k-1} - q_k s_k$
  $t_{k+1} \leftarrow t_{k-1} - q_k t_k$
  $k \leftarrow k + 1$
end
$g \leftarrow r_k$, $d' \leftarrow s_k$, $e' \leftarrow r_k$

In Alg. 10, the input f needs to be tested to determine whether it is reduced by the test function shown in Alg. 11. The non-reduced conditions are simplified as $|a| < |b| \parallel |c| < |b|$, and the reduced conditions are increased with minor transformation. When f is reduced, output $flag = 1$; otherwise output $flag = 0$.

When f is not reduced, execute the loop in Alg. 10 for reduction. First, a leading one detector (LOD) is used to calculate the effective bits of the inputs (a, b, c). After a_{num}, b_{num}, and c_{num} are obtained, the maximum and minimum of these values are calculated. Then, determine whether $max_{num} - min_{num}$ is greater than 31, if not, normalize f. For the normalization algorithm shown in Alg. 3, the large-number division is an extremely complex calculation for hardware implementation. We notice that the normalization algorithm makes variables (a, b, c) close to each other, and hence the iterations of reduction will be decreased. Therefore, we choose specified parameters to update (a, b, c), which achieves the same target without the large-number division shown in Alg. 12. As a result, the large-number division is replaced by shifts, subtractions, and an addition of large numbers.

After normalization, we use (x, y, z) to represent the most significant 64 bits of (a, b, c), respectively. Next, adjust the values of (x, y, z) according to $(a_{num}, b_{num}, c_{num})$ by shifting. Meanwhile, small auxiliary coefficients (u, v, m, n)

Algorithm 10: Fast Reduction Algorithm

Input: $f(a,b,c),\ \Delta<0,\ a>0$
Output: $f(a,b,c), -a<b\le a,\ a\le c$, and if $a=c$ then $b\ge 0$

1 $flag \leftarrow$ `TestReduced(`f`)`
2 **if** $flag == 1$ **then**
3 **Return** f
4 **else**
5 **repeat**
6 $(a_{num}, b_{num}, c_{num}) \leftarrow$ `LOD(`a,b,c`)`
7 $max_{num} \leftarrow$ `MAX(`$a_{num}, b_{num}, c_{num}$`)`
8 $min_{num} \leftarrow$ `MIN(`$a_{num}, b_{num}, c_{num}$`)`
9 **if** $max_{num} - min_{num} > 31$ **then**
10 `ModifiedNorm(`f`)`
11 **else**
12 $(x,y,z) \leftarrow$ the most significant 64 bits of (a,b,c)
13 $x >> (max_{num} - a_{num} + 1)$
14 $y >> (max_{num} - b_{num} + 1)$
15 $z >> (max_{num} - c_{num} + 1)$
16 $u \leftarrow 1, v \leftarrow 0, m \leftarrow 0, n \leftarrow 1$
17 **while** $x \ge z \&\& z \le 0$ **do**
18 $\delta \leftarrow (y \ge 0)?(y+z)/2z : -(-y+z)/2z$
19 $(x', y', z') \leftarrow (z, -y+2\delta z, x-\delta y+\delta^2 z)$
20 $(u', v', m', n') \leftarrow (v, -u+\delta v, n, -m+\delta n)$
21 $(x,y,z) \leftarrow (x', y', z')$
22 $(u,v,m,n) \leftarrow (u', v', m', n')$
23 **end**
24 $a' \leftarrow u^2 a + umb + m^2 c$
25 $b' \leftarrow 2uva + (un+vm)b + 2mnc$
26 $c' \leftarrow v^2 + vnb + n^2 c$
27 $(a,b,c) \leftarrow (a', b', c')$
28 **end**
29 **until** f *is reduced;*
30 **end**
31 **Return** f

are initialized and calculated in the small loop steps 17-22 for updating (a,b,c). The operators involved in this loop are all with 64 bits, which are much smaller than those in Alg. 2. When $x \ge z \&\& z \le 0$, output coefficients (u,v,m,n) and update (a,b,c) until f is reduced.

4 Hardware Architecture

4.1 Top-level Hardware Architecture for VDF Evaluation

The overall architecture of the VDF evaluation is shown in Fig. 2, which mainly includes the squaring module and reduction module. For the squaring operation in an iteration, a reduced form $f = (a,b,c)$ is imported into the squaring module, and after a series of operations, $f' = (A,B,C) = f^2$ is obtained and used as the input of the reduction module. Then the form f' is reduced to be $\tilde{f} = (\tilde{A}, \tilde{B}, \tilde{C})$ through reduction module. Supposing that the discriminant Δ of the form f is a $2N$-bit large number (N is 1024 in our design), the inputs (a,b,c) of the squaring module are N bits, N bits, and $2N$ bits, respectively. The outputs (A,B,C) of the squaring module all both $2N$ bits,

