Modular Multiplication Algorithm Suitable For Low-Latency Circuit Implementations

Erdinç Öztürk
Faculty of Engineering and Natural Sciences
Sabancı University, Istanbul, Turkey
e-mail: erdinco@sabanciuniv.edu

***Abstract*—Modular multiplication is one of the most compute-intensive arithmetic operations. Most public-key cryptosytems utilize modular multiplications of integers of various lengths, depending on security requirements. Efficient algorithms and implementations are required to realize a practical public-key cryptosystem. Different parameters, such as area, power and time, can be optimized for different implementation requirements. Low latency was not as important as high throughput requirement for modular multiplication implementations before. However, with recent work on Verifiable Delay Functions (VDFs), a necessity for lowest possible latency for modular multiplication implementations emerged. VDFs are designed to take a prescribed time to realize the underlying computation that can be publicly verified. VDF constructions are required to utilize inherently sequential arithmetic operations. Efficient VDF constructions have been proposed recently, based on time-lock puzzles constructed by Rivest, Shamir and Wagner. An exponentiation operation in an RSA group needs to be realized for these VDF constructions. For these VDF constructions to be practical, low-latency modular multiplication algorithms and implementations are required. In this paper, a modular multiplication algorithm suitable for low-latency circuit implementations is proposed and an FPGA-optimized variant of this algorithm is presented.**

***Index Terms*—Verifiable Delay Function (VDF), Modular Multiplication, Reduction**

I. Introduction

Modular arithmetic is essential for various cryptographic families. RSA [1] is one of the most widely utilized public-key cryptosystems that is used for secure data transmission. In RSA, public key is generated by multiplying two secret prime numbers and security of RSA is based on integer factorization problem [2]. This public key is used as modulus for main operations of RSA and modulus length is determined according to acceptable security levels. For example, National Institute of Standards and Technology (NIST) currently recommends usage of key sizes of 2048 bits [3].

For RSA, main operation is modular exponentiation. There are various methods for efficient modular exponentiation implementations [4]. Modular multiplication and modular squaring of large integers are the most compute-intensive common building blocks of these methods. Therefore, performance of implementations of modular multiplication is directly related to performance of RSA operations.

There are well-studied high-level algorithms and methods enabling efficient modular multiplication implementations. One of the most commonly used modular multiplication algorithm is Montgomery Multiplication [5]. Montgomery Multiplication algorithm realizes $C' \equiv A \cdot B \cdot R^{-1} \ mod \ M$ instead of the desired $C = A \cdot B \ mod \ M$ outcome. To achieve the desired outcome, a post-processing step is required. Therefore, Montgomery Multiplication algorithm is efficient only for applications involving many modular arithmetic operations, such as exponentiation, due to inherent pre-processing and post-processing overheads.

Barrett Reduction [6] can also be utilized for implementing modular multiplication operation. Barrett Reduction algorithm computes the desired $C = A \cdot B \ mod \ M$ outcome, which makes it a better choice for a single modular multiplication scheme, instead of exponentiation which involves large number of modular multiplication operations. Although Barrett Reduction and Montgomery Reduction algorithms present similar complexities, they present different performances for different implementation settings [7], [8].

There are also other methods for modular multiplication, which can be employed for different realistic constraints. These implementations mostly focus on a high-throughput yield, instead of a low-latency yield. For public-key cryptographic applications, low-latency modular multiplication implementation is not a practical requirement. For resource-constrained devices, low-power is the main requirement for implementations. For datacenter applications, high-throughput is the main requirement. For client applications, a public-key operation in the milliseconds range is sufficient for a practical implementation. With the proposal of RSA-based Verifiable Delay Functions (VDFs) [9], [10], [11], a significant requirement for a low-latency modular multiplication operation has emerged.

A VDF is required to realize an inherently sequential computational task. Exponentiation in a group of unknown order is believed to have this property [10], and was used by Rivest, Shamir, and Wagner [12] to construct the notion of time-lock puzzles. The two recent VDF proposals by Pietrzak [9] and Wesolowski [11] are based on this notion. An exponentiation operation in an RSA group needs to be realized for these VDF constructions. For these VDF constructions to be practical, low-latency modular multiplication algorithms and implementations are required. It should be noted that for hardware implementations of VDF, circuit depth is the most important optimization constraint for modular multiplication.

Efficient constuctions for multiplication operation is crucial for the efficiency of modular multiplication. Both Montgomery and Barrett Reduction are high-level algorithms and they utilize integer multiplication operations as their low-level building blocks. There are algorithms that allow lower circuit depth than classical multiplication operation [13], [14]. Schonhage-Strassen is an NTT-based method presented in 1971 that can multiply two n-bit integers in $O(nlognloglogn)$ time [15]. Harvey et. al. presented an improved algorithm that can multiply two n-bit integers in $O(nlogn4^{logn})$ operations [16], [17]. However, efficient low-latency hardware implementations of these schemes do not exist in literature. A theoretical log depth circuit algorithm is presented in [14]. To the best of our knowledge, no practical low-latency implementation of this algorithm exists.

In this paper, a low-level low-latency multiplication algorithm is presented. This algorithm enables an $O(logn)$ depth circuit implementations for modular multiplication and modular squaring operations on large integers. A proof-of-concept FPGA implementation of the proposed algorithm is also presented. The results show that proposed algorithm is suitable for utilization in VDF circuits.

The remainder of the paper is structured as follows: Section II describes classical schoolbook multiplication algorithm for integers and for polynomials, which presents the inherent complexity of multiplication operations. Section III presents the modular multiplication algorithm suitable for low-latency circuit implementations. Section IV analyzes the complexity of the circuit in terms of area and timing. Section V describes an FPGA implementation and presents performance results.

