Low Latency Montgomery Multiplier for Cryptographic Applications

Muhammad Huzaifa¹, Khalid Javeed¹, Safiullah Khan² and Atif Raza Jafri¹

¹Department of Computer Engineering, Bahria University, Islamabad Pakistan
²Department of Computer Engineering, College of IT Convergence, Gachon University, Seongnam, Republic of Korea

Correspondence Author: Khalid Javed (malikkhaled@gmail.com)

Received: August 11, 2020; Revised: October 29, 2020; Accepted: July 02, 2021

Abstract

In this modern era, data protection is very important. To achieve this, the data must be secured using either Public-Key (PKC) or Private-Key Cryptography (P-KC). PKC eliminates the need of sharing keys at the beginning of communication. PKC systems such as ECC and RSA are implemented for different security services such as the key exchange between sender, receiver, and key distribution between different network nodes and authentication protocols. PKC is based on computationally intensive finite field arithmetic operations. In the PKC schemes, Modular Multiplication is the most critical operation. Usually, this operation is performed by Integer Multiplication (IM) followed by a reduction module M. However, the reduction step involves a long division operation that is expensive in terms of area, time, and resources. Montgomery multiplication algorithm facilitates faster MM operation without the division operation. In this paper, low latency hardware implementation of the Montgomery multiplier is proposed. Many interesting and novel optimization strategies are adopted in the proposed design. The proposed Montgomery multiplier is based on School-Book (SB) multiplier, Karatsuba-Ofman (KO) algorithm, and fast adders techniques. The Karatsuba-Ofman algorithm and School-Book multiplier recommend; cutting down the operands into smaller chunks while adders facilitate fast addition for large size operands. The proposed design is simulated, synthesized, and implemented using Xilinx ISE Design Suite by targeting different Xilinx Field-Programmable Gate Arrays (FPGA) devices for different bit sizes (64-1024). The proposed design is evaluated based on computational time, area consumption, and throughput. The implementation results show that the proposed design can easily outperform the state-of-the-art.

Index Terms: Elliptic Curve Cryptography, Field-Programmable Gate Arrays, Karatsuba-Ofman Algorithm, Montgomery Multiplication, Private-Key Cryptography.

I. INTRODUCTION

Cryptography has generally been divided into two types, Private-Key Cryptography - symmetric and Public-Key Cryptography (PKC) - asymmetric [1]. In Private-Key Cryptography, encryption and decryption use the same key but in PKC there is a key pair, private key, and public key. The public key is open and accessible for everyone, but the private or personal key is solely known by the message receiver. Both keys are mathematically related. In PKC, the public key is used for the encryption of data while for decryption, the private key is used. It is impossible to derive the private key from the public key. This property allows us to freely exchange public keys over the network.

PKC algorithms that are most commonly used are RSA and elliptic curve cryptography (ECC) [2]. Most of the PKC algorithms implementation need arithmetic operations over a finite field of large characteristics. These operations are modular addition (or subtraction) modular multiplication and modular inversion with large size operands over the finite fields. The security of the scheme can be increased by increasing operand lengths. In RSA, large operands are used for the same security level as compared to ECC. Hence, the ECC is the preferred scheme, particularly when utilized in resource-constrained environments.

The software implementation of ECC achieves a greater flexibility level at the price of higher computational time [3]. The hardware implementations of ECC provide higher performance if the modular multiplier is designed efficiently. Operations like modular multiplication and inversion are most time and resource-consuming. Modular multiplication is comparatively faster in terms of hardware and software when compared to modular inversion. To avoid inversion, extra multiplications are added and changing the coordinate system to a projective coordinates system. By doing so, the total performance depends on the modular multiplication, which causes the bottleneck in cryptosystems that need to be optimized. Modular multipliers in the literature are divided into three types. The regular technique for modular multiplication is division-using modulus (M). The division is costly in terms of execution time and utilization of resources. The second type of modular multipliers is Interleaved Modular Multiplication (IMM), which can do the reduction during multiplication. The last type of modular multiplier is Montgomery Modular Multiplication.
(MMM), which is faster when the operands are large compared to the other methods [4]. Large operand divided into several small parts and then Karatsuba-Ofman algorithm is employed which gives the best output in terms of area and performance [5]. Cryptographic applications are deployed in several smart devices like mobile phones and Wi-Fi devices. These cryptographic applications utilize the varying size of mathematical operands starting from 160 to 1024-bits. The security of the devices can be increased by increasing the size of the operands. Therefore, we require operations with varying field sizes depending upon the applications.