Algorithm 11: Test Reduced Algorithm

Input: $f(a,b,c),\ \Delta<0,\ a>0$
Output: $f(a,b,c), flag$

1 **if** $|a| < |b| \ \|\ |c| < |b|$ **then**
2 **Return** $f(a,b,c)$ and $flag = 0$
3 **else**
4 **if** $a > c$ **then**
5 $f(a',b',c') = f(c,-b,a)$
6 **else if** $a == c \&\& b < 0$ **then**
7 $f(a',b',c') = f(a,-b,c)$
8 **else**
9 $f(a',b',c') = f(a,b,c)$
10 **Return** $f(a',b',c')$ and $flag = 1$
11 **end**

Algorithm 12: Modified Normalization Algorithm

Input: $f(a,b,c),\ \Delta<0,\ a>0$
Output: $f(a,b,c)$

1 $\eta(f) \leftarrow (c, 4c-b, a-2b+4c)$
2 $f \leftarrow \eta(f)$
3 **Return** f

and the outputs $(\tilde{A}, \tilde{B}, \tilde{C})$ of the reduction module are N bits, N bits, and $2N$ bits, respectively. A complete VDF evaluation needs to operate 2^t squarings and reductions, where t is a delay parameter.

4.2 Architecture for the Squaring Algorithm

According to Alg. 4, the architecture of the squaring algorithm is shown in the left dashed block in Fig. 2. It contains a parallel XGCD module, a Newton's iteration (NI) module for computing reciprocal, several multipliers, and adders.

The operations in the squaring are listed in TABLE 1. Since the maximum bits of (a,b,c) are $(N,N,2N)$, we can obtain the sizes of different operations. In addition to the parallel XGCD module and NI module, the squaring module contains 10 multiplications and 5 additions. As shown in TABLE 1, #1, #7, and #11 are N×N-bit multiplications, and #4, #9 are 2N×N-bit multiplications. In particular, #3, #8, and #13 are used for calculating quotients, and since the divisors are 2N-bit, 2N×2N-bit multiplications are required. Since these multiplications have dependencies that cannot be computed in parallel, we only need to apply four N×N-bit multipliers. The 2N×N-bit multiplier can be composed of two N×N-bit multipliers, and the 2N×2N-bit multiplier can be composed of four N×N-bit multipliers. In addition, #5, #10, #12, and #14 are all 2N-bit subtractions that can be computed by 2N-bit adders.

Considering the dependencies between operations in the squaring, the longest path in the squaring is: #6→ #7→ #8→ #9→ #10→ #11→ #12→ #13→ #14. The longest path of the squaring module is shown as the red dashed line in Fig. 2, which includes the delay of the parallel XGCD module, five multipliers, and three $2N$-bit adders. Therefore, to reduce the calculation time for squaring, a low-latency XGCD module and fast large-number multipliers and adders are required.

