IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 1, pp. 438–462. DOI:10.46586/tches.v2023.i1.438-462

Low-Latency Design and Implementation of the Squaring in Class Groups for Verifiable Delay Function Using Redundant Representation

Danyang Zhu, Rongrong Zhang, Lun Ou, Jing Tian* and Zhongfeng Wang*

School of Electronic Science and Engineering, Nanjing University, Nanjing 210046, China
zhudanyang10@foxmail.com;{zhangrr,olun}@smail.nju.edu.cn;
{tianjing,zfwang}@nju.edu.cn
*Corresponding author

Abstract. A verifiable delay function (VDF) is a function whose evaluation requires running a prescribed number of sequential steps over a group while the result can be efficiently verified. As a kind of cryptographic primitives, VDFs have been adopted in rapidly growing applications for decentralized systems. For the security of VDFs in practical applications, it is widely agreed that the fastest implementation for the VDF evaluation, sequential squarings in a group of unknown order, should be publicly provided. To this end, we propose a possible minimum latency hardware implementation for the squaring in class groups by algorithmic and architectural level co-optimization. Firstly, low-latency architectures for large-number division, multiplication, and addition are devised using redundant representation, respectively. Secondly, we present two hardware-friendly algorithms which avoid time-consuming divisions involved in calculations related to the extended greatest common divisor (XGCD) and design the corresponding low-latency architectures. Besides, we schedule and reuse these computation modules to achieve good resource utilization by using compact instruction control. Finally, we code and synthesize the proposed design under the TSMC 28nm CMOS technology. The experimental results show that our design can achieve a speedup of 3.6x compared to the state-of-the-art implementation of the squaring in the class group. Moreover, compared to the optimal C++ implementation over an advanced CPU, our implementation is 9.1x faster.

Keywords: Verifiable delay functions · squaring · extended GCD · low-latency · ASIC · architecture · class groups · redundant representation

1 Introduction

Verifiable delay functions (VDFs), formalized by Boneh *et al.* [BBBF18], are functions that need inherently sequential computation to evaluate but the results can be verified exponentially faster. Recently, the use of VDFs has been proven effective in a wide range of exciting applications, such as generation of public verifiable randomness beacons [GLOW21, BGB17, SJH+21], computational time-stamps [LSS20], proofs of replication [BDG17, Fis19], and resource-efficient blockchains [CP19, Eth21, AVD21].

According to [BBBF18], a complete VDF is composed of the following three components:

- **Setup**$(\lambda, T) \rightarrow (pp)$: The input for this operation are λ and T, where parameter λ is related to the security level and parameter T corresponds to the delay. Moreover, T is positively correlated with the delay but unequal to the value of delay. Based on input, this operation will generate and output the public parameters, pp, that will be used in the VDF, such as the domain and range of VDF.

Received: 2022-07-15 Accepted: 2022-09-15 Published: 2022-11-29

- **Eval**$(pp, x) \to (y, \pi)$: This operation performs the inherent computation on the input x based on the public parameter and outputs the result y. Meanwhile, evaluation needs to output proof π, which are intermediate evaluation results for efficient verification.
- **Verify**$(pp, x, y, \pi) \to (accept, reject)$: This operation verifies that the result y of the evaluation is the correct output of x. Using the proof π given by the evaluator, the result can be verified efficiently without re-doing the costly computation in the evaluation.

In the VDF, the most critical and time-consuming operation is the evaluation, which requires a prescribed number of steps to calculate and cannot be parallelized. The operation of repeated squarings in a group of unknown order, defined as time-lock puzzles by Rivest, Shamir, and Wagner in [RSW96], is considered a relatively simple and efficient solution for evaluation. The two elegant VDF constructions proposed by Pietrzak [Pie18] and Wesolowski [Wes19] utilize the serial nature of repeated squaring in groups of unknown order, such as RSA groups or class groups of imaginary quadratic fields.

The repeated squaring over the RSA group is to compute: $a^{2^T} \bmod N$, where $N = pq$ (p, q are both unknown large primes). Since only a, T, and N are given, the fastest algorithm to compute this takes T squaring steps. However, this assumption is broken if the factorization of N is known. VDF construction over the RSA group requires the factors of N to be generated by a trusted setup and not revealed. In contrast to RSA groups, using class groups is a more elegant solution where the setup is transparent because the order of the class group is nearly impossible to compute [BV07, CL84]. Therefore, VDF in class groups potentially has a wide range of applications without a trusted third-party setup, such as cryptocurrencies that require technical disclosure [CP19].

All VDF applications share a common constraint: they can only guarantee a prescribed number of steps of sequential computation, but real-world execution time for VDF computation depends heavily on the speed of computing platforms. The system could be broken if a malicious participant can compute the VDF evaluation much faster than all honest users. Therefore, to secure VDF applications, the fastest implementation for VDF evaluation needs to be thoroughly studied and made public. Since VDF evaluation is inherently sequential squaring over groups, a low-latency squaring algorithm and its architecture with the minimum latency are crucial for making VDF schemes practical.

Related works. The sequential computation of VDF in the RSA group is performed through modular multiplication or modular squaring, whose implementation has been extensively studied. However, implementation results for VDF evaluation in the class group have been scarcely presented in the open literature, though the latter is more advantageous than the former because it does not need a third-party. In 2019, *Öztürk* proposed an efficient low-latency multiplication algorithm for VDF in RSA groups with a fixed modulus [Özt19]. This algorithm was adopted in the competition held by VDF Alliance [VDF19] to achieve the lowest latency in operating a number of sequential squarings on FPGA platforms. Based on this algorithm, Mert *et al.* fully analyzed and compared the existing ultra-low latency algorithms of modular squaring for ASIC implementation [MÖS20]. Besides, [San21] presented a low-latency Montgomery modular multiplication algorithm for FPGA implementation with a slight increase in latency compared to work in [Özt19]. However, it is easy to replace the modulus, and there is no need for a large number of look-up tables and pre-computations.

The sequential computation of VDF evaluation in the class group includes successive squaring and reduction of binary quadratic forms [BV07]. Their computations contain calculations of the extended greatest common divisor (XGCD), divisions, multiplications, and additions of large numbers. In 2019, Chia Network held a contest for the fast software implementation of VDF evaluation with a 2048-bit discriminant [Chi19], and optimized C++ implementations have been developed through this competition. At last, a C++ implementation using the NUDUPL algorithm [JP02] stood out and was adopted as the solution for Chia Network [CP19]. The NUDUPL algorithm performs squaring and reduces output value to assist with the later reduction. The NUDUPL algorithm takes more computation time than the conventional squaring algorithm, but the following reduction time is reduced dramatically, making the evaluation more efficient.

In 2020, Zhu *et al.* proposed an efficient hardware accelerator for squaring by partially parallelizing the squaring, and this resulted in a 2x speedup compared to the state-of-the-art (SOTA) C++ implementation over a standard CPU [ZST+20]. For XGCD, the most time-consuming computation in the VDF evaluation, Sreedhar *et al.* improved a two-bit plus-minus (PM) algorithm with carry-save adders to calculate XGCD [SHT22]. The design was implemented in ASIC, and the implementation results show that it is 8x faster than the XGCD implementation in Zhu's work [ZST+20]. Recently, complete hardware implementation for the VDF evaluation in the class group was proposed in [ZTLW22] which achieves a 3.6x speedup compared to the C++ implementation result. This design chose the standard squaring algorithm rather than the more efficient NUDUPL algorithm, and a modified parallel XGCD algorithm was used for XGCD calculation.

Contributions. Since the VDF application has only been proposed for a short time and is computationally complex, the results of the hardware implementation are still limited. The works in [ZTLW22] and [SHT22] are implementations of XGCD in VDF, and [ZST+20] did not implement the reduction. Among these previous works, only that of [ZTLW22] is a complete implementation for the squaring in the class group, but the algorithm and the hardware design in it can be greatly optimized. Compared to these work, we propose a low-latency implementation for the VDF in class groups with a 2048-bit discriminant by using an efficient NUDUPL algorithm for the first time. As mentioned earlier, the calculation of VDF evaluation is the sequential calculation of a number of squarings in the class group, so we aim to accelerate the squaring in the class group. In addition to VDF, the calculation of squaring in the class group can also be used in many other applications, such as accumulators [BBF19, Lip12], timed commitments [TCLM21], and succinct non-interactive argument of knowledge [LM19].

In this paper, we firstly review and conclude the efficient NUDUPL algorithm for squaring in the class group. We also modify the complex operations involved (such as XGCD and partial XGCD) to be more hardware friendly. Then, we provide low-latency large-number adders and large-number multipliers by utilizing redundant representation. At last, we devise the architecture for squaring in the class group using customized instruction. The main contributions are summarized as follows:

- We utilize the k-ary algorithm proposed in [Sor90] to calculate large-number XGCD only by simple additions and shifts. The goal of partial XGCD is to use the Euclidean algorithm to reduce the input to a determined size. Therefore, for the first time, we propose an efficient scheme to calculate the partial XGCD by only using comparisons, shifts, and subtractions, avoiding the difficult division involved in the Euclidean algorithm.

- We introduce a redundant representation system with a "carry-free" nature to realize arithmetic modules, and extremely low-latency large-number adders and large-number multipliers are devised, respectively. We adopt Goldschmidt algorithm [Gol64] to implement the divisor, which can reduce the number of iterations and reuse the large-number multipliers.

- We develop a compact control logic and design the instruction set for efficient scheduling, making the implementation area-efficient and flexible. At last, we propose a low-latency design for squaring using the NUDUPL algorithm by integrating the proposed compact control logic, memory, and arithmetic modules.

We code the proposed design in SystemVerilog and synthesize it under the TSMC 28nm CMOS technology. The experimental results show that our implementation takes an average of 2.0 μs for a squaring in the class group, which is the fastest implementation among the existing work.

Paper organization. Section 2 overviews binary quadratic forms, the NUDUPL algorithm, and the redundant representation. The proposed architectures for main operations in the squaring are presented in Section 3. In Section 4, the method of instruction scheduling is presented. The experimental results of the proposed design and the performance comparison are given in Section 5. The work is summarized in Section 6.