II. RELATED WORK

Efficient implementations in hardware are categorized into two main classes, word-wise, and bit-wise implementations. In modern Field-Programmable Gate Arrays (FPGA), dedicated multipliers are used in word-wise implementations. The utilization of dedicated multipliers on FPGA is faster than the standard design-based FPGA. The operands are divided into different parts and these parts utilize the dedicated multipliers for multiplication, which offer faster speed in time-critical applications for the word-wise implementations.

The investigators provided the concept to use 64x64 bit cores for multiplication and provided different implementations for different field sizes [6]. Authors in 2013 presented a concept for the modular multiplier that can be used in Barreto-Naehrig (BN) curves. They allotted the special prime number for implementations of BN curves and utilized the atypical division to adjust the FPGA, Digital Signal Processor (DSP) block [7]. The authors extended the concept to high speed and low-cost Montgomery algorithm with the carry-save adder for addition operations [8]. A Carry-save adder is used to cut off the extra clock cycles for implementation and conversion. Investigators advanced the concept of high-speed Montgomery multiplier architecture using digit-serial computation [9]. It uses binary multiplication in high-radix partial multiplication. Consecutive zero-bit multiplications can be performed within the one-clock cycle. Some authors performed the addition of 512-bit operands and multiplication of 256-bit operands by utilization of the carry chain of 64-bit with soft-core multiplier and achieved 188 MHz frequency [10]. Investigators elaborated an implementation scheme of using IP cores of the FPGA with the addition of 512-bit and 256-bit multiplication to achieve 50% better results as compared to standard implementations [11].

Researchers contributed to implementing the hardware processor for Error Correcting Code [12]. Full block MMM that utilizes 256-bit integer multiplication is achieved by 16-bit multipliers in cascaded form and this method is pursued until the specified size of the multiplier is attained. Fast carry look-ahead adders are used for the addition. A design concept was put forward to divide the operands and performing computations on them. The complexity is not depending on the operand size, it relies on the divided part of the operand [13].

To increase the efficiency of MMM architecture, many solutions are provided using the Karatsuba algorithm. Another contribution was that; the design concept of 256-bit MMM using the dedicated multiplier on FPGA with pipelining stages. Using the Karatsuba algorithm, the operands are divided into two parts to cut down the number of the multiplier on FPGA. The hardware architecture decreased the clock cycle utilization [14]. Another design concept was put forward which consists of a series of nine multipliers of 64-bit to create a block to deploy the Karatsuba algorithm-based number multiplier [15]. The integer multiplier can be used to create a huge block of 256-bit Montgomery modular multiplier. The design provides low space architecture and path delay cost is decreased to increased iterations. An investigator G. C. Chow and his associates presented the design to decrease routing delay that is increased in large multipliers. For decreasing the routing delay dividing the operands into small parts and these parts are using the dedicated multipliers to increase the efficiency of the hardware [16]. The authors extended the design concept of the Karatsuba algorithm to divide the operand into two parts and use it in the MMM algorithm with higher radix. The architecture uses the dedicated blocks of the multiplier, which increases the utilized space of the hardware [17].

In research, investigators advanced the design concept of the Karatsuba algorithm, divided into 4 levels, use in the Montgomery modular multiplier. Using the splitting method operand is divided into two parts. The divided part is again divided into two other parts in the reappearance style until the divided parts are length matches with the DSP blocks of an FPGA [18]. The Lookup-Table (LUT)-based Montgomery multiplier was designed by researchers rather than the FPGA dedicated multipliers. Radix-4 based modular multiplier with the serial interleaved design is employed [19]. The authors came up with the design concept of parallel interleaved modular multiplier implementation of hardware architecture. According to the architecture, an operand is working on four parallel processing elements to complete the dedicated task according to the algorithm [20]. Some researchers presented the design to deploy of systolic architecture array in Montgomery modular multiplier. The systolic architecture array, repeating the structures in parallel, to overcome the path delay [21]. The authors provided the design to deploy a radix-4 serial multiplier in MMM with the ladderding method of power to cut off the 50% in clock cycles [22].