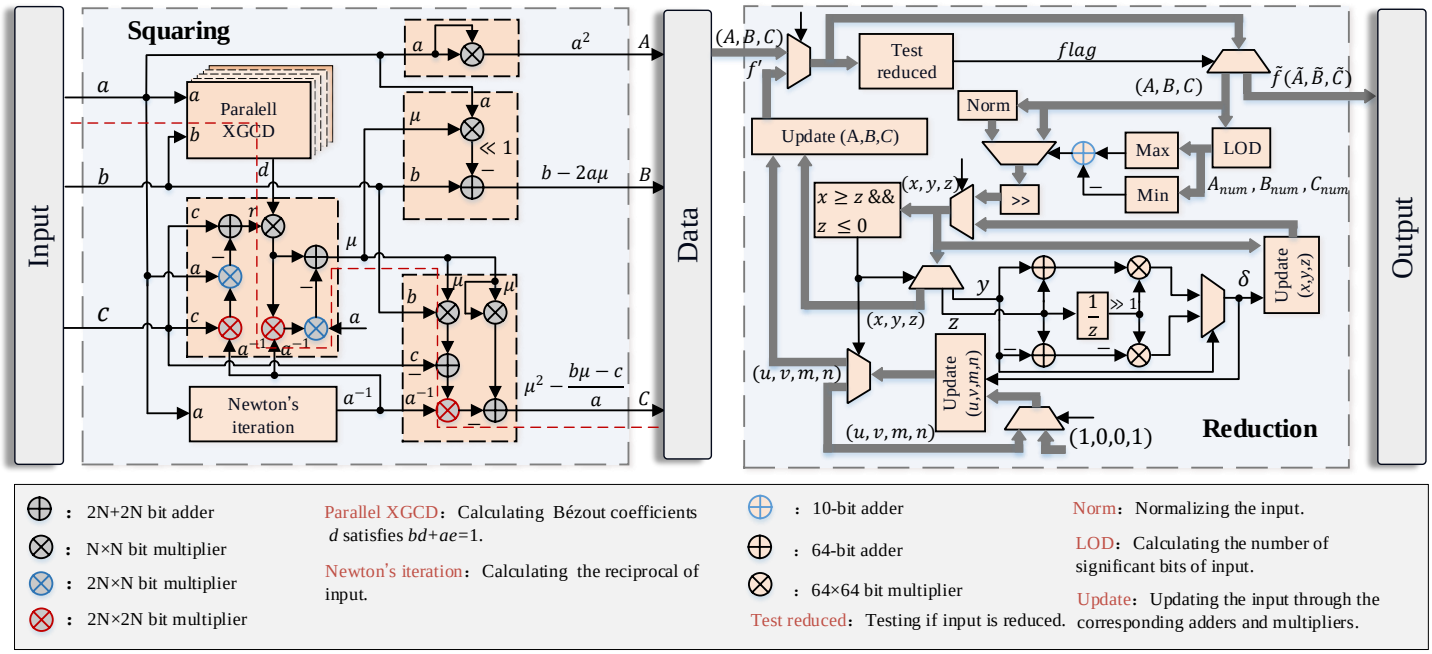


Fig. 2: The overall hardware architecture for the VDF evaluation.

TABLE 1: Bit-width and Dependencies of Operations in Squaring

Number	Operation	Size (bits)	Dependence
#1	$A \leftarrow a^2$	$N \times N$	\
#2	$a^{-1} \leftarrow NI(a)$	\	\
#3	$q_1 \leftarrow c \times a^{-1}$	$2N \times 2N$	#2
#4	$q_1 \times a$	$2N \times N$	#3
#5	$r_1 \leftarrow c - q_1 a$	$2N - 2N$	#4
#6	$(d, e) \leftarrow XGCD(a, b)$	\	\
#7	$r_1 \times d$	$N \times N$	#5, #6
#8	$q_2 = r_1 d \times a^{-1}$	$2N \times 2N$	#2, #7
#9	$q_2 \times a$	$2N \times N$	#8
#10	$\mu \leftarrow r_1 d - q_2 a$	$2N - 2N$	#7, #9
#11	$a \times \mu, b \times \mu, \mu^2$	$N \times N$	#10
#12	$b\mu - c, B \leftarrow b - 2a\mu$	$2N - 2N$	#11
#13	$(b\mu - c) \times a^{-1}$	$2N \times 2N$	#12
#14	$C \leftarrow \mu^2 - (b\mu - c) \times a^{-1}$	$2N - 2N$	#13

4.3 Architecture for the Reduction Algorithm

According to Alg. 12, the architecture for the reduction algorithm is shown in the right dashed block in Fig. 2. The reduction needs a test reduced module which only contains several 2N-bit (N is 1024 in our design) comparators and simple control logic to determine if the input $f'(A, B, C)$ is reduced, and outputs the final reduced result. If the input is not reduced, then $f'(A, B, C)$ needs to be updated. First, LODs are used to calculate the number of significant bits $(A_{num}, B_{num}, C_{num})$ of (A, B, C). Next, 12-bit comparators are applied to find the maximum max_{num} and the minimum min_{num} of the three values $(A_{num}, B_{num}, B_{num})$. If $max_{num} - min_{num}$ is greater than 31, then (A, B, C) will be input to the Norm module. The Norm module only contians 2048-bit adders and used to normalize the input based on Alg. 12. Both input and output of this module are $f(A, B, C)$. After normalization, a loop is used to calculate the auxiliary parameters (x, y, z) and (u, v, m, v) for updating (A, B, C), which mainly requires a 64-bit divisor, 64-bit multipliers, and 64-bit adders. When $x \geq z \&\& z \leq 0$, the loop stops and (A, B, C) are updated by 64×2048-bit multipliers and 2048-bit adders until the reduced result $\tilde{f}(\tilde{A}, \tilde{B}, \tilde{C})$ is obtained.

4.4 Architecture for the Parallel XGCD Algorithm

The parallel XGCD module is the most complex and time-consuming module in the squaring block. The computation of this algorithm includes the parallel XGCD reduction algorithm, the ρ-Euclid algorithm, and the EEA algorithm for repeated XGCD reductions. The total time for XGCD is mainly decided by the number of XGCD reductions and the delay of one reduction. As shown in Alg. 5, the parameter m is closely related to the degree of parallelism. For example, when $m = 3$, the degree of parallelism of the parallel XGCD reduction module is $2^m = 8$. Intuitively, the larger m is, the less the XGCD reduction is required. At the same time, the larger m is, the more complex the XGCD reduction modules are. Therefore, we choose the parameter by using Monte Carlo simulation to obtain a good trade-off. According to our simulation, when inputs (a, b) are 1024-bit numbers, m set to 8 can achieve a relatively small latency for the parallel XGCD module.