2 Background

2.1 Binary Quadratic Forms

The VDF construction used in Chia Network is the [Wes19]'s construction using squaring of binary quadratic forms over a class group. In this subsection, we briefly introduce several definitions of binary quadratic forms used in the VDF construction.
Form. An integral binary quadratic form is defined as:

$$f(x,y) = ax^2 + bxy + cy^2, \tag{1}$$

where $a, b, c \in \mathbb{Z}$ and $a, b, c \neq 0$. For the sake of simplicity, we will write only *form* instead of *integral binary quadratic form*. Moreover, *form* $f(x,y) = ax^2 + bxy + cy^2$ is written as $f = [a, b, c]$.
Discriminant. The discriminant of a form $f = [a, b, c]$ is $\Delta(f) = b^2 - 4ac$. The discriminant Δ in Chia Network's VDF is a large negative prime, e.g., 2048 bits, and $|\Delta| \equiv 3 \bmod 4$, making the order of the class group unknown.
Positive definite. A form f of positive discriminant is called indefinite and a form of negative discriminant is called positive/negative definite, according to whether f represents positive or negative integers. The forms relevant to the Chia Network VDF are positive definite forms, where $\Delta(f) < 0$ and $a > 0$.
Reduced. A primitive positive definite form $f = [a, b, c]$ is reduced if $|b| \leq a \leq c$, and when $a = c$ then $b \geq 0$. In particular, each proper equivalence class of positive definite form contains a unique reduced representative.
Squaring. The squaring is to calculate $F = f^2 = AX^2 + BXY + CY^2$, where $[A, B, C]$ can be obtained by performing specific calculations on $[a, b, c]$.

2.2 NUDUPL Algorithm

Squaring operation on the binary quadratic form f of discriminant Δ will make the resulting form $F = f^2$ larger. Since the VDF evaluation is performing repeated squaring operations, a reduction is required to turn F into a unique reduced form while avoiding increasing the form. The NUDUPL algorithm is used to compute reduced squaring of a positive definite binary quadratic form by applying reduction before squaring. Compared to the original calculation process of one squaring followed by one reduction, using the NUDUPL algorithm takes less time to compute reduced squaring, which is essential for the implementation of VDF evaluation.

We adopt the NUDUPL algorithm for low-latency implementation of squaring of binary quadratic forms, and the NUDUPL algorithm is detailed in Algorithm 1. As shown in

Algorithm 1: The NUDUPL algorithm

```
   Input: $(a, b, c)$, where $a, b, c \in \mathbb{Z}$
   Output: $(A, B, C)$, where $A, B, C \in \mathbb{Z}$
1  Precompute $L \leftarrow |\Delta|^{\frac{1}{4}}$
2  $(x, y, G) \leftarrow$ XGCD($a$,$b$)                    ▷ satisfy $ax + by = G$
3  Set $A_x \leftarrow G$,$B_y \leftarrow a/G$, $D_y \leftarrow b/G$        ▷ $G$ always equals 1 in Chia's VDF
4  $B_x \leftarrow (cy) \bmod B_y$
5  Set $b_x \leftarrow B_x$, $b_y \leftarrow B_y$
6  if $|b_y| \leq L$ then
7      $d_x \leftarrow (b_x D_y - c)/B_y$
8      $A \leftarrow b_y^2$, $C \leftarrow b_x^2$
9      $B \leftarrow b - (b_x + b_y)^2 + A + C$
10     $C \leftarrow C - Gd_x$
11 else
12     Set $x \leftarrow 1$, $y \leftarrow 0$, $z \leftarrow 0$
13     while $|b_y| > L$ and $b_x \neq 0$ do
14         $q \leftarrow \lfloor \frac{b_y}{b_x} \rfloor$, $t \leftarrow b_y \bmod b_x$     ▷ This block is the partial XGCD operation
15         $b_y \leftarrow b_x$, $b_x \leftarrow t$
16         $t \leftarrow y - qx$, $y \leftarrow x$,$x \leftarrow t$
17         $z \leftarrow z + 1$
18     end
19     if $z$ is odd then
20         $b_y \leftarrow -b_y$, $y \leftarrow -y$
21     end
22     $a_x \leftarrow Gx$, $a_y \leftarrow Gy$
23     $d_x \leftarrow (b_x D_y - cx)/B_y$
24     $Q_1 \leftarrow d_x y$, $d_y \leftarrow Q_1 + D_y$
25     $B \leftarrow G(d_y + Q_1)$
26     $d_y \leftarrow d_y/x$
27     $A \leftarrow b_y^2$, $C \leftarrow b_x^2$
28     $B \leftarrow B - (b_x + b_y)^2 + A + C$
29     $A \leftarrow A - a_y d_y$, $C \leftarrow C - a_x d_x$
30 end
```

Algorithm 1, the relatively expensive operations in the NUDUPL algorithm include XGCD (step 2), modular multiplication (step 4), division (step 7, step 23, and step 26), and partial XGCD (steps 13 to 21). We implement these operations as follows:

- *XGCD*: An efficient two-bit PM algorithm is adopted, and the corresponding low-latency architecture using redundant signed digit (RSD) representation is designed.
- *Modular multiplication*: Since the modulus B_y is not fixed and B_y is also a divisor in other divisions, we first calculate the reciprocal of B_y and then apply the multiplication.
- *Division*: We use Goldschmidt algorithm [Gol64] to calculate division, which has a fast convergence rate especially for large numbers.
- *Partial XGCD*: The partial XGCD uses the Euclidean algorithm to reduce the input to a given size based on a specific rule. The large-number division (step 14) in partial XGCD is highly time-consuming and must be performed many times, which is unacceptable for low-latency design. We propose an efficient scheme to calculate

partial XGCD only by analyzing the relationship between outputs by comparisons, shifts, and subtractions.

In the following section, we will discuss the above algorithms and their hardware architecture in detail. In addition to the above calculations, the NUDUPL algorithm also involves many sequential multiplications, additions, and subtractions. We implement these operations by instantiating a few adders and multipliers and designing an appropriate instruction set.

2.3 Redundant Representation

Among the operations involved in the VDF evaluation, additions/subtractions are the most common and basic operations. Since the operands are very large, using carry propagation adders (CPAs) will result in a very long carry chain. Also, the VDF evaluation is performing T squarings continuously and outputs the final result, where T usually is a large number (10^6 or larger). Therefore, a redundant representation system works well for the low-latency implementation of the VDF evaluation. It avoids carry propagation and requires only one redundant representation to two's complement representation conversion at final output.

In a regular base 2 representation system, an L-bit number can represent 2^L different numbers where each number has unique representations, called non-redundant representation. In contrast, redundant representation represents the 2^L number by more bits where a number can have two or more representations. The most common redundant representations are Carry-Save (C-S) and RSD representations.
In C-S representation, a signed number A can be expressed as the sum of two signed numbers A_c and A_s:

$$A = A_c + A_s = \sum\nolimits_{i=0}^{k-1} (a_c + a_s)2^i \tag{2}$$

In RSD representation, a signed number A can be expressed as the difference between two unsigned numbers A^+ and A^-:

$$A = A^+ - A^- = \sum\nolimits_{i=0}^{k-1} a_i 2^i = \sum\nolimits_{i=0}^{k-1} (a_i^+ - a_i^-)2^i, \tag{3}$$

where $a_i^+, a_i^- \in \{0, 1\}$ and $a_i \in \{\bar{1}, 0, 1\}$. In RSD with radix-2, digits a_i are represented by 0, 1, and -1, where digit 0 is coded with 00 or 11, digit 1 is coded with 10, and digit -1 (written as $\bar{1}$) is coded with 01. Both C-S representation and RSD representation are effective in avoiding the long delays caused by the long carry chains in additions/subtractions. However, we choose the RSD representation for our implementation because of the following reasons:

- Even if the operands are signed, the RSD representation only needs unsigned numbers without additional sign bits, which is a great advantage in the VDF design since it involves signed operands.
- Compared to C-S representation, the "0" in RSD representation is determined [Avi61], and this feature makes it simple to determine whether a number equals zero.
- The subtraction is straightforward because the opposite operation in RSD representation is just swapping positions of a_i^+ and a_i^-.

3 Proposed Architectures for the Main Operations

3.1 RSD Multiplier

A bitwise integer multiplication by digital circuits can be summarized as three steps: (1) Generating partial products; (2) Adding all partial products by a partial product reduction tree until only two partial product rows remain; (3) Adding the two remaining rows by a fast CPA. Generating partial products is usually relatively simple, and we can use redundant binary numbers to perform the second step. When the operands are large, the delay caused by the long carry chain of CPA in the third step will be high. Fortunately, we use the redundant form for the entire VDF evaluation, thus avoiding step 3 in our multiplications.

As shown in Figure 1, the RSD multiplier consists of RSD partial product generators (PPGs) and RSD adders (RSDAs). The two inputs of the RSD multiplier are $A = \sum_{i=0}^{k-1} a_i 2^i$ and $B = \sum_{i=0}^{k-1} b_i 2^i$, where $a_i, b_i \in \{\bar{1}, 0, 1\}$. In our design, there is no need for Booth encoding, as the input is already in RSD representation. We consider A as the multiplier and B as the multiplicand, and the partial product is generated according to Table 1. The $\bar{B}$ is obtained by just swapping the bits in odd and even positions of B: $\bar{B} = \sum_{i=0}^{k-1} (b_i^- - b_i^+) 2^i$.

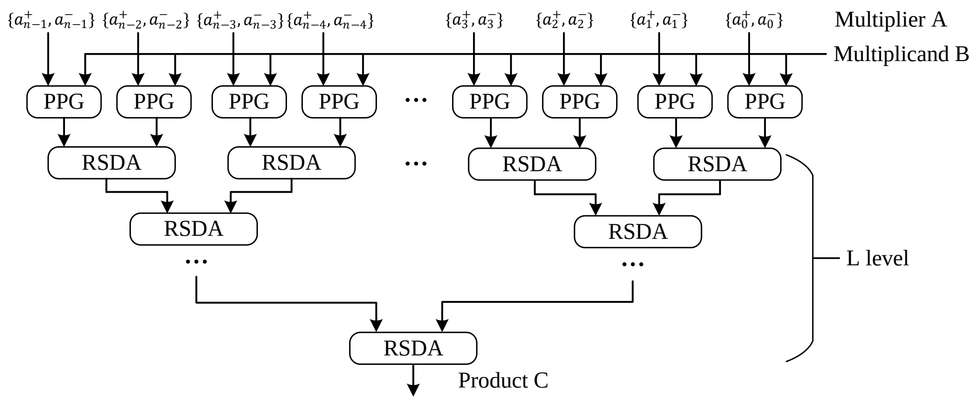

Figure 1: The block diagram of a RSD multiplier.