III. CONTRIBUTION

In the PKC system, modular multiplication is a basic operation. The MMM algorithm is commonly adopted for modular multiplication implementation. To enhance the efficiency of the MMM algorithm, we have utilized the School-Book (SB) multiplier and Karatsuba-Ofman (KA) algorithm. Karatsuba-Ofman algorithm suggests dividing the operands into smaller parts. The number of multiplications increases when we further decrease the size of the chunks of the operands. In this paper, we have optimized hardware utilization and computation time by utilizing the SB multiplier and KA algorithm. We have worked on three methods i.e. two-parts, four-parts, and
eight-parts splitting. After the splitting of operands, these operands utilize the integer multiplier architecture based on operands splitting techniques and the MMM algorithm. Increasing the speed of the integer multiplier can help to enhance the overall efficiency of MMM. The proposed design is simulated; synthesized and implemented using Xilinx ISE Design Suite by targeting different Xilinx FPGA devices for different bit sizes (64-1024). The proposed design of MMM is evaluated based on computational time, area consumption, and throughput.

IV. BACKGROUND

A. Karatsuba-Ofman Algorithm

In this algorithm, multiplication complexity is overcome by dividing the operands into equivalent small chunks. A School-Book multiplication complexity is O(\(n^2\)). The strategy found by Karatsuba-Ofman by dividing the operand into parts has decreased the complexity to O(\(n^\log_23\)) [23]. During the multiplication of two numbers A and B, the algorithm suggests dividing these into the higher and lower parts as given below:

\[
A_1 = (a_{n-1} \ldots \ldots a_{n/2}) \\
A_0 = (a_{n/2-1} \ldots \ldots a_0) \\
B_1 = (b_{n-1} \ldots \ldots b_{n/2}) \\
B_0 = (b_{n/2-1} \ldots \ldots b_0)
\]

The operand A and B can be written as:

\[
A = A_0 + 2^nA_1 \\
B = B_0 + 2^nB_1
\]

Now the multiplication result is given below:

\[
\text{Output} = A \times B = A_0B_0 + 2^n(A_1B_1 + A_0B_0 - (A_1 - A_0)(B_1 - B_0)) + 2^{2n}A_1B_1
\]

(1)

Four multiplications are required if the School-Book method is adopted as shown in eq. (1). They utilize the four DSP blocks for multiplication. The result of the School-Book multiplier is fast but they use more resources. However, using the Karatsuba technique, the required number of multiplications is three as shown in eq. (2):

\[
\text{Output} = A \times B = A_0B_0 + 2^n(A_1B_1 + A_0B_0 - (A_1 - A_0)(B_1 - B_0)) + 2^{2n}A_1B_1
\]

(2)

The number of multiplications required to multiply the two operands in the Karatsuba algorithm is three, which can save one multiplication operation and increase the addition and subtraction operations. They have also utilized the signed bit operations which decrease the speed of the multiplication but utilize fewer resources of hardware. The repeated division of operands into small parts until reaching the required size of the operand increases efficiency. Utilizing the School-Multiplying multiplier improves the speed of multiplication and the Karatsuba algorithm uses fewer resources of hardware.

B. Operands Splitting

The operands may be divided into a different number of parts and utilize either School-Book or Karatsuba technique for the multiplication. Table I shows the multipliers required according to operand splitting with two different techniques of multiplication.

| Division | Size | Multiplier |
|----------|------|------------|
| School Book Method | Karatsuba Algorithm |
| 2-Part | N/2 | 4 | N/2 | 3 |
| 3-Part | N/3 | 9 | N/3 | 6 |
| 4-Part | N/4 | 16 | N/4 | 10 |
| 5-Part | N/5 | 25 | N/5 | 15 |

When we further divide, the operand size is decreasing, but the number of multipliers is increased with the addition of adder and subtraction operations to produce the final output. Further splitting of the operands may not be suitable because the area of the hardware is increased due to extra addition and subtraction operations.