According to Alg. 5, the overall architecture of the parallel XGCD algorithm is shown in Fig. 3, which mainly includes multipliers, adders, and the Newton iteration module for divisions. First, an LOD is used to calculate the number of significant bits (n, p) of inputs (a, b). Then, according to the relationship between n and p, the control signal generator will update the $Start$ 1, $Start$ 2, or $Start$ 3

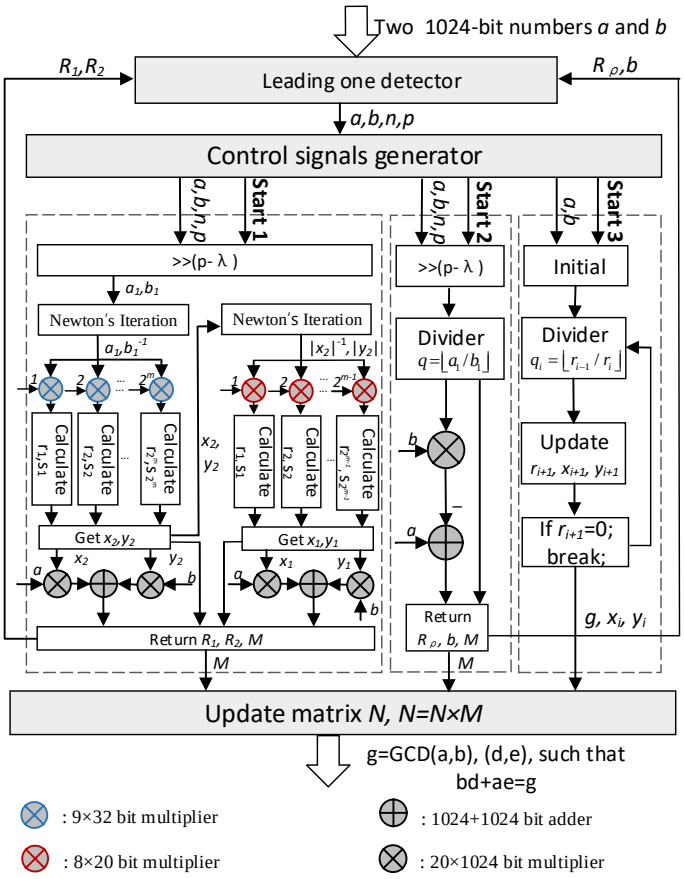


Fig. 3: The architecture for the parallel XGCD algorithm.

signal to select the parallel XGCD reduction module, the ρ-Euclid module, or the EEA reduction module, respectively. At the same time, the matrix N is also updated by 2×2 matrix M.

The parallel XGCD reduction module is the most computationally intensive and has the largest area among the three modules. To get x_2 and y_2 for updating the R_2, it needs a Newton's iteration unit for calculating b^{-1}, 2^m 9×32-bit multipliers, and corresponding 2^m units for calculating the quotients s_i and remainders r_i in parallel. To get x_1 and y_1 for updating the R_1, it also needs a Newton's iteration unit for calculating $|x_2|^{-1}$, 2^{m-1} 8×20-bit multipliers, and corresponding 2^{m-1} units for calculating the quotients s_i and remainders r_i in parallel. When (x_1, y_1) and (x_2, y_2) are obtained, the new pair (R_1, R_2) are calculated by 20×1024-bit multipliers and 1024-bit adders. Compared to the parallel XGCD reduction module, the ρ-Euclid module is simple, it only contains a divider of small number, a 20×1024-bit multiplier and a 1024-bit adder. When the signal is $Start_3$ for the EEA reduction module, it means that the input (a, b) has been reduced to small numbers ($b \leq 2^{20}$), so the EEA reduction module only needs a divider, multipliers and adders for small numbers. When the EEA module finishes, the whole XGCD module stops and the final results (x, y) are obtained.

4.5 Architecture of Multipliers, Dividers, and Adders

4.5.1 Architecture of the Large-Number Multiplier

Since large-number multiplications are extremely time-consuming operations, it is necessary to accelerate them to achieve a low-latency implementation of VDF evaluation. Hence, we design a low-latency large-number multiplier by utilizing the Karatsuba multiplication approach [37].

For the multiplication of n-bit A and B, they can be written as:

$$A = A_H 2^{n/2} + A_L,\ B = B_H 2^{n/2} + B_L. \tag{9}$$

Then, the multiplication $C = A \times B$ equals:

$$\begin{aligned} C &= (A_H 2^{n/2} + A_L) \times (B_H 2^{n/2} + B_L) \\ &= A_H B_H 2^n + (A_H B_L + A_L B_H) 2^{n/2} + A_L B_L. \end{aligned} \tag{10}$$

Moreover, based on the Karatsuba multiplication scheme, the multiplication can be written as:

$$\begin{aligned} C &= A_H B_H 2^n + A_L B_L \\ &+ [(A_H + A_L)(B_H + B_L) - A_H B_H - A_L B_L] 2^{n/2}. \end{aligned} \tag{11}$$

As a result, for each Karatsuba multiplication performed, the number of sub-multiplication is reduced from 4 to 3. If we apply k-level Karatsuba multiplication, an $n \times n$-bit multiplier can be split into 3^k $(n/2^k) \times (n/2^k)$-bit multipliers. In our design, we apply 6-level Karatsuba multiplication to 1024×1024-bit multiplier, and the multiplier is deposed to 729 16×16-bit multipliers. The architecture for this Karatsuba multiplier is shown in Fig. 4. By splitting the large-number multiplication into multiplications of small numbers and additions of large numbers, the critical path delay can be effectively reduced.