Table 1: Partial product generation of the RSD representation.

a_i	a_i^+	a_i^-	partial product
0	0	0	0
0	1	1	0
1	1	0	B
-1	0	1	$\bar{B}$

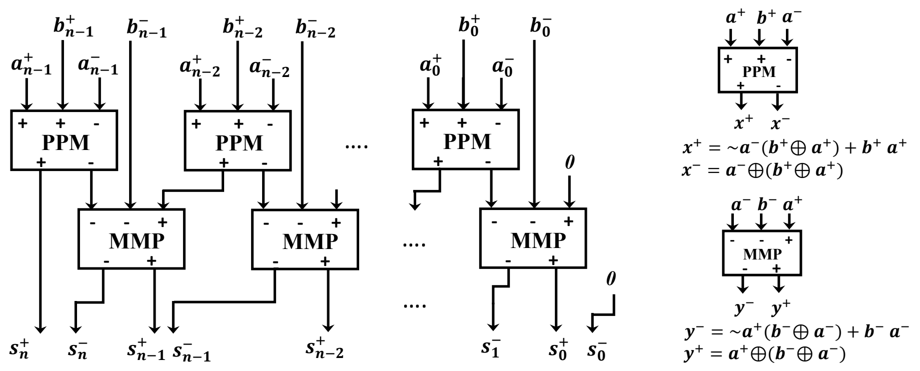
b_{n-1}^+ b_{n-1}^- b_{n-2}^+ b_{n-2}^- b_0^+ b_0^-
a_{n-1}^+ a_{n-1}^- a_{n-2}^+ a_{n-2}^- a_0^+ a_0^-
PPM
MMP
0
s_n^+ s_n^- s_{n-1}^+ s_{n-1}^- s_{n-2}^+ s_1^- s_0^+ s_0^-
a^+ b^+ a^-
x^+ x^-
$x^+ = \sim a^-(b^+ \oplus a^+) + b^+ a^+$
$x^- = a^- \oplus (b^+ \oplus a^+)$
a^- b^- a^+
y^- y^+
$y^- = \sim a^+(b^- \oplus a^-) + b^- a^-$
$y^+ = a^+ \oplus (b^- \oplus a^-)$

Figure 2: The architecture of a 2n-bit RSD adder.

After obtaining the partial product, RSD adders are needed to add the partial product. The architecture of a n-digit RSD adder is shown in Figure 2. The critical path of an RSD adder is the delay of a minus-minus-plus (MMP) adder and a plus-plus-minus (PPM) adder, regardless of the digit size of the input. In addition, as shown in Figure 2, the digit size of output becomes n+1. According to block diagram of the RSD multiplier in Figure 1, a $n \times n$-digit multiplication requires L level additions, where $L = \lceil \log_2 n \rceil$. If RSD adders are used directly for the RSD multiplier, then the digit size of the output of the $n \times n$-digit multiplier will become $n + L$. This digit size increase is unacceptable considering that the VDF evaluation is repeated squaring. We need to process the overflow digits of the multiplier to ensure the digit size of each squaring is fixed.

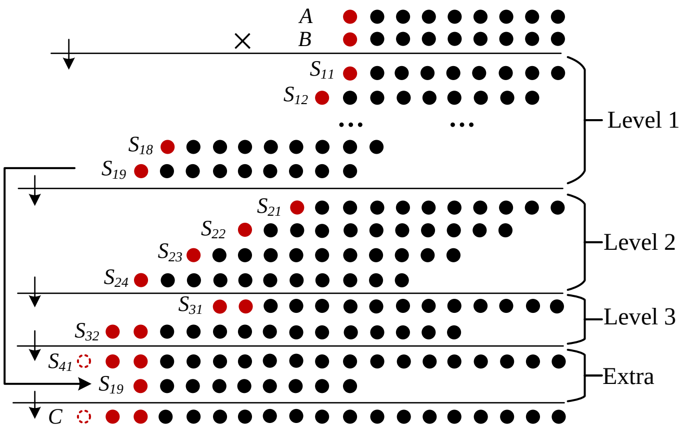

Figure 3: The process of a 8 × 8-digit RSD multiplication.

We solve the above digit width increase problem by extending the bit width of input. For a $n \times n$-digit RSD multiplier, $n+1$ digits represent the input and $2n+2$ digits represent the product. We show a 8 × 8-digit RSD multiplication as an example to illustrate our processing method. Figure 3 shows the process of a 8 × 8-digit RSD multiplication, and one dot represents on digit: $a_i = \{a_i^+, a_i^-\}$. For the 8 × 8-digit multiplication in Figure 3, we use 9 digits to represent the inputs, and the red dots represent the extended digit.

First, partial products S_{11} to S_{19} are calculated. At level 1 addition:

$$\begin{aligned} &S_{11} \in [-2^8 + 1, 2^8 - 1], S_{12} \in [-2^9 + 2, 2^9 - 2], \\ &S_{21} = S_{11} + S_{12} \in [-3 \times 2^8 + 3, 3 \times 2^8 - 3]. \end{aligned} \tag{4}$$

As a result, 10 digits are sufficient to represent the S_{21}, but 11 digits are used. The case of S_{22}, S_{23}, and S_{24} are also the same as S_{21}. Then, at level 2 addition:

$$\begin{aligned} &S_{21} \in [-3 \times 2^8 + 3, 3 \times 2^8 - 3], S_{22} \in [-3 \times 2^{10} + 12, 3 \times 2^{10} - 12], \\ &S_{31} = S_{21} + S_{22} \in [-3825, 3825]. \end{aligned} \tag{5}$$

Although 12 digits are sufficient to represent S_{31}, it is actually 14 digits for S_{31} when calculated by an RSD adder. Since two extended digits are used to represent S_{31} and S_{32}, the value represented by the first two digits can only be 00, 01, $0\bar{1}$, and $1\bar{1}$. Next, S_{41} is obtained by level 3 addition: $S_{41} = S_{31} + S_{32}$. As shown in Figure 2, the first digit of S_{41} are determined by the first four bits of an addend: $\{a_{n-1}^+, a_{n-1}^-, a_{n-2}^+, a_{n-2}^-\}$ and the first three bits of another: $\{b_{n-1}^+, b_{n-1}^-, b_{n-2}^+\}$. We suppose the S_{32} is the addend A and the S_{31} is addend B, and the possible values of the first two bits of S_{41} are shown in Table 2(a). As shown in Table 2(a), the only possible bits for the first two bits of S_{41} are 00 and 11, 0 in digit format, so this overflow bits can be directly truncated. Moreover, for the multiplication of large digit width, the overflow digits can be processed in this way at level k $(k \geq 3)$ additions.

Since we introduce an extended digit for addends of the RSD multiplication, an extra addition is needed to add the partial product S_{19}. Similar to the above analysis, for addition: $S_{41} + S_{19}$, the possible first digit of product C are shown in Table 2(a) and Table 2(b), so the overflow bits can also be truncated directly.

Table 2: The possible bits for s_n^+, s_n^-.

(a) $\{b_{n-1}^+, b_{n-1}^-, b_{n-2}^+\} = 000$

Value	$a_{n-1}^+, a_{n-1}^-, a_{n-2}^+, a_{n-2}^-$	s_n^+, s_n^-
00	0000	00
	0011	00
	1100	00
	1111	00
01	0010	00
	1110	00
$0\bar{1}$	0001	00
	1101	00
$1\bar{1}$	1001	11

(b) $\{b_{n-1}^+, b_{n-1}^-, b_{n-2}^+\} = 001$

Value	$a_{n-1}^+, a_{n-1}^-, a_{n-2}^+, a_{n-2}^-$	s_n^+, s_n^-
00	0000	00
	0011	00
	1100	00
	1111	00
01	0010	00
	1110	00
$0\bar{1}$	0001	00
	1101	00
$1\bar{1}$	1001	11

In summary, we ensure that the digit size of operands remains fixed during each squaring by extending the digit size of the RSD multiplier. The architecture of the proposed RSD multiplier is shown in Figure 4, where the digit size of the input of a $n \times n$-digit multiplication is $n+1$, and the digit size of output is $2n+2$. This RSD multiplier needs $L+1$ level RSDAs, where $L = \lceil \log_2 n \rceil$. The critical path delay can be easily reduced by adding pipelines between RSDAs.

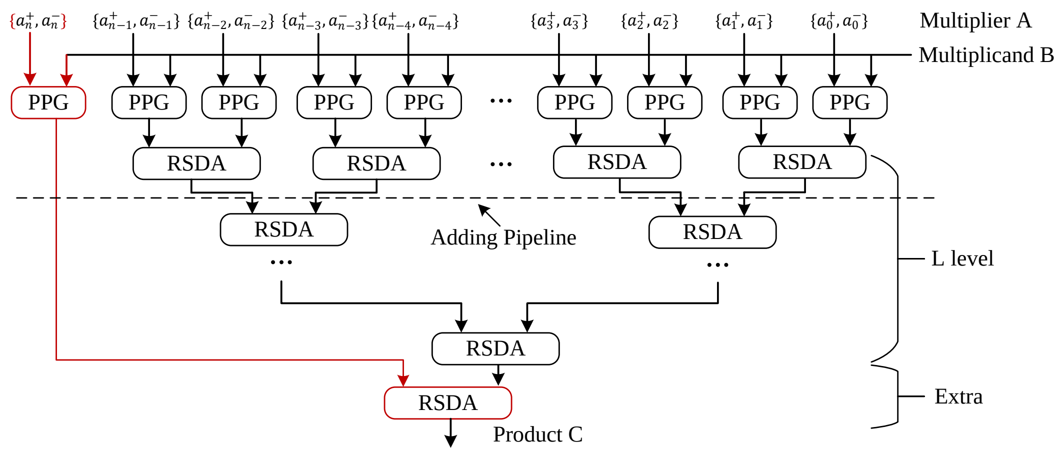

Figure 4: The architecture of the proposed RSD multiplier.