II. Background

Throughout this paper, following notations are used. All operands are n-bit integers. An n-bit integer A is represented in radix r as $A = (A_{k-1}, A_{k-2}, ..., A_1, A_0)_r$, where $r = 2^d$ and $k = \lceil n/d \rceil$. Here, d is the digit length, A_i is i^{th} digit where $A_i \in [0, r-1]$ and k is the number of digits of A in radix r representation. The i^{th} digit A_i can be calculated as:

$$A_i = (A >> (d * i))\ mod\ r.$$

Multi-precision integer A can be written in terms of its digits as:

$$A = \sum_{i=0}^{k-1} A_i \cdot r^i.$$

Notation T^j is used to represent j^{th} element of an array T.

A k-digit integer A can be represented using a polynomial $A(x)$ of degree k-1 as:

$$A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i$$

where $x = r$. It should be noted that the polynomial representation is only used for ease of explanation of the algorithm, providing a sense of abstraction, due to a high level of redundancy.

A. Classical Digitwise Integer Multiplication

Given two k-digit integers A and B, Algorithm 1 presents a straightforward method of multiplying these two integers to realize the operation $C = A \cdot B$, where C is a $2k$-digit integer. It should be noted that this is a representative version of the classical digitwise multiplication algorithm, constructed in order to present the complexity of the operation. There are various other methods and orders of operation that can bu utilized to construct a classical digitwise multiplication algorithm.

Algorithm 1 Classical Digitwise Integer Multiplication

```
Input: $A = (A_{k-1}, A_{k-2}, ..., A_1, A_0)_r$
Input: $B = (B_{k-1}, B_{k-2}, ..., B_1, B_0)_r$
Output: $C = (C_{2k-1}, C_{2k-2}, ..., C_1, C_0)_r$
 1: for $i$ from 0 to $2k-1$ do
 2:     $C_i = 0$
 3: end for
 4: for $i$ from 0 to $k-1$ do
 5:     for $j$ from 0 to $k-1$ do
 6:         $T^j = (A_i \cdot B_j)$                    ▷ $T^j = (T_1^j, T_0^j)_r$
 7:     end for
 8:     $carry = 0$
 9:     for $j$ from 0 to $k-1$ do
10:         $\{carry, C_{j+i}\} = C_{j+i} + T_0^j + carry$
11:     end for
12:     $C_{k+i} = C_{k+i} + carry$
13:     $carry = 0$
14:     for $j$ from 0 to $k-1$ do
15:         $\{carry, C_{j+i+1}\} = C_{j+i+1} + T_1^j + carry$
16:     end for
17: end for
```

Algorithm 1 demonstrates the inherent complexity of a multi-precision multiplication operation. There are a total of k^2 d-bit core multiplication operations, which are followed by costly addition operations. Although these addition operations can be realized via redundant arithmetic using efficient constructions such as Wallace Tree [18], there is an inherent costly n-bit carry propagation, which makes this algorithm not suitable for low-latency applications such as VDF.

B. Classical Polynomial Multiplication

Given two polynomials $A(x)$ and $B(x)$ of degree k-1, Algorithm 2 presents a straightforward method of multiplying these two polynomials to realize the operation $C(x) = A(x) \cdot B(x)$, where $C(x)$ is a polynomial of degree $2k$-2.

Algorithm 2 demonstrates the inherent complexity of a polynomial multiplication operation. Although seemingly simpler than integer multiplication operation with much shorter carry propagation, polynomial multiplication in this form is not suitable for exponentiation. Since coefficients of the inputs are d-bit numbers, coefficients of the output are required to be d bits wide to be suitable for exponentiation operation. However, for Algorithm 2, each output C_i is $2d$+$log(k)$ bits wide, since k $2d$-bit numbers are added together in the worst case.

In this paper, a low-level multiplication algorithm that is taking advantage of the shorter carry chain utilized in polynomial multiplications is proposed. Polynomial multiplication algorithm defined in Algorithm 2 is modified to be suitable for use in modular exponentiation.

Algorithm 2 Classical Polynomial Multiplication

```
Input: $A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i$
Input: $B(x) = \sum_{i=0}^{k-1} B_i \cdot x^i$
Output: $C(x) = \sum_{i=0}^{2k-2} C_i \cdot x^i$
1: for $i$ from 0 to $2k-2$ do
2:     $C_i = 0$
3: end for
4: for $i$ from 0 to $k-1$ do
5:     for $j$ from 0 to $k-1$ do
6:         $C_{j+i} = C_{j+i} + (A_i \cdot B_j)$
7:     end for
8: end for
```

III. The Proposed Modular Multiplication Algorithm

In this section, an incomplete modular multiplication algorithm working on integers with redundant representations that is suitable for modular exponentiation is presented.

A complete modular multiplication algorithm is required to realize

$$C = A \cdot B \ mod \ M$$

operation, where C is an integer satisfying the condition $0 < C < M$. Since the proposed algorithm is designed to work in exponentiation operation, output of the modular multiplication operation can be in redundant form, not satisfying the said condition.

First, we define a function that converts a k-digit integer $A = (A_{k-1}, A_{k-2}, ..., A_1, A_0)_r$ to a polynomial of degree k-1 with d-bit coefficients:

$$A(x) = inttopoly(A)$$

where

$$A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i.$$

We also need to define a function that converts a polynomial of degree K with arbitrary-length coefficients back to integer form:

$$A = polytoint(A(x))$$

where

$$A = \sum_{i=0}^{K} A_i \cdot 2^{d \cdot i}.$$

A. Multiplication

The proposed algorithm is a modified version of the polynomial multiplication algorithm presented in Algorithm 2. Main problem of this algorithm is that the output polynomial is in different redundant representation than the inputs of the algorithm, which makes it inefficient for modular exponentiation. A modified version of the algorithm with an attempt to fix this issue is presented in Algorithm 3.