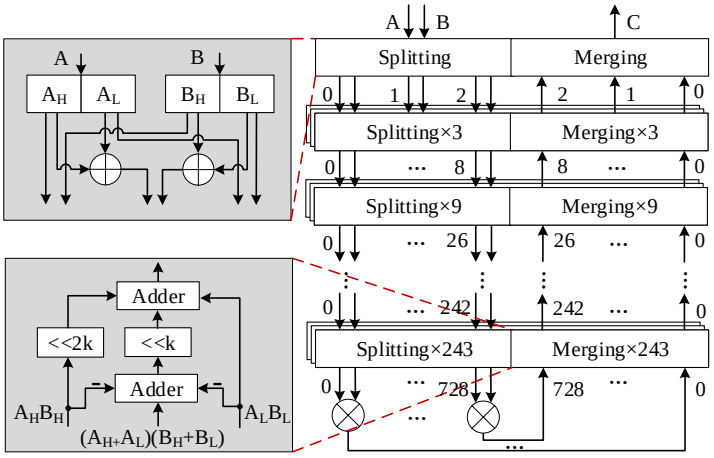


Fig. 4: Architecture for the Karatsuba multiplier.

4.5.2 Architecture of Divider

Division $q = n/a$ can be written as the product of the dividend and the reciprocal of the divisor: $q = n \times \frac{1}{a}$. Therefore, a divider contains a Newton's iteration module for calculating the reciprocal and a multiplier.

This module is used to calculate the reciprocal of input a, *i.e.*, a^{-1} in the squaring block. To compute a^{-1} by applying Newton's method, we choose a function $f(x) = \frac{1}{x} - a$ that

has a zero at $x = 1/a$. Newton's method is used to find an approximation to a^{-1}:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \frac{1/x_i - a}{1/x_i^2} = x_i(2 - ax_i), \quad (12)$$

where $f'(x_i)$ is the first order derivative of $f(x_i)$. Specially, a in Eq. (12) should be scaled to $[0.5, 1]$ to make the function converge faster. First, an LOD is applied to calculate the number of significant bits n_a of a. Then, $a' = a >> (1024 - n_a)$, where ">>" means the right shift operation, can be used for iterations. After several iterations, the reciprocal of a can be obtained by simply bit shifting the final result as $\frac{1}{a} = x_{i+1} >> n_a$. The approximation error decreases quadratically after each iteration and an appropriate initial estimate x_0 is needed. To reduce the absolute error of approximation to the reciprocal and be efficient for hardware implementation, we choose

$$x_0 = 3 - 2a' \ (x_0 \in [1, 2]). \quad (13)$$

In our design, a' is a 1024-bit unsigned number, where one bit is for the integer part and 1023 bits are for the decimal part. The subtraction $3 - 2a'$ can be written as:

$$3 - 2a' = (3 << 1023 + (\sim (2a') << 1023 + 1)) >> 1023, \quad (14)$$

where "<<" means the left shift operation and "$\sim$" means the not operator. The difference between $\sim 2a' << 1023$ and $\sim 2a' << 1023{+}1$ is quite small and can be ignored. We calculate $x_0 = 3{+}(\sim 2a')$ rather than $x_0 = (3 << 1023{+}(\sim 2a' << 1023 + 1)) >> 1023$ to avoid the latency caused by carry-propagation of adding "1". In addition, 1023 decimal bits of "3" are zeros, and then $3 + (\sim 2a')$ only requires inverter for not operation. Thus, the 1024-bit subtraction is reduced to a 1024-bit inversion.

Similarly, the formula $x_{i+1} = x_i(2 - a'x_i)$, which involves two multiplications and a subtraction can be written as $x_{i+1} = x_i(2{+} \sim a'x_i)$. Then, each iteration for x_{i+1} only needs two multiplications and one not operation. As a result, the hardware architecture for the iteration is shown in Fig. 5. In particular, the bits in dashed boxes are for explanation purposes only and do not exist in the actual design. The signal sel_1 is used to choose whether to calculate the initial estimate x_0 or intermediate value x_i. The signal sel_2 is used to decide when to stop the iteration. For a 1024-bit division $d = \frac{n}{a}$, 10 iterations are enough to obtain the accurate integer quotient.