3.2 RSD Divisor

Division of large numbers are very complex operations in hardware implementation. For additions and multiplications with fixed bit-width input, the bit-width of their output is determinable, but division do not have this property. In practice, the precision of the quotient needs to be determined first and then the division is performed according to the required precision, so the number of iterations of the division depends on the set precision rather than the bit-width of the operands.

Division algorithms can be roughly divided into two categories: slow division and fast division. Slow division include restoring, non-performing restoring, non-restoring, and SRT division [HOH97]. This class of division algorithm are derived from the hand division process, in which 1 bit of quotient is determined by the remainder in each iteration so that the number of iterations and quotient's bit-width are linearly related. For large-number divisions, it is slow to converge to the quotient by using these algorithms.

The leading fast division algorithms are the Newton-Raphson algorithm and the Goldschmidt algorithm [Gol64]. Such algorithms use functions to achieve quotient with quadratic convergence and combine with fast multiplication to achieve a low-latency design for large-number division. The operations of the Newton-Raphson algorithm and Goldschmidt algorithm require several iterations, and each iteration requires two multiplications and some additions and subtractions. But the difference is that the two multiplications in the iteration of the Newton-Raphson algorithm have dependencies and cannot be computed in parallel. However, the multiplications in the Goldschmidt algorithm can be paralleled. By parallelizing two multiplications, the computation time for division can be effectively reduced, which is why we choose the Goldschmidt algorithm to implement the low-latency divider.

For a division: $Q = N/D$, the Goldschmidt algorithm repeatedly multiplies the numerator and the denominator by a common factor F_i, which is chosen to make the numerator converge to 1. This results in the dividend convergent to quotient Q:

$$Q = \frac{N}{D} \times \frac{F_1}{F_1}\frac{F_2}{F_2}\frac{F_{\dots}}{F_{\dots}}. \tag{6}$$

A simple Goldschmidt method is to utilize the binomial theorem. First, scale the N/D by a power of two such that $D \in (\frac{1}{2}, 1]$. Then, we choose $D_0 = 1 + x$ and $F_i = 1 + x^{2^i}$. Thus,

$$\begin{aligned} \frac{N}{D} &= \frac{N}{1+x} = \frac{N \cdot (1-x)}{1-x^2} = \frac{N \cdot (1-x) \cdot \left(1+x^2\right)}{1-x^4} = \cdots = Q' \\ &= \frac{N' = N \cdot (1-x) \cdot \left(1+x^2\right) \cdots \left(1+x^{2^{(k-1)}}\right)}{D' = 1 - x^{2^k} \approx 1}. \end{aligned} \tag{7}$$

After k iterations, the denominator $1 - x^{2^n}$ approaches 1, the numerator is near Q, and Q is the exact integer quotient when k is large enough. Considering that most of the divisors involved in the NUDUPL algorithm are the same, making $N = 1$ and computing the value of $1/D$ simplifies the computation. The calculation process is shown in Algorithm 2.

Algorithm 2: The Goldschmidt algorithm

```
  Input: $D \in (0.5, 1]$
  Output: $1/D \in [1, 2)$
1 Initialize: $D_0 \leftarrow D, F_0 \leftarrow 2 - D, N_0 \leftarrow 1$
2 for $i = 0$ to $k$ do
3     $D_{i+1} \leftarrow D_i * F_i$
4     $N_{i+1} \leftarrow N_i * F_i$          ▷ Compute two multiplications in parallel
5     $F_{i+1} \leftarrow 2 - D_i$
6 end
7 $1/D \leftarrow N_{k+1}$
```

As shown in Algorithm 1, the divisions in NUDUPL algorithm are step 4, step 7, step 23, and step 26 in Algorithm 1, where the bit-width of divisor B_y in binary form is 1024 and the bit-width of divisor x in binary form is 512. Division can be written as the product of the dividend and the reciprocal of the divisor. The architecture for computation of reciprocal of B_y can be used for computation of reciprocal of x, so we present the architecture for computation of reciprocal of a 1024-bit number in this section.

Architecture. As shown in Figure 5, this is the architecture for computing the reciprocal of a 1024-digit number D by using the Goldschimdt algorithm, where digit can be $\{\bar{1}, 0, 1\}$. First, the input D should be scaled to $(0.5, 1]$ to satisfy the convergence requirement, so a preprocessing module is needed. The preprocessing module mainly contains a leading one detector (LOD) to calculate the number of significant digits n_D of D, and obtain $D_0 = D << (1024 - n_D)$ where the highest digit of D_0 is 1. The design of our LOD adopts the efficient method in [Kor09]. Moreover, to utilize the RSD multiplier mentioned earlier, as shown in Figure 5, the highest digit of D_0 is extended by two digits and becomes a 1026-digit number. The architecture is used to calculate the reciprocal of D, so the other input N_0 is 1. After D_0 and N_0 are input, iterations are performed.

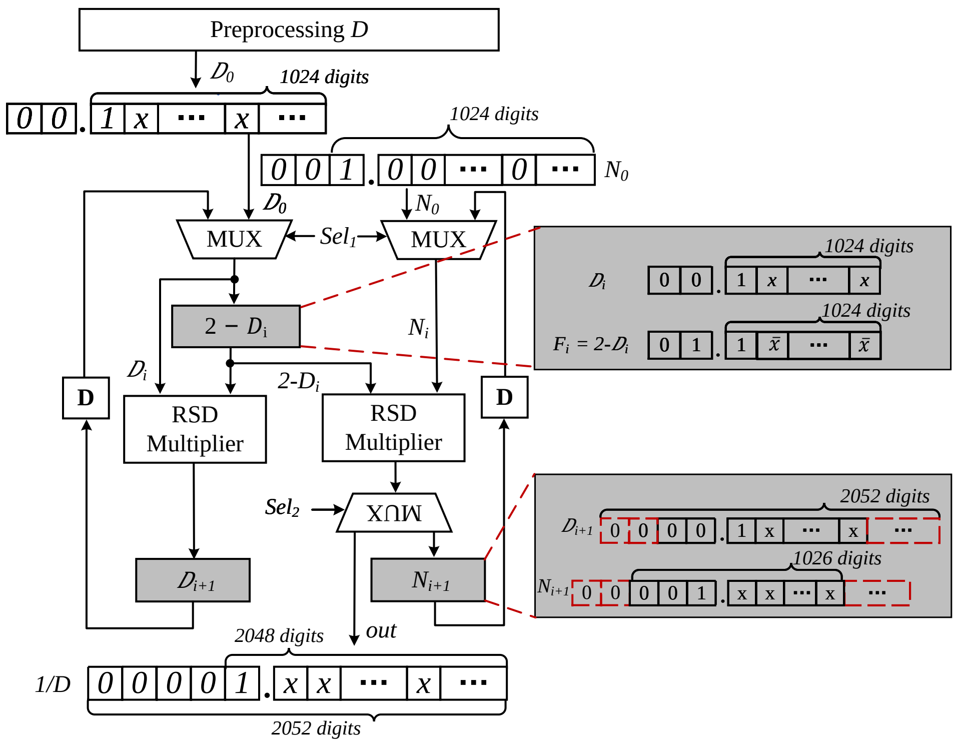

Figure 5: The architecture for calculating the reciprocal using Goldschmidt algorithm.

During the iteration, $F_i = 2 - D_i$ is first computed by simple logical operations. Then, $D_{i+1} = D_i * F_i$ and $N_{i+1} = N_i * F_i$ are computed by RSD multipliers in parallel. However, the digit size of the output of the RSD multiplier is 2052, and the output needs to be truncated to 1026 digits to perform the iterations. As shown in Figure 5, the last 1024 digits of D_{i+1} and N_{i+1} can be truncated directly, but the first two digits cannot be removed in RSD representation. Since the first two digits are actually 0, we can remove the first two digits by converting the first three digits to one according to the rule shown in Table 3.

Table 3: The rule of truncating the highest two digits.

$(n+2)$-th digit	$(n+1)$-th digit	n-th digit	The new n-th digit
1	$\bar{1}$	$\bar{1}$	1
0	1	$\bar{1}$	1
0	0	1	1
$\bar{1}$	1	1	$\bar{1}$
0	$\bar{1}$	1	$\bar{1}$
0	0	$\bar{1}$	$\bar{1}$
0	0	0	0

In fact, the new n-th digit: $\{n^+_{new}, n^-_{new}\}$ is only determined by the n-th digit: $\{n^+, n^-\}$ and $(n+1)$-th digit: $\{n^+_1, n^-_1\}$ and the rule to calculate the new n-th **bit** is as follows:

$$\begin{aligned} n^+_{new} &= |n^+_1|n^-_1 n^+|n^- + |n^+_1 n^-_1|n^+ n^- + n^+_1 n^-_1 n^+|n^- + n^+_1|n^-_1|n^+ n^- \\ n^-_{new} &= |n^+_1|n^-_1|n^+ n^- + |n^+_1 n^-_1 n^+|n^- + n^+_1 n^-_1|n^+ n^- + n^+_1|n^-_1 n^+|n^- \end{aligned} \tag{8}$$

After processing the highest digits, N_{i+1} and D_{i+1} can be input to RSD multipliers for iteration. For a 1024-digit division, 12 iterations are enough to obtain the accurate integer quotient. The output N_{k+1} of the last iteration is the reciprocal of D we need.

3.3 Architecture for the XGCD Algorithm

The XGCD calculation problem is to solve $AX + BY = GCD(A, B)$ for two given integers A and B, where (X, Y) are also known as B*é*zout coefficients. The most popular XGCD algorithms include the extended Euclidean algorithm (EEA), the plus-minus (PM) algorithm, and the binary algorithm [Jeb93]. Among these algorithms, the EEA has the least number of iterations, making it popular for software implementation. However, the original EEA includes an extremely time-consuming division of large numbers, which makes it hard to implement in hardware. Compared to the EEA, the PM and binary algorithms require more iterations, but only additions, shifts, and comparisons are performed in each iteration. Considering that the total time for the calculation of XGCD is equal to the number of iterations multiplied by the time, a two-bit PM algorithm proposed in [YZ86] can effectively achieve the low-latency hardware design.