Each product T can be written as a polynomial as:

$$T_1 x + T_0$$

Algorithm 3 Modified Polynomial Multiplication Algorithm

```
Input: $A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i, 0 \leq A_i < 2^d$
Input: $B(x) = \sum_{i=0}^{k-1} B_i \cdot x^i, 0 \leq B_i < 2^d$
Output: $C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2k-1} C_i \cdot x^i, 0 \leq C_i < 2^{d+1}$
1: for $i$ from 0 to $2k-1$ do
2:     $D_i = 0$
3: end for
4: for $i$ from 0 to $k-1$ do
5:     for $j$ from 0 to $k-1$ do
6:         $T = (A_i \cdot B_j)$                 ▷ $T = (T_1, T_0)_r$
7:         $D_{i+j} = D_{i+j} + T_0$
8:         $D_{i+j+1} = D_{i+j+1} + T_1$
9:     end for
10: end for
11: for $i$ from 0 to $2k-1$ do
12:     $C_i = 0$
13: end for
14: for $i$ from 0 to $2k-2$ do            ▷ $\forall i, D_i = (D_{i_1}, D_{i_0})_r$
15:     $C_i = C_i + D_{i_0}$
16:     $C_{i+1} = C_{i+1} + D_{i_1}$
17: end for
18: $C_{2k-1} = C_{2k-1} + D_{2k-1}$
```

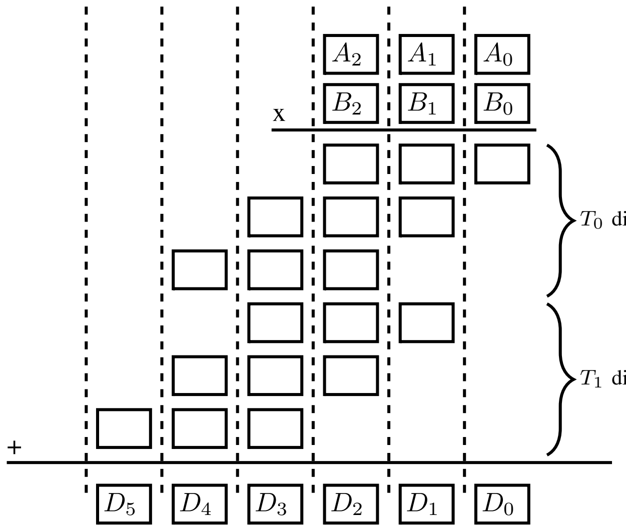


Fig. 1. Initial computations of Algorithm 3.

Therefore, T_1 can be accumulated into the higher coefficient D_{i+j+1} instead of D_{i+j}. Therefore, it should be noted that, output and inputs of Algorithm 3 satisfies the equation:

$$polytoint(C(x)) = polytoint(A(x)) \cdot polytoint(B(x)).$$

First part of Algorithm 3 is detailed in Figure 1 for $k = 3$. Final part of the Algorithm involves addition of two d-bit numbers, which results in a $(d + 1)$-bit integer. Therefore, coefficients of output polynomial of Algorithm 3 are $(d+1)$-bit integers, which is still not acceptable for exponentiation. In order to fix this issue, another modification to the algorithm is proposed. A redundant-representation polynomial multiplication algorithm is presented in Algorithm 4.

Algorithm 4 Redundant-Representation Polynomial Multiplication Algorithm

```
Input: $A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i, 0 \le A_i < 2^{d+1}$
Input: $B(x) = \sum_{i=0}^{k-1} B_i \cdot x^i, 0 \le B_i < 2^{d+1}$
Output: $C(x) = A(x) \cdot B(x) = \sum_{i=0}^{2k} C_i \cdot x^i, 0 \le C_i < 2^{d+1}$
 1: for $i$ from 0 to $2k$ do
 2:     $D_i = 0$
 3: end for
 4: for $i$ from 0 to $k-1$ do
 5:     for $j$ from 0 to $k-1$ do
 6:         $T = (A_i \cdot B_j)$                    ▷ $T = (T_2, T_1, T_0)_r$
 7:         $D_{i+j} = D_{i+j} + T_0$
 8:         $D_{i+j+1} = D_{i+j+1} + T_1$
 9:         $D_{i+j+2} = D_{i+j+2} + T_2$            ▷ $0 \le T_2 < 2^2$
10:     end for
11: end for
12: for $i$ from 0 to $2k$ do
13:     $C_i = 0$
14: end for
15: $C_{2k} = D_{2k}$                                ▷ $0 \le D_{2k} < 2^2$
16: for $i$ from 0 to $2k-1$ do                      ▷ $\forall i, D_i = (D_{i_1}, D_{i_0})_r$
17:     $C_i = C_i + D_{i_0}$
18:     $C_{i+1} = C_{i+1} + D_{i_1}$
19: end for                                          ▷ $0 \le C_i < 2^{d+1}\ \forall i \in [0, 2k]$.
```

It should be noted that, output and inputs of Algorithm 4 still satisfies Equation III-A.

Step 6 of Algorithm 4 shows the multiplication operation of the coefficients of two polynomials. These multiplications can be realized in parallel utilizing a single (d+1)-bit multiplier for each multiplication. Since coefficients of input polynomails in Algorithm 4 are d+1 bits wide, result of $T = (A_i \cdot B_j)$ operation in Step 6 is $2d$+2 bits wide. Steps 7, 8, 9 of Algorithm 4 shows accumulation of the results of each core multiplication operation to their respective coefficient. Instead of accumulating the result of each coefficient into its respective coefficient and growing the coefficient size of the intermediate result to $2d$+$log(k)$ bits, each ($2d$+2)-bit result T is split into 3 digits utilizing the property:

$$T = T_2 * x^2 + T_1 * x^1 + T_0$$

Since weight of each coefficient is d bits, this operation does not change the validity of the result.

Algorithm 4 is detailed in Figure 2 for $k = 3$. It can be seen that each T result is split into 3 digits and accumulated into corresponding resulting coefficients. In this algorithm, carry propagation happens only across 2 coefficients. Therefore, critical path of this algorithm includes a $2d$-bit carry chain, instead of n bits. Since both input coefficients and output coefficients are both d+1 bits wide, this algorithm is suitable for use in modular exponentiation.