C. Two-Parts Splitting

Karatsuba algorithm suggests dividing the operands into two parts. The main operations include the calculation of the difference of divided operands \((A_1 - A_0)\) and \((B_1 - B_0)\), it is an effective and vital part of the Karatsuba calculation. Then compute the product of the divided operands \((A_1 - A_0) \text{ and } (B_1 - B_0)\). The result of these two operations is \((A_1B_1 + A_0B_0 - (A_1 - A_0)(B_1 - B_0))\) which saves one multiplication compared to the School-Book method. The latest FPGA chips contain DSP blocks which consist of built-in multipliers. In Virtex-5 and Virtex-6, the DSP blocks consist of an 18x25 asymmetrical and signed dedicated multiplier. The size of the dedicated multiplier is \(n\)-bit the output of the multiplication is \(2n\)-bits. Which can use the three dedicated multipliers for the \(2n\)-bits multiplication, the hardware architecture utilized the three dedicated multipliers. Operand size is less than \(2n\)-bits it always utilizes the three dedicated multipliers in the hardware architecture. It is a generalized theory of multiplication. The old FPGA devices do not support a dedicated multiplier. The latest FPGA devices are fast for multiplication due to the dedicated multiplier inside the DSP blocks, which also perform the addition inside the dedicated multiplier.

D. Four-Parts Splitting

Repeatedly Karatsuba-Ofman is applied on the operands. Four parts are obtained after applying operation splitting into the two parts of the operands. The four sections of the operands are given below:

\[
A = A_0 + 2^nA_1 + 2^{2n}A_2 + 2^{3n}A_3 \\
B = B_0 + 2^nB_1 + 2^{2n}B_2 + 2^{3n}B_3
\]

The general output of the multiplication is given below:

\[
\text{Output} = A \times B = A_0B_0 + 2^n(A_1B_0 + A_0B_1 + A_1B_2 + 2^n(A_3B_0 + A_2B_1 + A_1B_3 + A_2B_3) + 2^{2n}(A_2B_2 + A_1B_3) + 2^{3n}(A_2B_3 + A_3B_2) + 2^{4n}A_3B_3
\]

(3)
The above condition demonstrates that 16 multiplications and 15 adders are needed for the output of eq. (3). If the size of the operand part is equal to DSP block size then we needed 16 dedicated multipliers to perform the multiplication of the equation. After applying the Karatsuba-Ofman calculation on the same equation the output is:

\[
\text{Output} = A \times B = P_{00} + 2^n(P_{11} + P_{00} - D_{10}) + 2^{2n}(P_{22} + P_{11} + P_{00} - D_{20}) + 2^{3n}(P_{33} + P_{22} + P_{11} + P_{00} - D_{30} - D_{21}) + 2^{4n}(P_{33} + P_{22} + P_{11} + P_{00} - D_{30} - D_{21}) + 2^{5n}(P_{33} + P_{22} - D_{32}) + 2^{6n}(P_{33})
\]

(4)

After comparing both equations, it is observed that reduction occurs in the number of multiplications from 16 to 10 and with an increase in 15 adders and 18 subtractions. The operand size remains the same as one-fourth of the first operand. If the single multiplication utilizes the single DSP block then 10 numbers of DSP block are acquired to do complete multiplication. The 6 numbers of the DSP block were saved through Karatsuba-Ofman calculation. These numbers of operations utilize in the main equation:

\[
P_{00} = A_0B_0 \\
P_{11} = A_1B_1 \\
P_{22} = A_2B_2 \\
P_{33} = A_3B_3 \\
D_{10} = (A_1 - A_0)(B_1 - B_0) \\
D_{20} = (A_2 - A_0)(B_2 - B_0) \\
D_{30} = (A_3 - A_0)(B_3 - B_0) \\
D_{21} = (A_2 - A_1)(B_2 - B_1) \\
D_{31} = (A_3 - A_1)(B_3 - B_1) \\
D_{32} = (A_3 - A_2)(B_3 - B_2)
\]

E. Eight-Parts Splitting

Eight parts are obtained after the application of splitting operation into the two parts of the operands recursively. The eight sections of the operands are given below:

\[
A = A_0 + 2^{2n}A_1 + 2^{3n}A_2 + 2^{4n}A_3 + 2^{6n}A_6 + 2^{7n}A_7 \\
B = B_0 + 2^{2n}B_1 + 2^{3n}B_2 + 2^{4n}B_4 + 2^{5n}B_5 + 2^{6n}B_6 + 2^{7n}B_7
\]