We adopt the two-bit PM algorithm and design the corresponding hardware architecture for XGCD calculation in the NUDUPL algorithm. The two-bit PM algorithm is shown in Algorithm 3. First, initialize a and b to the input A and B, respectively, and also initialize (x, y, z, w) to $(1, 0, 0, 1)$ for calculating the B*é*zout coefficients. In particular, the algorithm introduces a small auxiliary number δ to assist in determining the computation case, and the comparison of a and b can be avoided. The sign of δ and whether a and b can be divided by 2 or 4 determines the case of the computation. Then, a and b are updated by addition/subtraction and shifts. Meanwhile, (x, y, z, w) is calculated by function *CalBézout* which is shown in Algorithm 3. By performing these calculations, values of a and b keep decreasing, and when one of them becomes 0, the iteration stops. After the iteration, simple post-processing is performed and outputs the final value (G, X, Y).

According to Algorithm 3, the proposed architecture for the two-bit PM algorithm is shown in Figure 6. The components used to update a and b mainly include RSD adders, multiplexers (MUXs), and shifters, which are easy to implement in low-latency hardware. The module that computes B*é*zout coefficients is used to perform the recursive function *CalBézout*. We reduce the latency of this module by computing all possible results in advance, and then the result is selected by the last two digits of x and B. As shown in Algorithm 3, at the end of each iteration, it is necessary to determine whether a or b is equal to 0 in this iteration. Compared to the C-S representation, the advantage of using

Algorithm 3: The two-bit PM algorithm

```
Input: $(A, B)$, where $A, B \in \mathbb{Z}$
Output: $(G, X, Y)$, where $AX + BY = G$
$a \leftarrow A$, $b \leftarrow B$
$x \leftarrow 1$, $y \leftarrow 0$, $z \leftarrow 0$, $w \leftarrow 1$, $\delta \leftarrow 0$          ▷ Initialization
while $a \neq 0$ and $b \neq 0$ do
    if $mod(a, 4) == 0$ then
        $a \leftarrow a/4$, $\delta \leftarrow \delta - 2$, $(x, y) \leftarrow$ CalBézout($x$,$y$,$A$,$B$,$2$)
    else if $mod(a, 2) == 0$ and $mod(a, 4) \neq 0$ then
        $a \leftarrow a/2$, $\delta \leftarrow \delta - 1$, $(x, y) \leftarrow$ CalBézout($x$,$y$,$A$,$B$,$1$)
    else if $mod(b, 4) == 0$ then
        $b \leftarrow b/4$, $\delta \leftarrow \delta + 2$, $(z, w) \leftarrow$ CalBézout($z$,$w$,$A$,$B$,$2$)
    else if $mod(b, 2) == 0$ and $mod(b, 4) \neq 0$ then
        $b \leftarrow b/2$, $\delta \leftarrow \delta + 1$, $(z, w) \leftarrow$ CalBézout($z$,$w$,$A$,$B$,$1$)
    else if $mod(a + b, 4) == 0$ then
        if $\delta \geq 0$ then
            $a \leftarrow (a + b)/4$, $\delta \leftarrow \delta - 1$, $(x, y) \leftarrow$ CalBézout($x$+$z$,$y$+$w$,$A$,$B$,$2$)
        else
            $b \leftarrow (a + b)/4$, $\delta \leftarrow \delta + 1$, $(z, w) \leftarrow$ CalBézout($x$+$z$,$y$+$w$,$A$,$B$,$2$)
        end
    else
        if $\delta \geq 0$ then
            $a \leftarrow (a - b)/4$, $\delta \leftarrow \delta - 1$, $(x, y) \leftarrow$ CalBézout($x$-$z$,$y$-$w$,$A$,$B$,$2$)
        else
            $b \leftarrow (a - b)/4$, $\delta \leftarrow \delta + 1$, $(z, w) \leftarrow$ CalBézout($x$-$z$,$y$-$w$,$A$,$B$,$2$)
        end
    end
    if $a == 0$ then
        $G \leftarrow b$, $X \leftarrow z$, $Y \leftarrow w$
    else
        $G \leftarrow a$, $X \leftarrow x$, $Y \leftarrow y$
    end
    if $G < 0$ then
        $G \leftarrow -G$, $X \leftarrow -X$, $Y \leftarrow -Y$
    end
end
Function CalBézout($x, y, A, B, l$):
    if $mod(x, 2) == 0$ and $mod(y, 2) == 0$ then
        $x \leftarrow x/2$, $y \leftarrow y/2$
    else
        $x \leftarrow (x + B)/2$, $y \leftarrow (y - A)/2$
    end
    if $l == 1$ then
        return $x, y$
    else
        CalBézout($x$,$y$,$A$,$B$,$l$-$1$)
    end
End Function
```

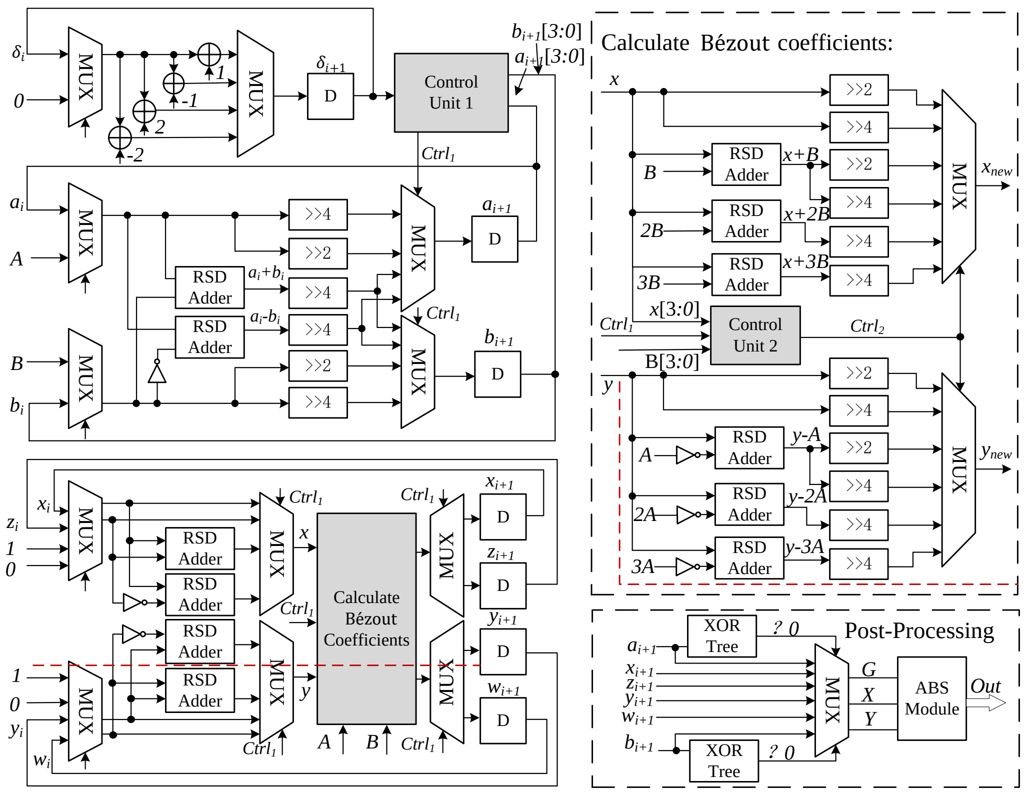

Figure 6: The architecture for the two-bit PM algorithm.

RSD representation is that this judgment is very simple to implement in hardware. In RSD representation, an XOR tree can be implemented to determine whether the input is equal to "0", and for an n-digit input, the delay of the XOR tree is the delay of $log_2 n + 1$ XOR gates.

The post-processing module needs a module to compute the absolute value (ABS), and this module can be implemented in various ways. The implementation method is unimportant because it is not in the iteration, so adding several pipelines can reduce the critical path. The critical path of the architecture for the two-bit PM algorithm is shown as the red dashed line in Figure 6, which includes the delay of four MUXs, two RSD adders, and a shifter. This critical path is short in hardware implementation, so in our design for the two-bit PM algorithm, one iteration takes only one cycle.

3.4 Architecture for the Partial XGCD Algorithm

The NUDUPL algorithm applies reduction before squaring the form so the reduced squaring form can be obtained faster, and the partial XGCD algorithm is mainly used to achieve this purpose. By performing this algorithm, the intermediate operands can be reduced from size $O(\Delta)$ to $O(\Delta^{1/2})$ in most cases and $O(\Delta^{3/4})$ in the worst case. As shown in steps 13 to 20 in Algorithm 1, the original partial XGCD using the Euclidean algorithm contains the extremely time-consuming division of large numbers. According to [vdP03], the partial XGCD algorithm makes the input b_x and b_y decrease in each iteration until one of them is smaller than another input L ($L = \Delta^{1/4}$). During the iteration, the following relation needs to be satisfied:

$$\begin{vmatrix} x & b_x \\ y & b_y \end{vmatrix} = \begin{vmatrix} x' & b'_x \\ y' & b'_y \end{vmatrix}, \tag{9}$$

where the $(x'b'_y - y'b'_x)$ remains constant in each iteration. Using the XGCD architecture detailed in Section 3.3 for the partial XGCD is the best choice, but unfortunately the iteration in two-bit PM does not satisfy Equation (9). The computation in each iteration can be seen as a matrix multiplication as Equation (10), and the auxiliary matrix satisfies $ad - bc = 1$. Besides, the choice of matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ needs to satisfy two more conditions: (1) The coefficients (a, b, c, d) should be hardware-friendly. (2) The coefficients (a, b, c, d) should ensure that b_x and b_y decrease in most iterations.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x & b_x \\ y & b_y \end{pmatrix} = \begin{pmatrix} x' & b'_x \\ y' & b_y \end{pmatrix}. \tag{10}$$

Assuming $b_y > b_x > 0$, we select a class of simple coefficients that satisfy the above conditions:

$$\begin{pmatrix} 1 & 0 \\ -2^k & 1 \end{pmatrix} \begin{pmatrix} x & b_x \\ y & b_y \end{pmatrix} = \begin{pmatrix} x & b_x \\ y - 2^k x & b_y - 2^k b_x \end{pmatrix}, k \in \mathbb{N} \tag{11}$$

As shown in Equation (11), b_y is decreased, and the calculation only involves subtraction and shifts. The coefficient k needs to satisfy b_y decreases as much as possible , but $b_y - 2^k b_x$ is still greater than 0. The best value of k will be different for different b_x and b_y, so we set multiple k and select the appropriate k by judging the relationship of b_x and b_y. If too many k are used, the hardware consumption and latency will increase.