B. Reduction

Reduction operation is defined as $C = A \cdot B \ mod \ M$. However, within the scope of this work, this does not need to be realized fully. Any redundant representation satisfying the condition $C \equiv A \cdot B \ mod \ M$ can be utilized for reduction in exponentiation operation. The only requirement is that the result of the multiplication operation needs to be reduced back to similar redundant form as input of the multiplication.

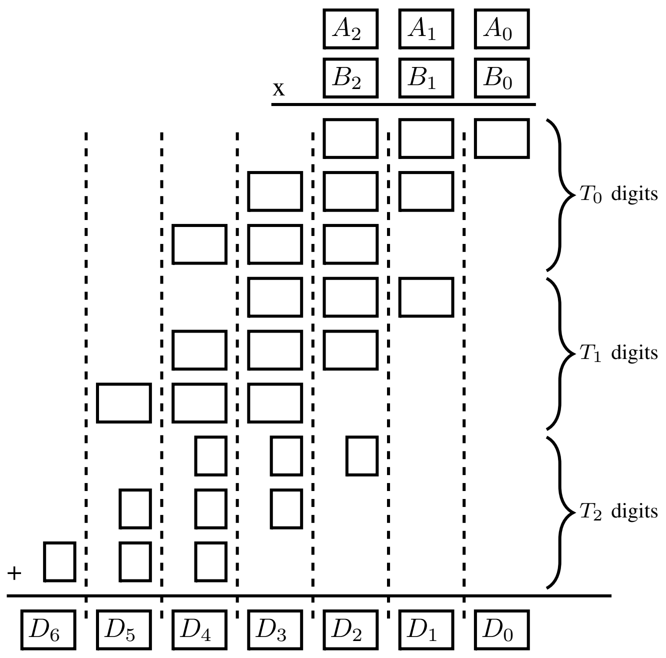


Fig. 2. Initial computations of Algorithm 4.

Different reduction algorithms achieve different form of results. A low-latency Montgomery Reduction [5] as well as a low-latency Barrett Reduction [6] implementation utilizes 2 sequential large-integer multiplications for the reduction operation after the multiplication. A full modular multiplication utilizing Montgomery Reduction or Barrett Reduction includes 3 separate large-integer multiplications in its critical path. For a low-latency implementation, this can be improved.

Assume a fixed modulus is used for the reduction operation. This is feasible considering the fact that for public key operations, same modulus is utilized for long periods of time. Also, for VDF implementations, a fixed modulus can be utilized.

Algorithm 5 Precomputation of look-up tables.

```
Input: Modulus $M$
Output: Precomputed tables $LUT[k+1][2^{d+1}][k]$
1: for $i$ from 0 to $k$ do
2:     for $j$ from 0 to $2^{d+1}-1$ do
3:         $T(x) = inttopoly(j \cdot 2^{n+d*i} \ mod \ M)$
4:         for $t$ from 0 to $k-1$ do
5:             $LUT[i][j][t] = T_t$
6:         end for
7:     end for
8: end for
```

Redundant-representation polynomial $C(x) = \sum_{i=0}^{2k} C_i \cdot x^i$ can be converted to its integer form as:

$$C = \sum_{i=0}^{2k} C_i \cdot 2^{d \cdot i}.$$

Therefore, each coefficient $C_i \cdot x^i$ holds the value:

$$C_i \cdot x^i = C_i \cdot 2^{d \cdot i}.$$

Since $k \cdot d = n + \epsilon$, where $0 \leq \epsilon < d$, $C_0 : C_{k-1}$ coefficients of $C(x)$ are not reduced. However, remaining $C_k : C_{2k}$ coefficients need to be reduced. Each coefficient $C_i \cdot x^i$ can be rewritten as:

$$T(x) = C_i \cdot x^i = inttopoly(C_i \cdot 2^{d \cdot i}).$$

which is a polynomial of degree k-1. Using this approach, look-up tables as shown in Algorithm 5 are precomputed. The aim of generating these look-up tables is to store precomputed modular values of each $C_k : C_{2k}$ coefficients resulting from Algorithm 4.

Using these precomputed look-up tables, reduction operation is applied on the result of the multiplication operation from Algorithm 4, as shown in Algorithm 6.

Algorithm 6 Reduction Algorithm

```
Input: $C(x) = A(x) \cdot B(x)$, where
 1: $C(x) = \sum_{i=0}^{2k} C_i \cdot x^i, 0 \leq C_i < 2^{d+1}$
 2: $A(x) = \sum_{i=0}^{k-1} A_i \cdot x^i, 0 \leq A_i < 2^{d+1}$
 3: $B(x) = \sum_{i=0}^{k-1} B_i \cdot x^i, 0 \leq B_i < 2^{d+1}$
Input: $LUT[k+1][2^{d+1}][k]$
Output: $Res(x) = \sum_{i=0}^{k} Res_i \cdot x^i, 0 \leq Res_i < 2^{d+1}$
 4: for $i$ from 0 to $k-1$ do
 5:     $D_i = C_i$
 6: end for
 7: for $i$ from $k$ to $2k$ do
 8:     for $j$ from 0 to $k-1$ do
 9:         $D_j = D_j + LUT[i-k][C_i][j]$
10:     end for
11: end for
12: for $i$ from 0 to $k$ do
13:     $Res_i = 0$
14: end for
15: for $i$ from 0 to $k-1$ do          ▷ $\forall i, D_i = (D_{i_1}, D_{i_0})_r$
16:     $Res_i = Res_i + D_{i_0}$
17:     $Res_{i+1} = Res_{i+1} + D_{i_1}$
18: end for                             ▷ $\forall i, 0 \leq Res_i < 2^{d+1}$.
```

Algorithm 6 is detailed in Figure 3 for $k = 3$.