4.5.3 Architecture of the Large-Number Adder

VDF evaluation contains many additions and subtractions of large numbers, where subtractions are also calculated with adders. To reduce the latency of a large-number adder, the square-root carry-select (SRCS) [38] method is adopted. For example, a 1024-bit unsigned addition can be divided into 2-bit, 3-bit, ..., 35-bit, 35-bit, 36-bit, ..., 43-bit, and 44-bit additions by using SRCS method. The architecture of the 1024-bit adder is shown in Fig. 6, where RCA is ripple carry adder. As a result, the delay of the 1024-bit adder is dramatically reduced from the delay of 1024 full adders to about 44 full adders.

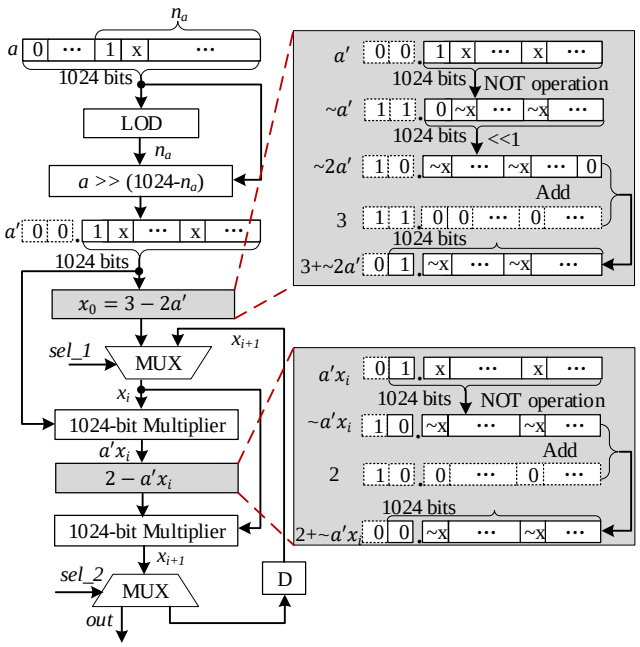


Fig. 5: Architecture for calculating the reciprocal.

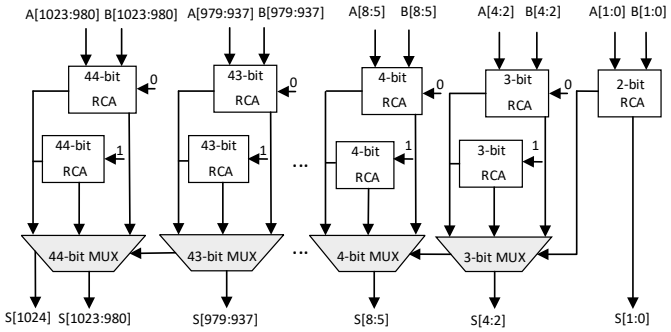


Fig. 6: Architecture of the large-number adder.

5 Experimental Results and Comparisons

In this section, both the behavioral-level simulation results and the synthesis results are given. The proposed design of the VDF evaluation in the class group of binary quadratic forms is coded in SystemVerilog, and a testbench is built to verify the correctness of the model. After that, the design is synthesized under TSMC 28-nm CMOS technology.

5.1 Behavioral-level Simulation and Hardware Implementation Results

We code the proposed design with SystemVerilog and use Vivado 2018.3 EDA platform for behavioral-level simulation. In the simulation, we randomly generate 1000 pairs $f(a, b, c)$ of 2048-bit discriminants and calculate the average number of clock cycles for the main operations.

The proposed design is synthesized under TSMC 28-nm CMOS technology, and the implementation results are shown in TABLE 3. As shown, the clock frequency of this implementation is 455 MHz, and the critical path delay is 2.2 ns. Moreover, the time for each operation can be calculated as: runtime = calculation cycles $\times$ period.

TABLE 2: Runtime Comparison of Our Implementation Results and Software Results

Operation	Runtime of software	Proportion of total time in software	Clock cycles in hardware	Proportion of total time in hardware	Runtime of our implementation	Speedup
Squaring+Reduction	25.6 μs	100%	3214	100%	7.1 μs	3.6x
Squaring	9.7 μs	38%	1573	49%	3.5 μs	2.8x
Reduction	15.9 μs	62%	1641	51%	3.6 μs	4.4x
XGCD	8.2 μs	32%	1283	40%	2.8 μs	2.9x

As shown in TABLE 2, the average clock cycles required to calculate per VDF evaluation is 3214. The squaring operation and reduction operation take up 49% and 51%, respectively. The XGCD calculation is the most time-consuming operation in the squaring, which occupies 40% of the VDF evaluation and 82% of the squaring.