The general output of the multiplication is given below:

\[
\text{Output} = A \times B = A_0B_0 + 2^{2n}(A_0B_1 + A_1B_0) + 2^{3n}(A_0B_2 + A_2B_0) + 2^{4n}(A_0B_3 + A_3B_0) + 2^{5n}(A_0B_4 + A_4B_0) + 2^{6n}(A_0B_6 + A_6B_0) + 2^{7n}(A_0B_7 + A_7B_0) + 2^{2n}(A_1B_2 + A_2B_1) + 2^{3n}(A_1B_3 + A_3B_1) + 2^{4n}(A_1B_4 + A_4B_1) + 2^{5n}(A_1B_5 + A_5B_1) + 2^{6n}(A_1B_6 + A_6B_1) + 2^{7n}(A_1B_7 + A_7B_1) + 2^{3n}(A_2B_3 + A_3B_2) + 2^{4n}(A_2B_4 + A_4B_2) + 2^{5n}(A_2B_5 + A_5B_2) + 2^{6n}(A_2B_6 + A_6B_2) + 2^{7n}(A_2B_7 + A_7B_2) + 2^{4n}(A_3B_4 + A_4B_3) + 2^{5n}(A_3B_5 + A_5B_3) + 2^{6n}(A_3B_6 + A_6B_3) + 2^{7n}(A_3B_7 + A_7B_3) + 2^{5n}(A_4B_5 + A_5B_4) + 2^{6n}(A_4B_6 + A_6B_4) + 2^{7n}(A_4B_7 + A_7B_4) + 2^{6n}(A_5B_6 + A_6B_5) + 2^{7n}(A_5B_7 + A_7B_5) + 2^{7n}(A_6B_7 + A_7B_6) + 2^{8n}(A_7B_7)
\]

(5)

The above condition demonstrates that 64 multiplications and 63 adders are needed for the output of the equation.

F. Montgomery Algorithm

A strategy for faster modular multiplication has been presented in 1985 by Peter L. Montgomery. The MMM architecture computes A x B mod M, where A and B are certain whole numbers and M is a large prime number [24]. Regular methodologies for processing the remainder involve the division task. In Montgomery multiplication shift and adds operations replace the costly division operation. Shift and adds operations strategy work only in the Montgomery domain. Before the task, the operands are first converted into the domain of the Residue Number System. After completing the operation, the output is re-converted. Word length must be selected in the power of two in the selection of Radix (R) and Modulus (M) must be smaller than radix R. For the run of algorithm R and M must be a prime number. Whether for modulus M, an n bit is a positive number of A and B are two n bit operands. In modular multiplication, Output = A x B mod M where 0 ≤ A; B < M.

V. FPGA IMPLEMENTATION

Integrated circuits such as Field Programmable Gate Arrays (FPGA) could be programmed by the user after fabrication. FPGA devices contain Configurable Logic Blocks (CLB) which are connected through programmable interconnects. This ability of the FPGA devices has made them suitable hardware accelerators for different applications and they are largely deployed in cryptographic applications. The modern FPGA devices provide the dedicated portion for software and hardware cores and configurable blocks. In modern Xilinx FPGAs different memory, cores, and dedicated blocks for the arithmetic operations are available which are already tailored for high speed and low power applications. These cores are easily changeable according to requirements and multiple blocks can be utilized at the same time. The CLB provides the facility to the programmer to minimize the code to maximize the speed of the hardware.

A. Integer Multiplier

The performance of the MMM fully depends on the Integer Multiplier (IM) efficiency. In this paper, we have deployed the School-Book algorithm and Karatsuba-Ofman algorithm to increase the performance of the IM. We adopt the three approaches to increase the efficiency of IM, i.e. two-part splitting four-part splitting, and eight-part splitting. They are discussed with their results.

B. Two-Parts Splitting Multiplier

Integer multiplication using the two-part splitting technique is given in Algorithm-1 as shown in figure 1. It describes all the steps involved in partial product generation and accumulation to achieve the final product. In a two-part splitting algorithm, the operands can be sliced into two equal parts using the Karatsuba-Ofman approach and generate the product using IM. Here the two unsigned multipliers of N/2-bits and one signed multiplier of (N/2 + 1)-bits generate the third partial product. Three multiplications are executed in parallel with the help of a multiplier.