As stated before, the only requirement is that the result of the multiplication operation needs to be reduced back to similar redundant form as input of the multiplication. It can be seen in Algorithm 6 that output is a polynomial of degree k with (d+1)-bit coefficients, whereas the inputs $A(x)$ and $B(x)$ are polynomials of degree $k-1$ with $(d+1)$-bit coefficients. Another level of reduction is required to bring the output polynomial to the same redundant form as the input. Since this will add another sequential operation to the reduction algorithm, increasing the critical path significantly, another option is studied.

Instead of reducing the output to a polynomial of degree k-1, input of the multiplication is modified as a polynomial of degree k. Here, redundant representation of each operand is defined as a polynomial of degree k with d+1-bit coefficients.

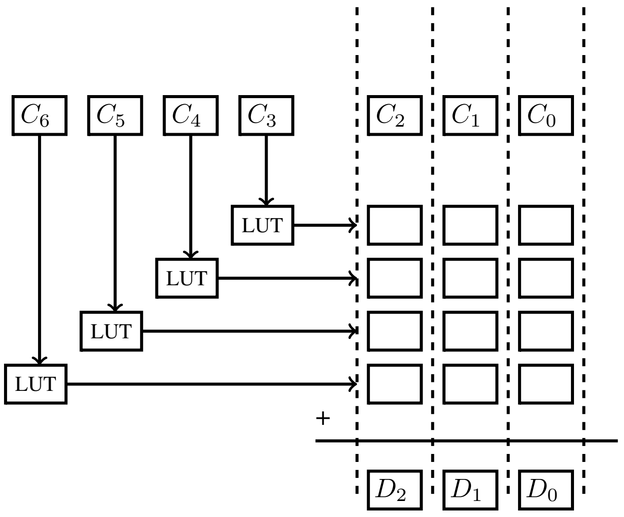


Fig. 3. Initial computations of Algorithm 6.

C. Modular Multiplication Operation

In this section, 3 different algorithms are presented. These algorithms can be utilized in combination to realize a modular multiplication operation.

Modified multiplication algorithm of redundant representation is detailed in Algorithm 7. This is a slightly modified version of Algorithm 4, to make it suitable for use in modular exponentiation, combined with the reduction algorithm presented in Algorithm 6.

Algorithm 7 Final Redundant-Representation Polynomial Multiplication Algorithm

```
Input: $A(x) = \sum_{i=0}^{k} A_i \cdot x^i, 0 \leq A_i < 2^{d+1}$
Input: $B(x) = \sum_{i=0}^{k} B_i \cdot x^i, 0 \leq B_i < 2^{d+1}$
Output: $C(x) = \sum_{i=0}^{2k+2} C_i \cdot x^i, 0 \leq Res_i < 2^{d+1}$
 1:
 2: for $i$ from 0 to $2k+2$ do
 3:     $D_i = 0$
 4: end for
 5: for $i$ from 0 to $k$ do
 6:     for $j$ from 0 to $k$ do
 7:         $T = (A_i \cdot B_j)$               ▷ $T = (T_2, T_1, T_0)_r$
 8:         $D_{i+j} = D_{i+j} + T_0$
 9:         $D_{i+j+1} = D_{i+j+1} + T_1$
10:         $D_{i+j+2} = D_{i+j+2} + T_2$       ▷ $0 \leq T_2 < 2^2$
11:     end for
12: end for
13: for $i$ from 0 to $2k+2$ do
14:     $C_i = 0$
15: end for
16: $C_{2k+2} = D_{2k+2}$                       ▷ $0 \leq D_{2k+2} < 2^2$
17: for $i$ from 0 to $2k+1$ do                ▷ $\forall i, D_i = (D_{i_1}, D_{i_0})_r$
18:     $C_i = C_i + D_{i_0}$
19:     $C_{i+1} = C_{i+1} + D_{i_1}$
20: end for                 ▷ $0 \leq C_i < 2^{d+1}$ $\forall i \in [0, 2k+2]$.
```

For Steps 8, 9 and 10 of Algorithm 7, at most $2k+1$ d-bit numbers and k 2-bit numbers are accumulated, as can be seen from Figure 2. Since each D_i is assumed to be a 2-digit number, $log(2k+2)$ cannot be larger than d. For combinations of n and d that results in $log(2k+2) > d$, final steps of Algorithm 7 needs to be modified. Thus, this algorithm works as it is for $n < d \cdot 2^{d-2}$. For example, if $d = 8$, Algorithm 7 works as it is for $n <= 512$. If $d = 16$, Algorithm 7 works as it is for $n <= 262144$.

Since number of coefficients of the redundant representations is modified, the number of precomputed look-up tables also needs to be modified. Algorithm 8 details precomputations of required look-up tables, which is a slight modified version of Algorithm 5.

Algorithm 8 Precomputation of look-up tables for Algorithm 9

```
Input: Modulus $M$
Output: Precomputed tables $LUT[k+3][2^{d+1}][k]$
1: for $i$ from 0 to $k+2$ do
2:     for $j$ from 0 to $2^{d+1}-1$ do
3:         $T(x) = inttopoly(j \cdot 2^{n+d*i}\ mod\ M)$
4:         for $t$ from 0 to $k-1$ do
5:             $LUT[i][j][t] = T_t$
6:         end for
7:     end for
8: end for
```

Finally, Algorithm 9 details the modified reduction algorithm. It can be seen that output of this reduction algorithm is in identical redundant representation as the input of the multiplication algorithm presented in Algorithm 7.