Algorithm 4: The proposed partial XGCD algorithm

```
   Input: $b_x, b_y, L$, where $b_x > L, b_y > L$
   Output: $x, y, b_x, b_y$
1  $x \leftarrow 1$, $y \leftarrow 0$                                  ▷ Initialization
2  while $b_x \geq L$ and $b_y \geq L$ do
3      if $b_x > 2^{k_2} b_y$ then
4          $x \leftarrow x - 2^{k_2} y$, $y \leftarrow y$, $b_x \leftarrow b_x - 2^{k_2} b_y$, $b_y \leftarrow b_y$; continue
5      else if $b_x > 2^{k_1} b_y$ then
6          $x \leftarrow x - 2^{k_1} y$, $y \leftarrow y$, $b_x \leftarrow b_x - 2^{k_1} b_y$, $b_y \leftarrow b_y$; continue
7      else if $b_x > b_y$ then
8          $x \leftarrow x - y$, $y \leftarrow y$, $b_x \leftarrow b_x - b_y$, $b_y \leftarrow b_y$; continue
9      else if $b_y > 2^{k_2} b_x$ then
10         $y \leftarrow y - 2^{k_2} x$, $x \leftarrow x$, $b_x \leftarrow b_y - 2^{k_2} b_x$, $b_y \leftarrow b_y$; continue
11     else if $b_y > 2^{k_1} b_x$ then
12         $y \leftarrow y - 2^{k_1} x$, $x \leftarrow x$, $b_x \leftarrow b_y - 2^{k_1} b_x$, $b_y \leftarrow b_y$; continue
13     else
14         $y \leftarrow y - x$, $x \leftarrow x$, $b_y \leftarrow b_y - b_x$, $b_x \leftarrow b_x$; continue
15     end
16 end
```

Based on our experiments, we recommend to use two k: k_1 and k_2 $(k_2 > k_1)$, and the proposed partial XGCD algorithm is shown in Algorithm 4. When b_x and b_y are larger than L, iteration is performed, and each iteration mainly consists of shifts, subtraction, and comparison. For different coefficients, k_1 and k_2, the computation time for each iteration is almost the same, and the number of iterations mainly determines the total computation of partial XGCD.

We choose the parameter k_1 and k_2 by using a Monte Carlo simulation. We randomly generate 2^{20} pairs of 1024-bit numbers as inputs for the partial XGCD algorithm and count the average number of the iterations. The relationship between the parameter k_1 and k_2 and the number of iterations are shown in Figure 7, where $k_1 = 1, 2, 3$ and k_2 ranges from 2 to 9 with an interval of 1. As shown in Figure 7, when $k_1 = 2$ and $k_2 = 6$, the number of the iterations is minimal, and we choose this set of parameters in our design.

According to Algorithm 4, the architecture of the partial XGCD algorithm is shown in Figure 8, which mainly contains several MUXs, shifters, modules for updating (b_x, b_y) and updating (x, y), and a control module. The module for updating (b_x, b_y) includes parallel RSD adders, parallel inverters for RSD numbers, and a MUX, shown in the right dashed block in Figure 8. The module for updating (x, y) is similar to the module for updating (b_x, b_y). Besides, the control unit mainly consists of compare modules, and the compare module is also used determining whether b_x or b_y is smaller than L.

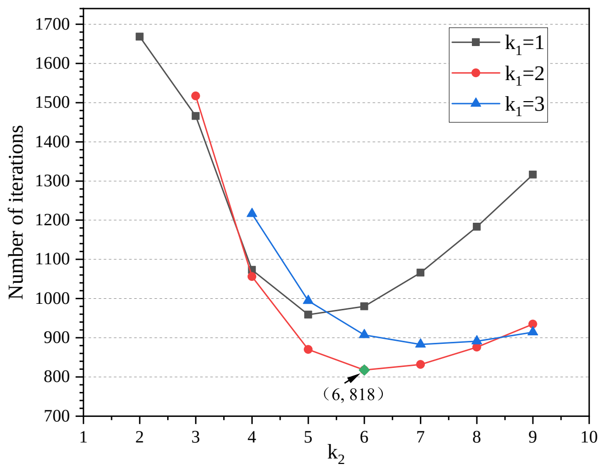

Figure 7: The relationship between (k_1, k_2) and the number of iterations.

Comparing the value of two numbers can be obtained by subtracting the two numbers and then determining whether the difference is positive or negative. The operands in the partial XGCD algorithm are in RSD representation without sign bits. For an RSD number, whether it is positive or negative can be determined by the position of its highest bit. We assume that the position serial numbers of an RSD number are $0, 1, 2, ..., N$ from left to right. Therefore, when the position of the highest bit of an RSD number is odd, the number is negative. Otherwise, it is positive. For two 1024-digit numbers, A and B in RSD representation, the architecture for comparing them is shown in Figure 9. First, an RSD subtractor is applied to calculate: $C = A - B$, and then a converter is needed to convert the digit 0 represented by 11 to 00 in C. We split the 1024-digit (2048-bit) number D into $16 \times 4 \times 32$-bit, then 4-bit and 32-bit leading one detectors (LODs) are used, refer to [AS06]. The architecture of the *Calculate Symbol* module is shown in the right block in Figure 9, and the output S is calculated. When $S = 1$, C is a negative number, and A is smaller than B; when $S = 0$, C is a positive number, and A is greater than B.

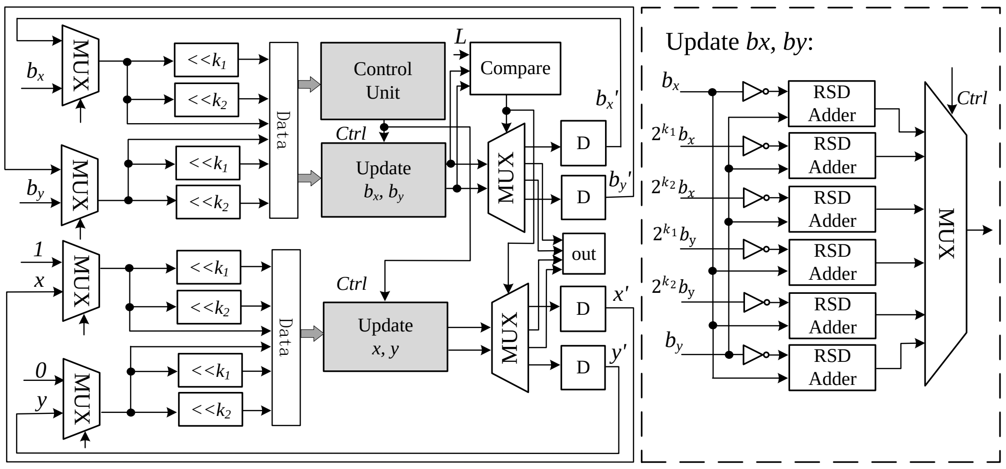

Figure 8: The architecture of the partial XGCD algorithm.

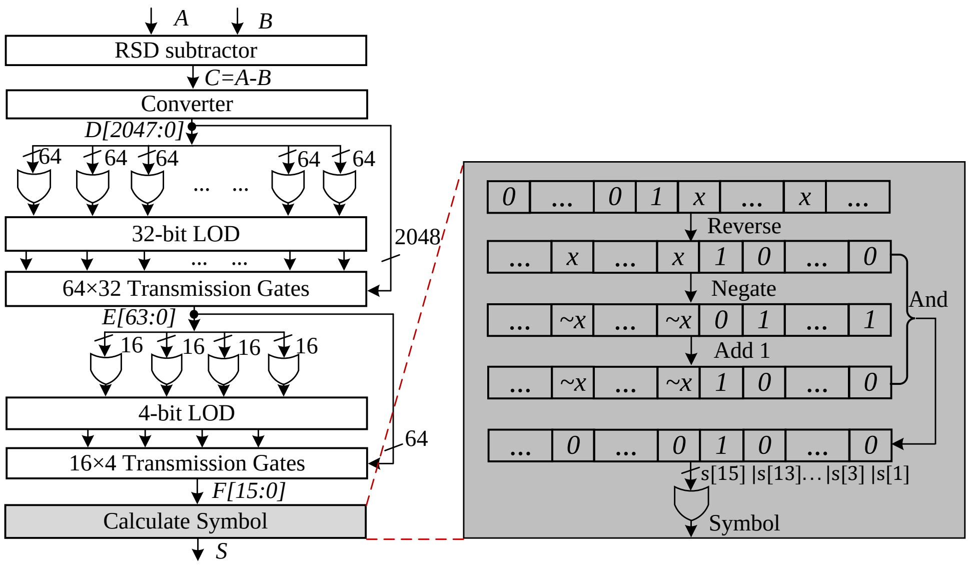

Figure 9: The architecture of compare module.

4 Instruction Scheduling

In the previous section, we describe how complex computations in the NUDUPL algorithm are designed in low-latency hardware. After designing the computation modules, we use the compact instruction scheduling method to utilize these modules fully, and the scheduling scheme is detailed in this section.

The proposed high-level architecture of squaring in the class group using the NUDUPL algorithm is depicted in Figure 10. The high-level architecture includes a ROM file that stores instructions, a control unit that reads instructions from ROM and output controls, a computation unit that contains mainly computation modules, MUXs that determine the order of written and read data, and four memories implemented by RAM or registers. The memories are used to store intermediate values within the NUDUPL algorithm and the final result of each squaring operation. Since the bit-width of the discriminant Δ of the binary quadratic form is 2048, and the proposed design uses the RSD representation, the bit-width of memories is set to 4096. The instructions were issued in advance and stored in the ROM, and the size depends on the bit-width and the number of instructions needed. When the start squaring signal arrives, the first instruction "LOAD" in the ROM is loaded into the control unit, and the input is fed into the memory. The control unit generates control signals according to the instructions in the ROM and reads the data from memories to the computation unit to perform the corresponding operations. When the computation is done, the results are controlled to write to the specified address in the memory. After the last instruction is executed, the final result can be fetched from the given address.