In Algorithm-1 as shown in figure 1, steps 1 to 12 explain the generation of partial product. eq. (2) represents that
the output utilizes the three multipliers. The reduction of the multiplier is achieved through the Karatsuba-Ofman algorithm. In the algorithm, steps 13 to 15 shows that the results of partial products are utilized as the inputs for the adders. Step 16 is the final addition to generate the final output product. The splitting depth of the operand is one.

**Algorithm-1: Two Parts Splitting Multiplication Algorithm**

1. Input A, B
2. \( A = \sum_{k=0}^{n-1} 2^k A_k \)
3. \( B = \sum_{k=0}^{n-1} 2^k B_k \)
4. Output Out=A \times B
5. for \( i = 1; i \geq 0; i = i - 1 \) do
6. \( j = i - 1 \)
7. \( P_{i,j} = A_j \times B_j \)
8. while \( j \geq 0 \) do
9. \( D_{j,j} = (A_j - A_{j+1}) \times (B_j - B_{j+1}) \)
10. \( j = j - 1 \)
11. end
12. end
13. \( S_0 = P_{00} \)
14. \( S_1 = P_{01} + P_{00} - D_{10} \)
15. \( S_2 = P_{02} + P_{01} + P_{00} - D_{11} - D_{20} \)
16. \( S_3 = P_{03} + P_{02} + P_{01} + P_{00} - D_{12} - D_{21} \)
17. \( S_6 = P_{04} + P_{03} - D_{32} \)
18. \( S_7 = P_{05} + P_{04} - D_{33} \)
19. \( \text{out} = \sum_{k=0}^{n-1} 2^k S_k \)
20. return Out

**Figure 1: Two Parts Splitting Multiplication Algorithm (Algorithm-1)**

### C. Four-Parts Splitting Multiplier

In four parts splitting technique, the depth for splitting is two. It shows that the operands are further subdivided divided into two parts. The prime benefit of four-part splitting is to optimize the resources. The four-part splitting multiplier utilized the fundamental multiplier of the DSP block in Virtex-6.

**Algorithm-2: Four Parts Splitting Multiplication Algorithm**

1. Input A, B
2. \( A = \sum_{k=0}^{n-1} 2^k A_k \)
3. \( B = \sum_{k=0}^{n-1} 2^k B_k \)
4. Output Out=A \times B
5. for \( i = 3; i \geq 0; i = i - 1 \) do
6. \( j = i - 1 \)
7. \( P_{i,j} = A_j \times B_j \)
8. while \( j \geq 0 \) do
9. \( D_{j,j} = (A_j - A_{j+1}) \times (B_j - B_{j+1}) \)
10. \( j = j - 1 \)
11. end
12. end
13. \( S_0 = P_{00} \)
14. \( S_1 = P_{01} + P_{00} - D_{10} \)
15. \( S_2 = P_{02} + P_{01} + P_{00} - D_{11} - D_{20} \)
16. \( S_3 = P_{03} + P_{02} + P_{01} + P_{00} - D_{12} - D_{21} \)
17. \( S_6 = P_{04} + P_{03} - D_{32} \)
18. \( S_7 = P_{05} + P_{04} - D_{33} \)
19. \( \text{out} = \sum_{k=0}^{n-1} 2^k S_k \)
20. return Out

**Figure 2: Four Parts Splitting Multiplication Algorithm (Algorithm-2)**

Figure 3, shows the architecture for the four-part splitting technique and Algorithm-2 describes the required steps in the generation of the result as depicted in figure 2. In a four-part splitting algorithm, the input can be split into four equal parts by using the Karatsuba-Ofman algorithm.

Here in figure 3, at the beginning of the multiplication, the inputs can store in the registers. Then, divide the operand into four parts using the Karatsuba-Ofman algorithm. Then generate the product using an integer multiplier. The partial products are generated using unsigned (4 number) and signed (6 number) multipliers. Ten multiplications are executed in parallel using the multiplier. In Algorithm-2, shown in figure 2, steps 1 to 12 explain the generation of partial product. eq. (4) represents that the output utilizes ten multipliers. The reduction of multipliers is achieved through the Karatsuba-Ofman algorithm. In Algorithm-2 steps, 13 to 19 shows that the result of the partial product is utilized as the inputs for the adders. The N/4 bits fast carry chain adders add the partial product shown in figure 3. Step 20 is the addition to create the final product. The splitting depth of the operand is two.