Algorithm 9 Final Reduction Algorithm

```
Input: $C(x) = A(x) \cdot B(x)$, where
1: $C(x) = \sum_{i=0}^{2k+2} C_i \cdot x^i, 0 \le C_i < 2^{d+1}$
2: $A(x) = \sum_{i=0}^{k} A_i \cdot x^i, 0 \le A_i < 2^{d+1}$
3: $B(x) = \sum_{i=0}^{k} B_i \cdot x^i, 0 \le B_i < 2^{d+1}$
Input: $LUT[k+3][2^{d+1}][k]$
Output: $Res(x) = \sum_{i=0}^{k} Res_i \cdot x^i, 0 \le Res_i < 2^{d+1}$
4: for $i$ from 0 to $k-1$ do
5:     $D_i = C_i$
6: end for
7: for $i$ from $k$ to $2k+2$ do
8:     for $j$ from 0 to $k-1$ do
9:         $D_j = D_j + LUT[i-k][C_i][j]$
10:    end for
11: end for
12: for $i$ from 0 to $k$ do
13:     $Res_i = 0$
14: end for
15: for $i$ from 0 to $k-1$ do          $\triangleright \forall i, D_i = (D_{i_1}, D_{i_0})_r$
16:     $Res_i = Res_i + D_{i_0}$
17:     $Res_{i+1} = Res_{i+1} + D_{i_1}$
18: end for                              $\triangleright \forall i, 0 \le Res_i < 2^{d+1}$.
```

For Step 9 of Algorithm 9, $k+3$ d-bit and 1 $(d+1)$-bit numbers are accumulated, as can be seen from Figure 3. Since each D_i is assumed to be a 2-digit number, $log(k+5)$ cannot be larger than d. For combinations of n and d that results in $log(k+5) > d$, final steps of Algorithm 9 needs to be modified. Since $log(k+5) < log(2k+2)$ for large numbers, reduction part does not constitute a bottleneck for parameter selection requirements.

Polynomials $A(x)$, $B(x)$ and $Res(x)$ are representing integers A, B and Res, respectively. The equation:

$$polytoint(Res(x)) \equiv polytoint(A(x)) \cdot polytoint(B(x))\ mod\ M \quad (1)$$

ensures that Algorithms 7, 8 and 9 can be utilized for an incomplete modular multiplication operation.

IV. Complexity Analysis

Complexity of the modular multiplication operation as a combination of 3 algorithms described in section III-C heavily depends on the choice of d.

From a circuit implementation point of view, Algorithm 7 has the following in its critical path:

- **Core multiplier**: Step 7 of Algorithm 7 consists of $(k+1)^2$ independent core multiplications of $(d+1)$-bit coefficients. $(k+1)^2$ core multipliers can be utilized to realize this step, with a circuit depth of a single $(d+1)$-bit core multiplier.
- **Adder Tree**: Results of the $(k+1)^2$ core multiplications are accumulated as shown in Steps 8, 9 and 10 of Algorithm 7. This accumulation can be realized using an adder tree, such as Wallace Tree [18]. A Wallace tree can be constructed with O(n^2) area and $O(logn)$ circuit depth.
- **Adder**: After the Wallace Tree in Algorithm 7, coefficients needs to be converted from redundant carry-save representation to integer representation. All of the operations after the Wallace Tree can be combined with a single (2d+2)-bit carry lookahead adder, since focus is implementing the fastest possible multiplier. Although carry lookahead adder has $O(logn)$ circuit depth, choice of d is not relevant to the integer length n. Therefore, this delay can be included as a constant delay.

Algorithm 8 can be used to precompute values that are stored in a memory structure. Depending on the value of d, complexity of the look-up table memory structure varies.

Algorithm 9 has the following in its critical path:

- **Look-up Table Read**: Each precomputed value needs to be read from the look-up table for Step 9 of Algorithm 9. Since complexity of these look-up tables depend on the choice of d, this delay can be considered as a constant.
- **Adder Tree**: Precomputed values read from the look-up tables are accumulated as shown in Step 9 of Algorithm 9. As can be seen from the algorithm, $(k+3)$ 16-bit and one 17-bit numbers are accumulated together. This accumulation can be realized using an adder tree, as described above, with an $O(logn)$ circuit depth.
- **Adder**: After the adder tree in Algorithm 9, coefficients needs to be converted from redundant carry-save representation to integer representation. This can be realized

TABLE I
PERFORMANCE RESULTS OF FPGA IMPLEMENTATION OF THE PROPOSED MODULAR MULTIPLICATION ALGORITHM.

bit-length	Area			Time			
	LUT	FF	DSP	logic delay	route delay	total	logic percentage
128	24120	1472	81	3.8ns	8.05ns	11.85ns	32.04%
256	87622	2952	289	4.405ns	13.14ns	17.55ns	25.11%
512	329868	4143	1089	4.73ns	20.95ns	25.68ns	18.42%

similar to the addition described for multiplication, with a constant delay.

V. PROOF OF CONCEPT IMPLEMENTATION

For proof of concept, an FPGA-optimized version of the proposed modular multiplication operation is implemented. XILINX VIRTEX-7 FPGA families utilize functional blocks named DSP48E1, which include $24 \cdot 17$-bit unsigned multipliers [19]. These multipliers can be used as $(d+1)$-bit core multipliers, realizing the operation $T = (A_i \cdot B_j)$ from Step 6 of Algorithm 7. Therefore, d=16 is chosen for a proof of concept FPGA implementation.

Although a theoretical $O(logn)$ circuit depth, reduction operation is impractical to realize for $d = 16$ since look-up tables with 17-bit read address need to be implemented. A single look-up table of this form holds $n \cdot 2^{16}$ bits of precomputed data. For large n, this is impractical for both area and timing reasons. For a low-latency FPGA implementation, a better reduction method is proposed for larger choices of d.