TABLE 3: Hardware Implementation Results

Frequency (MHz)	Critical Path Delay (ns)	Area (mm^2)
455	2.2	10.761

5.2 Comparisons with Previous Works

We compare our design to the state-of-the-art work for VDF squaring acceleration and also compare our design with the Chia Network's C++ implementation [39] over an Intel(R) Core(TM) i9-9900X @3.50GHz CPU.

Considering that the work in [30] is only for squaring rather than the whole VDF evaluation which includes squaring and reduction, we just compare the squaring block of our implementation with it. As shown in TABLE 4, the clock period of [30] is 2 ns, and the runtime for squaring is 6.319 μs. The runtime for squaring of our implementation is : $1573 \times 2.2\ ns = 3.460\ \mu s$, which achieves a speedup of 1.8x compared to [30]. The main reason for the speedup is that we adopt the parallel XGCD algorithm instead of the multi-precision Euclidean algorithm. By replacing the XGCD algorithm, the calculation cycles of the XGCD algorithm are reduced from approximately 3000 to 1283 cycles, bringing almost 60% reduction.

TABLE 4: Comparison of This Work and the State-of-the-art Work Under TSMC 28-nm CMOS Technology

	Area (mm^2)	Clock Period (ns)	Runtime (μs)
Squaring of [30]	9.895	2	6.3
Squaring of this work	6.474	2.2	3.5
XGCD of [30]	\	2	6.0
XGCD of this work	\	2.2	2.8

To demonstrate the efficiency of our design, we also compare our implementation results of VDF evaluation with an optimized C++ implementation proposed by the Chia Network Competition [39], which takes an average of 25.6μs per squaring and reduction over an Intel(R) Core(TM) i9-9900X @3.50GHz CPU (fabricated in 14 nm). A detailed runtime comparison of our implementation results and software results is shown in TABLE 2. For squaring, we achieve a 2.8x speedup, and a 2.9x speedup for XGCD. Meanwhile, we also achieve 4.4x speedup for reduction, resulting in a 3.6x speedup for the VDF evaluation which includes squaring (38%) and reduction (62%).

6 Conclusion

In this paper, we present the first hardware architecture for VDF evaluation in the class group of binary quadratic forms. Algorithm and hardware co-optimization is performed to achieve a possible minimum latency hardware implementation. First, a fast reduce algorithm is modified and a parallel XGCD algorithm is chosen for the latency optimization target. Second, highly parallelized and pipelined architectures for large-number divisions, multiplications, and additions are devised to reduce the calculation latency further. Finally, we code and synthesize the proposed design. The implementation results show that, with these optimization methods, our design can achieve a VDF squaring speedup of approximately 2x compared to the prior class-group-based VDF squaring accelerator, and a VDF evaluation speedup of 3.6x compared to the optimal Chia Network's software implementation.

References

[1] D. Boneh and M. Naor, "Timed commitments," in *Annual international cryptology conference*. Springer, 2000, pp. 236–254.

[2] S. A. K. Thyagarajan, T. Gong, A. Bhat, A. Kate, and D. Schröder, "Opensquare: Decentralized repeated modular squaring service," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 3447–3464. [Online]. Available: https://doi.org/10.1145/3460120.3484809

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

[4] P. Schindler, A. Judmayer, M. Hittmeir, N. Stifter, and E. Weippl, "Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness," 2021.

[5] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum," *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[6] Minimal VDF Randomness Beacon, "Minimal VDF randomness beacon," https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566, Mar. 2022, accessed: 2022-3-5.

[7] J. Benet, D. Dalrymple, and N. Greco, "Proof of replication," *Protocol Labs, July*, vol. 27, p. 20, 2017.

[8] K. P. Bram Cohen, "The chia network blockchain," *https://www.chia.net/assets/ChiaGreenPaper.pdf*, 2019.

[9] V. Attias, L. Vigneri, and V. Dimitrov, "Implementation study of two verifiable delay functions," in *2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[10] Ethereum, "Ethereum 2.0," https://ethereum.org/en/eth2/, 2021, accessed: 2022-3-6.

[11] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," 1996.

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

[13] B. Wesolowski, "Efficient verifiable delay functions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 379–407.

[14] J. Buchmann and H. C. Williams, "A key-exchange system based on imaginary quadratic fields," *Journal of Cryptology*, vol. 1, no. 2, pp. 107–118, 1988.

[15] K. Belabas, T. Kleinjung, A. Sanso, and B. Wesolowski, "A note on the low order assumption in class group of an imaginary quadratic number fields," p. 1310, 2020, https://ia.cr/2020/1310.

[16] J. Buchmann and U. Vollmer, "Binary quadratic forms," in *Binary Quadratic Forms*. Springer, 2007, pp. 9–20.

[17] H. Lipmaa, "Secure accumulators from euclidean rings without trusted setup," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 224–240.

[18] S. A. K. Thyagarajan, G. Castagnos, F. Laguillaumie, and G. Malavolta, "Efficient cca timed commitments in class groups," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2663–2684.