**Table II: Clock Cycle Four-Parts Splitting Multiplication**

| Task   | Clock Cycle |
|--------|-------------|
| 1st    | LR          |
| 2nd    | FA          |

**Table III: Four-Part Splitting Multiplication**

| Size | LUT | DSPs | SR | F  | CC | Period | Time | TP |
|------|-----|------|----|----|----|--------|------|----|
|      |     |      |    |    |    |        |      |    |
| 64   | 387 | 12   | 395| 155.42| 7 | 6.43  | 45.04 | 0.022|
| 128  | 1303| 36   | 967| 109.08| 7 | 9.17  | 64.17 | 0.016|
| 256  | 5246| 168  | 1927| 88.01 | 7 | 11.36 | 79.54 | 0.013|
| 512  | 21967| 693 | 3870| 63.93 | 7 | 15.64 | 109.49| 0.009|
| 1024 | 86748| 2394| 7731| 42.41 | 7 | 23.58 | 165.07| 0.006|

**Figure 3: Four Parts Splitting Multiplication**

**Virtex 7**

| Size | LUT | DSPs | SR | F  | CC | Period | Time | TP |
|------|-----|------|----|----|----|--------|------|----|
|      |     |      |    |    |    |        |      |    |
| 64   | 777 | 10   | 351| 297.49| 7 | 3.36  | 23.53 | 0.042|
| 128  | 1877| 40   | 1686| 155.42| 7 | 6.43  | 45.04 | 0.022|
| 256  | 5669| 120  | 3950| 109.08| 7 | 9.17  | 64.17 | 0.016|
| 512  | 21463| 560 | 7854| 88.01 | 7 | 11.36 | 79.54 | 0.013|
Table II shows the clock cycle for the four-part splitting multipliers. The complete product takes the seven-clock cycles. In figure 3, the first operation is to load the operand into the input register. In table II, the LR shows the Load Register which required a one-clock cycle. The second operation in table II is the calculation of the partial products, which is Partial Product Multiplication (PPM), which utilizes a one-clock cycle. The third operation in figure 3 is Partial Product Addition (PPA), which adds the partial product in the four-clock cycle. The PPA is the final stage. Final Addition (FA) of the product, which adds the PPA in the one-clock cycle. The whole multiplication utilized the seven-clock cycle for full product generation. The results for the implementation are shown in table III.

D. Eight-Parts Splitting Multiplier

In the eight-parts splitting technique, the depth is three. It shows that each input part is divided into two parts further. The prime benefit of eight-part splitting is to optimize the hardware resources further. The four-part splitting multiplier utilized the basic multiplier of DSP block in Virtex-6 and Virtex-7.

In the eight-part splitting algorithm, the operands are divided into eight equal parts. At the beginning of the multiplication, the inputs are stored in the registers. During the next step, divide the operand into eight-part using the splitting algorithm. Now after that generate the partial products using an integer multiplier.

Sixty-four unsigned multipliers of N/8-bits are required to generate the sixty-four partial products. In eq. (5), the output utilizes sixty-four multipliers. The splitting depth of the operand is three.

Table IV depicts the clock cycle instants for the four-part splitting multipliers. The complete product takes the two-clock cycles. The first operation is to load the operand into the input register. In table IV, the LR shows the Load Register which required a one-clock cycle. The second operation is the Final Addition (FA) of the product, which adds the PPA in one clock cycle. The whole multiplication utilized the two-clock cycle for the full product. Table 4, shows the implementation results for the eight-parts splitting multiplier.

| Size | LUT | DSPs | SR | F | CC | Period | Time | TP | GOPS |
|------|-----|------|----|---|----|--------|------|----|-----|
| 64   | 255 | 16   | 330 | 55.424 | 2 | 6.43   | 12.87 | 0.078 |
| 128  | 1139 | 48   | 898 | 109.077 | 2 | 9.17   | 18.34 | 0.055 |
| 256  | 5135 | 224  | 1794 | 88.009 | 2 | 11.36  | 22.72 | 0.044 |
| 512  | 19619 | 924  | 3608 | 63.93 | 2 | 15.64  | 31.28 | 0.032 |
| 1024 | 67403 | 3192 | 7214 | 42.405 | 2 | 23.58  | 47.16 | 0.021 |