Algorithm 10 Precomputation of look-up tables for $d = 16$.

```
Input: Modulus $M$
Output: $LUT8[k+3][256][k]$, $LUT9[k+1][512][k]$
 1: for $i$ from 0 to $k+2$ do
 2:     for $j$ from 0 to 255 do
 3:         $T(x) = inttopoly(j \cdot 2^{n+16*i} \ mod \ M)$
 4:         for $t$ from 0 to $k-1$ do
 5:             $LUT8[i][j][t] = T_t$
 6:         end for
 7:     end for
 8: end for
 9: for $i$ from 0 to $k+2$ do
10:     for $j$ from 0 to 511 do
11:         $T(x) = inttopoly(j \cdot 2^{n+16*i+8} \ mod \ M)$
12:         for $k$ from 0 to $k-1$ do
13:             $LUT9[i][j][t] = T_t$
14:         end for
15:     end for
16: end for
```

First, look-up tables as shown in Algorithm 10 are precomputed. The aim of generating these look-up tables is to store precomputed modular values of each $C_k : C_{2k+2}$ coefficients resulting from Algorithm 7. Instead of generating a 17-bit look-up table, one 8-bit look-up table and one 9-bit look-up table is generated for each coefficient. 8-bit look-up tables are generated for the low 8 bits of the 17-bit coefficients and 9-bit look-up tables are generated for the remaining high 9 bits of the 17-bit coefficients.

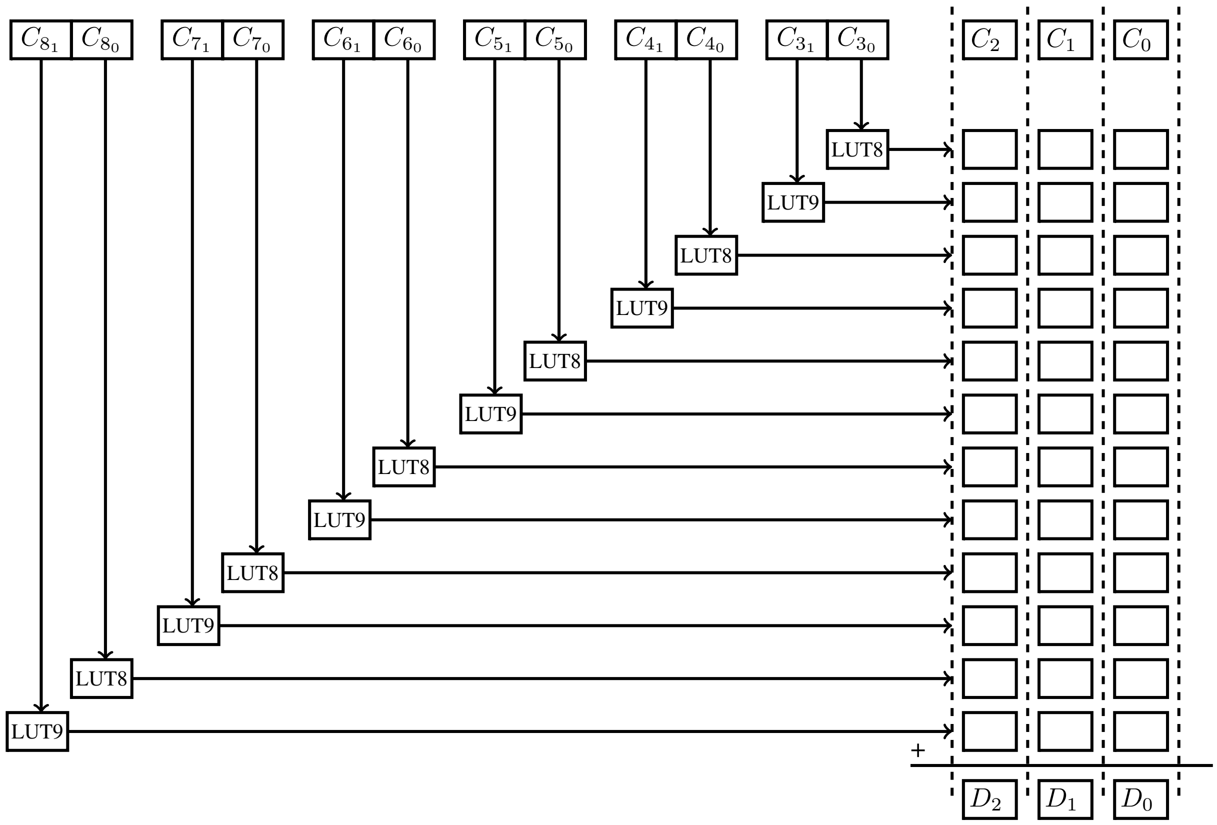
Using the precomputed tables LUT8 and LUT9, reduction operation is applied on the result of the multiplication operation from Algorithm 7, as shown in Algorithm 11.

Algorithm 11 FPGA-Optimized Reduction Algorithm for $d = 16$

```
Input: $C(x) = A(x) \cdot B(x)$, where
 1: $C(x) = \sum_{i=0}^{2k+2} C_i \cdot x^i, 0 \le C_i < 2^{17}$
 2: $A(x) = \sum_{i=0}^{k} A_i \cdot x^i, 0 \le A_i < 2^{17}$
 3: $B(x) = \sum_{i=0}^{k} B_i \cdot x^i, 0 \le B_i < 2^{17}$
Input: $LUT8[k+3][256][k]$
Input: $LUT9[k+3][512][k]$
Output: $Res(x) = \sum_{i=0}^{k} Res_i \cdot x^i, 0 \le Res_i < 2^{17}$
 4: for $i$ from 0 to $k-1$ do
 5:     $D_i = C_i$
 6: end for
 7: for $i$ from $k$ to $2k+2$ do
 8:     $T_H = C_i >> 8$                          ▷ higher 9 bits
 9:     $T_L = C_i \ mod \ 2^8$                     ▷ lower 8 bits
10:     for $j$ from 0 to $k-1$ do
11:         $D_j = D_j + LUT8[i-k][T_L][j]$
12:         $D_j = D_j + LUT9[i-k][T_H][j]$
13:     end for
14: end for
15: for $i$ from 0 to $k$ do
16:     $Res_i = 0$
17: end for
18: for $i$ from 0 to $k-1$ do            ▷ $\forall i, D_i = (D_{i_1}, D_{i_0})_r$
19:     $Res_i = Res_i + D_{i_0}$
20:     $Res_{i+1} = Res_{i+1} + D_{i_1}$
21: end for                               ▷ $\forall i, 0 \le Res_i < 2^{17}$.
```

First part of Algorithm 11 is detailed in Figure 4 for $k = 3$. It can be seen that for this version of the algorithm, 2(k+3) 16-bit numbers and one 17-bit number are accumulated together, which still results in $O(logn)$ circuit depth.