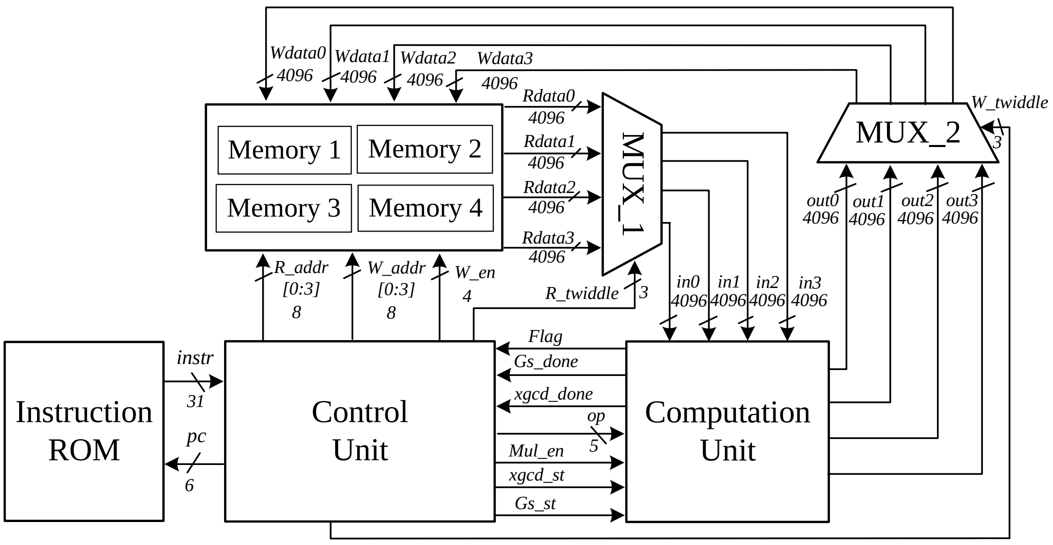

Figure 10: Proposed high-level architecture of squaring in the class group.

30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP code					Twiddle						Write enable				Read address								Write address							

Figure 11: The format of the instruction.

The ROM file stores 42 instructions and each instruction is 31 bits long. The instruction format is shown in Figure 11. Bits 0-7 indicate the write address used by memories, bits 0-1 for memory 0, bits 2-3 for memory 1, bits 4-5 for memory 2, and bits 6-7 for memory 3. Bits 8-15 indicate the read address for the corresponding memory. Bits 16-19 indicate whether the current data can be written into the corresponding memory. Bits 20-25 indicate which memory the data is being written/read to, where bits 20-22 for read selection and bits 23-25 for write selection. Bits 26-30 are used to place the operation (OP) codes.

The control unit reads the instruction based on the program counter (PC) and controls the operation in the computation unit by OP codes. The computation unit is shown in Figure 12, which contains XGCD and partial XGCD modules, a compare module, a Goldschmidt module for calculating the reciprocal of input, an LOD for preprocessing in the GS module, two 1024-digit RSD multipliers, some RSD adders and other MUXs for controlling. The MUX_0 generates control signals "CTRL"(includes *xgcd_ctrl*, *pxgcd_ctrl*, *cmp_ctrl*, *cmp_ctrl*, *etc*) for the corresponding computing module based on the OP codes. The input of the XGCD module, the partial XGCD module, and the compare module are read from the memories, with no data interaction with each other. The Goldschmidt

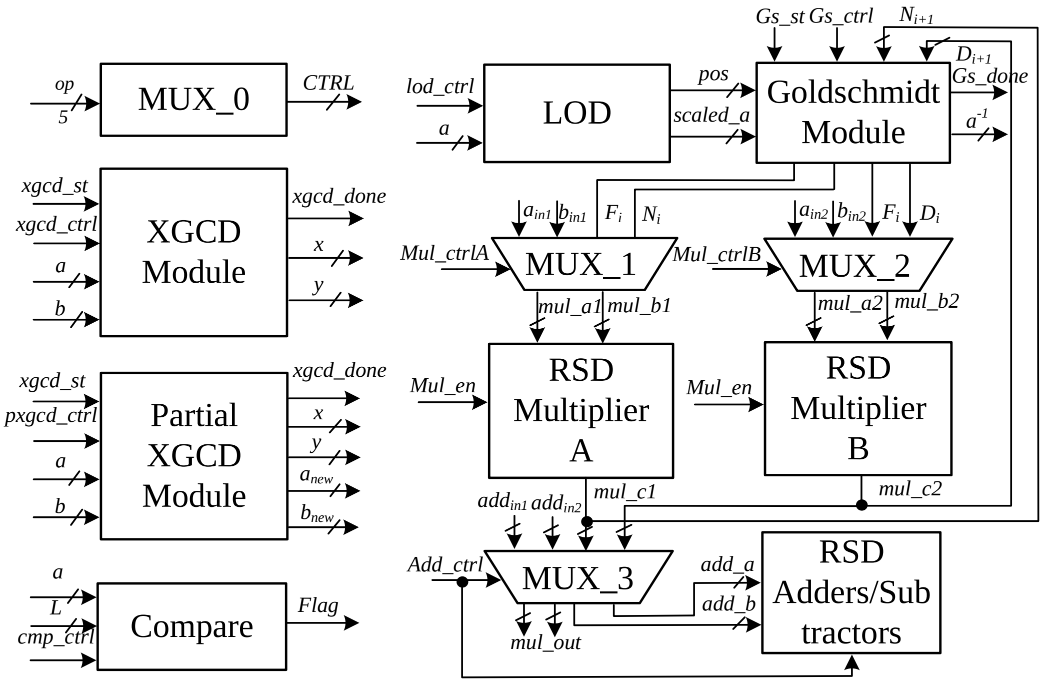

Figure 12: Proposed calculation modules in the computation unit.

module calculates the reciprocal of a number, e.g., a^{-1}, and the number needs to be preprocessed using the LOD. Besides, the $D_{i+1} = D_i * F_i$ and $N_{i+1} = N_i * F_i$ in the Goldschmidt iteration are performed using the external RSD multipliers. These RSD multipliers also perform the remaining multiplications in the NUDUPL algorithm, and MUX_1 and MUX_2 control their inputs. It is also necessary to instantiate several RSD adders/subtractors of different bit widths to calculate additions/subtractions of the whole NUDUPL algorithm. The input of RSD adders come from RSD multipliers or memories, and MUX_3 controls this.

There are many multiplications involved in the NUDUPL algorithm and the sizes of these multiplications are not unique. We use two 1024-digit RSD multipliers and implement all multiplications efficiently by scheduling these two multipliers. The main multiplication operations and the corresponding schedule methods are as follows:

- The multiplications in Goldschmidt algorithm. Two 1024-digit RSD multipliers calculate 1024-digit multiplications $D_{i+1} = D_i * F_i$ and $N_{i+1} = N_i * F_i$ in parallel shown in Algorithm 2.
- The 2048 × 1024 multiplications. For example, the multiplication $c \times y$ in step 4 of Algorithm 1, where c is a 2048-digit number and y is a 1024-digit number. This multiplication can be written as: $c[2047:1024] \times y << 1024 + c[1023:0] \times y$ and calculated with the two 1024-digit RSD multipliers in parallel.
- The 1024-digit squares. For example, the squares b_y^2 and b_x^2 in steps 8 and 27 of Algorithm 1, where b_y and b_x are both 1024-digit number. These squares can be calculated with two 1024-digit RSD multipliers in parallel.
- Other 1024-digit multiplications. For example, the multiplication $b_y \times D_y$ in step 7 of Algorithm 1, where b_y and D_y are both 1024-digit numbers. These multiplications can be calculated with either 1024-digit RSD multipliers.

As a result, we obtain a high utilization for the RSD multiplier through careful scheduling while reducing the computation time.

Table 4: The typical OP symbols and their operation descriptions.

OP symbol	Description
LOAD	Load the initial input (a, b, c) into specified memories.
LOD	This operation is the first computation step, calculating the effective bits of a. The XGCD module is also start at this step.
WFGS	Use GS module to calculate the reciprocal of a: a^{-1}.
CALQ, CALR, ADQR	Calculate the quotient q by: $q = N * a^{-1}$; Calculate the remainder r by: $r = c - a * q$; Make small adjustments to slightly deviated quotients and remainders.
LOADP	Load data into the partial XGCD module.
WFGCD, WFPGCD	Wait for XGCD calculation done; Wait for partial XGCD calculation done.
COMP	Compare the two input a and L and output the flag.
MUL, SQUA, ADDMUL, SUBMUL,...	Control RSD multipliers and RSD adders to perform arithmetic operations of specified bit width.
DONE	This indicates the calculation is done, and the final result (A, B, C) can be retrieved from the given address.

The OP codes determine the operations in the computation unit, and we define 21 different OP codes in advance. We describe each OP code as an OP symbol to illustrate the process. For example, we define OP code "00110" as "MUL", representing multiplication. The typical OP symbols and their corresponding operations are classified and described as Table 4.

5 Experimental Results and Comparisons

The proposed architecture of the NUDUPL algorithm is coded in SystemVerilog language and synthesized under TSMC 28-nm CMOS technology. The design is synthesized by Synopsys Design Compiler. We use Vivado 2018.3 EDA platform for behavioral-level simulation, and the correctness of the register-transfer level (RTL) model is verified by the software results provided by Chia's Network [Chi19].

5.1 Implementation Results and Discussion

One complete reduced squaring in the class group requires a squaring and a reduction operation. For squaring operation, one option is to use the squaring algorithm summarized in [Lon18], which is also the algorithm adopted by [ZTLW22], and another is to use the

NUDUPL algorithm which is the algorithm adopted in this work. For these two algorithms, the computation time required and the ratio of computation time of squaring are different. We run the code of two algorithms provided by Chia's competition [Chi19] on the same platform, Intel(R) Core(TM) i9-9900X @3.50GHz CPU. We run the code for 2^{21} iterations, and the average runtime for squaring and reduction is shown in Table 5. The results show that the total runtime is decreased by almost 30% by using the NUDUPL algorithm. Although it takes more time to compute the squarings when using the NUDUPL algorithm than the original squaring algorithm, the reduction almost takes no time (2%) by using the NUDUPL algorithm.

Table 5: The software runtime for two squaring algorithms.