[19] R. W. Lai and G. Malavolta, "Subvector commitments with application to succinct arguments," in *Annual International Cryptology Conference*. Springer, 2019, pp. 530–560.

[20] D. Boneh, B. Bünz, and B. Fisch, "A survey of two verifiable delay functions." *IACR Cryptology ePrint Archive*, vol. 2018, p. 712, 2018.

[21] L. De Feo, S. Masson, C. Petit, and A. Sanso, "Verifiable delay functions from supersingular isogenies and pairings," in *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2019, pp. 248–277.

[22] E. Landerreche, M. Stevens, and C. Schaffner, "Non-interactive cryptographic timestamping based on verifiable delay functions," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020, pp. 541–558.

[23] N. Ephraim, C. Freitag, I. Komargodski, and R. Pass, "Continuous verifiable delay functions," in *Advances in Cryptology – EUROCRYPT 2020*. Cham: Springer International Publishing, 2020, pp. 125–154.

[24] N. Döttling, S. Garg, G. Malavolta, and P. N. Vasudevan, "Tight verifiable delay functions," in *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2020, pp. 65–84.

[25] VDF alliance, "VDF alliance FPGA competition," https://www.vdfalliance.org/contest, Sep. 2019, accessed: 2022-3-6.

[26] E. Öztürk, "Design and implementation of a low-latency modular multiplication algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 6, pp. 1902–1911, 2020.

[27] Eric Pearson, "VDF competition on FPGA - round2," https://github.com/supranational/vdf-fpga-round2-results, Sep. 2019, accessed: 2022-3-NA.

[28] A. C. Mert, E. Öztürk, and E. Savas, "Low-latency asic algorithms of modular squaring of large integers for vdf evaluation," *IEEE Transactions on Computers*, 2020.

[29] Chia's Network, https://github.com/Chia-Network/vdf-competition, Jul. 2019, accessed: 2022-3-NA.

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

[31] S. M. Sedjelmaci, "A parallel extended gcd algorithm," *Journal of Discrete Algorithms*, vol. 6, no. 3, pp. 526–538, 2008.

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

[33] A. Dutta, "Fast reduction," https://github.com/Akashnil/chia-vdf-competition/tree/master/Entry1, 2019, accessed: 2021-6-4.

[34] L. Long, "Binary quadratic forms," *https://github.com/Chia-Network/vdf-competition/blob/master/classgroups.pdf*, 2018.

[35] T. Jebelean, "Comparing several gcd algorithms," in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. IEEE, 1993, pp. 180–185.

[36] D. H. Lehmer, "Euclid's algorithm for large numbers," *The American Mathematical Monthly*, vol. 45, no. 4, pp. 227–233, 1938.

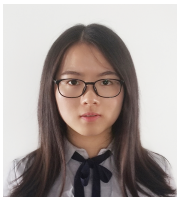
[37] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. Russian Academy of Sciences, 1962, pp. 293–294.

[38] Y. He, C.-H. Chang, and J. Gu, "An area efficient 64-bit square root carry-select adder for low power applications," in *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 2005, pp. 4082–4085.

[39] Akashnil, "VDF Competition," https://github.com/Chia-Network/vdftrack1results/tree/main/akashnil, 2019.

Danyang Zhu received the B.S. degree in communication engineering from Nanjing University, Nanjing, China. Now she is working toward her Ph.D. degree in information and communication engineering from Nanjing University, Nanjing, China.

Her research interests include very large scale integration design, specifically VLSI design for digital signal processing and cryptographic engineering.



Jing Tian received her B.S. degree in microelectronics and Ph.D. degree in information and communication engineering from Nanjing University, Nanjing, China, in 2015 and 2020, respectively. She is now an associate research fellow in Nanjing University. She has published over 20 technical papers. Her research interests include VLSI design for digital signal processing and cryptographic engineering.

Minghao Li received his B.S. degree in Integrated Circuit Design and Integrated System from Nanjing University, China, in 2021.He is currently pursuing the M.S. degree in Electronic Information at Nanjing University.

His research interests include VLSI design for digital signal processing.

Zhongfeng Wang has been working for Nanjing University, China, as a Distinguished Professor since 2016. Previously he worked for Broadcom Corporation, California, and Oregon State University. Dr. Wang is a world-recognized expert on Low-Power High-Speed VLSI Design for Signal Processing Systems. He has published over 200 technical papers with multiple best paper awards received from the IEEE technical societies, among which is the VLSI Transactions Best Paper Award of 2007. In the past, he has served as Associate Editor for IEEE Trans. on TCAS-I, TCAS-II, and TVLSI for many terms. He has also served as TPC member and various chairs for tens of international conferences. Moreover, he has contributed significantly to the industrial standards. So far, his technical proposals have been adopted by more than fifteen international networking standards. His current research interests are in the area of Optimized VLSI Design for Digital Communications and Deep Learning.