Algorithms 7, 10 and 11 are implemented targeting XILINX Virtex UltraScale+ XCVU9P FPGA using XILINX Vivado Design Suite. For proof of concept implementation, n =128, 256 and 512 are chosen with d =16. Results are presented in Table I. Logic delay and routing delay are extracted from the reports generated by the Vivado tool. Although slightly worse logic delays exist in other critical paths, only the worst critical paths are reported here. Logic delay numbers demonstrate the $O(logn)$ circuit depth of the proposed modular multiplication algorithm. Routing delays demonstrate the wiring delays introduced by the $O(n^2)$ area of the proposed multiplier.



C_{8_1} C_{8_0} C_{7_1} C_{7_0} C_{6_1} C_{6_0} C_{5_1} C_{5_0} C_{4_1} C_{4_0} C_{3_1} C_{3_0}
C_2 C_1 C_0
LUT8
LUT9
+
D_2 D_1 D_0

Fig. 4. Initial computations of Algorithm 11.

VI. Conclusion

In this paper, a novel and practical modular multiplication algorithm is presented. Proof of concept implementation results show that this algorithm is suitable for low-latency applications. This algorithm can be tailored for different constraints to achieve lowest possible latency for different environments. It should be noted that a custom 2048-bit implementation of this algorithm was utilized to solve the LCS35 puzzle [20] in almost 60 days [21].

References

[1] Rivest, R. L. and Shamir, A. and Adleman, L., “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: {http://doi.acm.org/10.1145/359340.359342}

[2] D. Boneh, “Twenty years of attacks on the rsa cryptosystem,” *NOTICES OF THE AMS*, vol. 46, pp. 203–213, 1999.

[3] Barker, E. and Dang, Q., “Recommendation for Key Management,” *NIST Special Publication 800-57 Part 3 Revision 1*, 01 2015.

[4] D. M. Gordon, “A survey of fast exponentiation methods,” *Journal of Algorithms*, vol. 27, no. 1, pp. 129 – 146, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677497909135

[5] Montgomery, Peter L., “Modular Multiplication without Trial Division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[6] P. Barrett, “Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor,” in *Advances in Cryptology — CRYPTO’ 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[7] Bosselaers, Antoon and Govaerts, René and Vandewalle, Joos, “Comparison of Three Modular Reduction Functions,” in *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO ’93. Berlin, Heidelberg: Springer-Verlag, 1994, pp. 175–186. [Online]. Available: {http://dl.acm.org/citation.cfm?id=188105.188147}

[8] M. Knezevic and F. Vercauteren and I. Verbauwhede, “Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods,” *IEEE Transactions on Computers*, vol. 59, no. 12, pp. 1715–1721, Dec 2010.

[9] Krzysztof Pietrzak, “Simple Verifiable Delay Functions,” Cryptology ePrint Archive, Report 2018/627, 2018, https://eprint.iacr.org/2018/627.

[10] Dan Boneh and Joseph Bonneau and Benedikt Bünz and Ben Fisch, “Verifiable Delay Functions,” Cryptology ePrint Archive, Report 2018/601, 2018, https://eprint.iacr.org/2018/601.

[11] Benjamin Wesolowski, “Efficient verifiable delay functions,” Cryptology ePrint Archive, Report 2018/623, 2018, https://eprint.iacr.org/2018/623.

[12] Rivest, R. L. and Shamir, A. and Wagner, D. A., “Time-lock Puzzles and Timed-release Crypto,” Cambridge, MA, USA, Tech. Rep., 1996.

[13] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, “Nfllib: Ntt-based fast lattice library,” in *Topics in Cryptology - CT-RSA 2016*, K. Sako, Ed. Cham: Springer International Publishing, 2016, pp. 341–356.

[14] P. W. Beame and S. A. Cook and H. J. Hoover, “Log Depth Circuits For Division And Related Problems,” in *25th Annual Symposium on-Foundations of Computer Science, 1984.*, vol. , no. , Oct 1984, pp. 1–6.

[15] A. Schönhage and V. Strassen, “Schnelle multiplikation großer zahlen,” *Computing*, vol. 7, no. 3, pp. 281–292, Sep 1971. [Online]. Available: https://doi.org/10.1007/BF02242355

[16] David Harvey and Joris van der Hoeven, “Faster integer multiplication using short lattice vectors,” *CoRR*, vol. abs/1802.07932, 2018. [Online]. Available: {http://arxiv.org/abs/1802.07932}

[17] ——, “Faster integer multiplication using plain vanilla FFT primes,” *CoRR*, vol. abs/1611.07144, 2016. [Online]. Available: {http://arxiv.org/abs/1611.07144}

[18] C. S. Wallace, “A Suggestion for a Fast Multiplier,” *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, Feb 1964.

[19] XILINX Inc., “UG479-7 Series DSP48E1 Slice User Guide,” 2018, https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[20] Ron Rivest, *Description of the LCS35 Time Capsule Crypto-Puzzle*, 1999. [Online]. Available: http://people.csail.mit.edu/rivest/lcs35-puzzle-description.txt

[21] THE WIRED, *A Programmer Solved a 20-Year-Old, Forgotten Crypto Puzzle*, 2019. [Online]. Available: https://www.wired.com/story/a-programmer-solved-a-20-year-old-forgotten-crypto-puzzle/