Algorithm	Total runtime (μs)	Proportion of squaring	Proportion of reduction
Squaring in [Lon18] + Fast reduction	25.6	38%	62%
NUDUPL + Fast reduction	18.2	98%	2%

As shown in Table 5, the NUDUPL algorithm takes up almost all time for the squaring, so the design of NUDUPL is much more important than the design of reduction. We chose the reduction design in [ZTLW22] to obtain a complete hardware simulation result. We coded the overall design and run the code on the EDA platform for simulation. In the behavioral-level simulation, we randomly chose 1000 sets of inputs for simulation and calculated the average number of clock cycles for each squaring. According to the simulation results, a reduced squaring requires 1765 cycles, including 1726 (98%) for the NUDUPL algorithm and 33 (2%) for the fast reduction.

Table 6: Implementation results of the proposed NUDUPL architecture and XGCD architecture on TSMC 28-nm CMOS technology.

Module name	Area (mm^2)	Freq (MHz)	Latency (cc)	Total time (ns)
NUDUPL	5.237	847	1726	2038
-XGCD (unfolded)	0.336	847	601	709
XGCD only	0.216	2041	1202	589

We synthesized the proposed XGCD architecture under TSMC 28-nm CMOS technology separately, and the implementation result is shown in row "XGCD only" of Table 6. Since the critical path delay is significantly lower than that of other modules in the NUDUPL, we performed one unfolding operation on the XGCD module when placed it in the NUDUPL module. The proposed NUDUPL architecture was also synthesized under TSMC 28-nm CMOS technology and the implementation results of the proposed NUDUPL architecture and unfolded XGCD architecture in the overall design are shown in Table 6. As shown in Table 6, the critical path delay of the XGCD module is increased but the clock cycles for XGCD computation is decreased.

5.2 Comparison to Previous Work

We compare our implementation with existing hardware implementation for reduced squaring in the class group and the optimized software implementation over an Intel(R) Core(TM) i9-9900X @3.50GHz CPU. The comparison of implementations is shown in Table 7. Compared to the squaring implementation result in [ZST+20], we achieve a 3.2x speedup, even if the NUDUPL algorithm we use is more complex. Compared to the only existing complete implementation of reduced squaring in [ZTLW22], we achieve a 3.6x speedup. Compared to the optimized software implementation over an advanced CPU, our implementation is 9.1x faster. In summary, we achieve the fastest implementation for the squaring in the class group. Moreover, compared to the implementation of squaring under TSMC 28nm CMOS technology in [ZST$^+$20] and [ZTLW22], the area of our implementation is the smallest, because we avoid the time-consuming division in the XGCD and the partial XGCD and design the compact instruction to reuse the resources.

Table 7: Comparison to implementations of the reduced squaring in the class group.

Work	Platform	Area of squaring (mm^2)	Freq (MHz)	Time (μs)	
				Squaring	Total time
[ZST$^+$20]	TSMC 28nm	9.895	500	6.3	\
[ZTLW22]	TSMC 28nm	6.474	454	3.5	7.1
[Chi19]	Intel i9-9900X	\	3500	17.6	18.2
This work	TSMC 28nm	5.237	847	2.0	2.0

Table 8: Comparison to implementations of the XGCD.

Work	Platform	Freq (MHz)	Latency (cc)	Total time (ns)
[ZST$^+$20]	TSMC 28nm	500	3000	6000
[ZTLW22]	TSMC 28nm	454	1283	2825
[ZTW21]	TSMC 28nm	250	1623	6492
[SHT22]	TSMC 16nm	3890 (1525*)	1143	295 (750*)
[Chi19]	Intel i9-9900X	3500	\	8159
This work	TSMC 28nm	2041	1202	589

* Normalized to the 28nm technology based on the inverter fanout-of-4 (FO4) delay according to [SB17]. The delay is 9ps under TSMC 16nm [SHT22] and 22.5ps under TSMC 28nm, so the scale factor is: $22.5/9 = 2.5$.

The computation of XGCD served as the most time-consuming operation in the squaring in the class group, and we compare our implementation for the XGCD with the existing work. The comparison to the implementation of the XGCD is shown in Table 8. Our implementation is about 1.3x faster than the implementation in [SHT22] and significantly faster than the other implementations. Our implementation and the implementation in [SHT22] and both adopt the two-bit PM algorithm in [YZ86], but we implement it using the RSD representation and [SHT22] using the C-S representation. The reason for using RSD representation is stated in Section 2.3, and this is why our implementation is faster than the implementation in [SHT22].

6 Conclusion

This paper presents an ultra-low latency architecture of the squaring in class groups for VDF applications using redundant representation. The fastest implementation of the squaring in the class group is achieved by optimization at both algorithmic and architectural levels. We present low-latency arithmetic architectures, efficient XGCD and partial XGCD architectures, and scheduling methods with high resource utilization. Through optimization of both computational logic and control logic, the squaring architecture dramatically reduces the computation time and increases the area efficiency. Our implementation achieves a squaring speedup of 3.6x compared to the SOTA hardware implementation and a 9.1x speedup compared to the optimal software implementation over an advanced CPU. This result can be a reference for VDF applications. Besides, our implementation can facilitate other cryptographic applications without a trusted setup, such as timed commitments and zero-knowledge. Future work could consider calculating XGCD and partial XGCD with the same hardware or reducing the calculation time for the partial XGCD. It is also possible to consider adding RSD multipliers to reduce the multiplication time.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 62104097, in part by the Key Research Plan of Jiangsu Province of China under Grant BE2022098, and in part by the High-Level Personnel Project of Jiangsu Province under Grant JSSCBS20210034.

References

[AS06] Khalid H Abed and Raymond E Siferd. VLSI implementations of low-power leading-one detector circuits. In *Proceedings of the IEEE SoutheastCon 2006*, pages 279–284. IEEE, 2006.

[AVD21] Vidal Attias, Luigi Vigneri, and Vassil 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.

[Avi61] Algirdas Avizienis. Signed-digit numbe representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10(3):389–400, 1961.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[BDG17] Juan Benet, David Dalrymple, and Nicola Greco. Proof of replication. *Protocol Labs, July*, 27:20, 2017.

[BGB17] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in Ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[BV07] Johannes Buchmann and Ulrich Vollmer. Binary quadratic forms. In *Binary Quadratic Forms*, pages 9–20. Springer, 2007.

[Chi19] Chia's Network. https://github.com/Chia-Network/vdf-competition, July 2019. Accessed: 2022-3-NA.

[CL84] Henri Cohen and Hendrik W Lenstra. Heuristics on class groups of number fields. In *Number Theory Noordwijkerhout 1983*, pages 33–62. Springer, 1984.

[CP19] Bram Cohen and Krzysztof Pietrzak. The Chia network blockchain. *https://www.chia.net/assets/ChiaGreenPaper.pdf*, 2019.

[Eth21] Ethereum. Ethereum 2.0. https://ethereum.org/en/eth2/, 2021. Accessed: 2022-3-6.

[Fis19] Ben Fisch. Tight proofs of space and replication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 324–348. Springer, 2019.

[GLOW21] David Galindo, Jia Liu, Mihair Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 88–102. IEEE, 2021.

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

[HOH97] David L Harris, Stuart F Oberman, and Mark A Horowitz. SRT division architectures and implementations. In *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*, pages 18–25. IEEE, 1997.

[Jeb93] Tudor Jebelean. Comparing several GCD algorithms. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 180–185. IEEE, 1993.

[JP02] Michael J Jacobson and Alfred J Poorten. Computational aspects of NUCOMP. In *International Algorithmic Number Theory Symposium*, pages 120–133. Springer, 2002.

[Kor09] Peter Kornerup. Correcting the normalization shift of redundant binary representations. *IEEE Transactions on Computers*, 58(10):1435–1439, 2009.

[Lip12] Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In *International Conference on Applied Cryptography and Network Security*, pages 224–240. Springer, 2012.

[LM19] Russell WF Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *Annual International Cryptology Conference*, pages 530–560. Springer, 2019.

[Lon18] Lipa Long. Binary quadratic forms. *https://github.com/Chia-Network/vdf-competition/blob/master/classgroups.pdf*, 2018.

[LSS20] Esteban Landerreche, Marc Stevens, and Christian Schaffner. Non-interactive cryptographic timestamping based on verifiable delay functions. In *International Conference on Financial Cryptography and Data Security*, pages 541–558. Springer, 2020.

[MÖS20] Ahmet Can Mert, Erdinc Öztürk, and Erkay Savas. Low-latency ASIC algorithms of modular squaring of large integers for VDF evaluation. *IEEE Transactions on Computers*, 2020.

[Özt19] Erdinç Öztürk. Modular multiplication algorithm suitable for low-latency circuit implementations. *Cryptology ePrint Archive*, 2019.

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

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[San21] Ismail San. LLMonPro: Low-latency Montgomery modular multiplication suitable for verifiable delay functions. *Cryptology ePrint Archive*, 2021.

[SB17] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration (Amst.)*, 58:74–81, 2017.

[SHT22] Kavya Sreedhar, Mark Horowitz, and Christopher Torng. Fast large-integer extended GCD algorithm and hardware design for verifiable delay functions and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 163–187, 2022.

[SJH$^+$21] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar Weippl. Randrunner: Distributed randomness from trapdoor VDFs with strong uniqueness. 2021.

[Sor90] Jonathan Sorenson. The k-ary GCD algorithm. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[TCLM21] Sri Aravinda Krishnan Thyagarajan, Guilhem Castagnos, Fabian Laguillaumie, and Giulio Malavolta. Efficient cca timed commitments in class groups. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2663–2684, 2021.

[VDF19] VDF alliance. VDF alliance FPGA competition. https://www.vdfalliance.org/contest, September 2019. Accessed: 2022-3-6.

[vdP03] Alfred van der Poorten. A note on NUCOMP. *Mathematics of computation*, 72(244):1935–1946, 2003.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 379–407. Springer, 2019.

[YZ86] David YY Yun and Chang Nian 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*, pages 82–84, 1986.

[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.

[ZTLW22] Danyang Zhu, Jing Tian, Minghao Li, and Zhongfeng Wang. Low-latency hardware architecture for VDF evaluation in class groups. Cryptology ePrint Archive, Paper 2022/755, 2022. https://eprint.iacr.org/2022/755.

[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, 2021.