Table V: School-Book Algorithm Two, Four, and Eight-Part Splitting Multiplier

VI. MONTGOMERY MULTIPLIER ARCHITECTURE

The architecture of MMM is demonstrated in Algorithm-3 as shown in figure 4. In this algorithm, there are 3 n-bit Integer multipliers are required. The general efficiency of the Montgomery multiplier depends on the Integer Multiplier. In this paper, we present an efficient MMM that is implemented on modern FPGA devices. The proposed architecture of the Montgomery multiplier is shown in figure 5. This architecture consists of an n-bit Integer Multiplier. The intermediate multiplication results are held in a 2n-bit register. This result in the register is utilized in the next steps. All three multiplications are performed in series. The clock cycle in the Karatsuba algorithm and two-clock cycle in the School-Book algorithm to calculate the multiplication of the operands and 2n-bit product result stored in the register, which is
represented by ‘W’ in Table VI. The result of the product written in the register file utilizes the one-clock cycle.

Algorithm 3: Montgomery Multiplier

Input: $A, B, M, n = \log_2 M, R = 2^n$

\[ M_1 = -M^{-1} \mod R \]

Result: Output $= A \times B \times R^{-1} \mod M$

\[ D \leftarrow A \times B \]

\[ E \leftarrow D \times M_1 \mod R \]

Output $= (D + E \times M) / R$

If Output $> M$ then return Output $- M$

Else return Output

Figure 4: Montgomery Multiplier

The outcome of the first integer multiplication is stored in the register. The stored result than got added to the third multiplication result. The final step is the reduction that utilizes to compute the Montgomery multiplication.

In table VI the proposed architecture performs the series of operations for executing the MMM. In the first clock cycle, the operands are loaded in Register A and B which is represented by Load Register (LR). The first integer multiplication utilizes the input operands A and B. When the multiplier gets the operands A and B, the multiplication operation gets started.

Table VI: Clock Cycle Wise Execution

| KA Clock Cycle | Operations | SB Clock Cycle |
|----------------|------------|----------------|
| 1<sup>n</sup> | LR | 1<sup>n</sup> |
| 2<sup>n</sup>-8<sup>n</sup> | IR | 2<sup>n</sup>-8<sup>n</sup> |
| 9<sup>n</sup> | WR | 4<sup>n</sup> |
| 10<sup>n</sup> | L | 5<sup>n</sup> |
| 11<sup>n</sup>-17<sup>n</sup> | IR | 6<sup>n</sup>-7<sup>n</sup> |
| 18<sup>n</sup> | WR | 8<sup>n</sup> |
| 19<sup>n</sup> | L | 9<sup>n</sup> |
| 20<sup>n</sup>-26<sup>n</sup> | IR | 10<sup>n</sup>-11<sup>n</sup> |
| 27<sup>n</sup> | A | 12<sup>n</sup> |
| 28<sup>n</sup> | C | 13<sup>n</sup> |
| 29<sup>n</sup> | S | 14<sup>n</sup> |

During the operation of Load Register (LR), new operands are loaded in the input register A and B. The second multiplication utilizes the modulus of the first multiplication output and M1 as an operand. The second multiplication also utilizes the seven clock cycles in the Karatsuba algorithm and the two clock cycles in the School-Book algorithm to compute the product and the one-clock cycle required to store the result in the register file. The three-multiplication required twenty-six clock cycles in series in the Karatsuba algorithm and eleven clock cycles in the School-Book algorithm. In the last step, three more clock cycles are required for the addition of the products of the three multiplications as per the Algorithm-3, which is depicted in figure 4. Comparison and subtraction operation computes if required. In this way, MMM architecture required twenty-nine clock cycles in Karatsuba Algorithm and fourteen clock cycles in the School-Book algorithm to compute the complete result.

Figure 5: Montgomery Modular Multiplier with Four Parts Splitting Multiplication

VII. IMPLEMENTATION RESULTS

The proposed architectures are made flexible to be implemented in any FPGA family. The length of operands meets the length of the fundamental multipliers of the FPGA devices.

The proposed architectures of two-part splitting, four-part splitting, and eight-part splitting are calculated for five common operands i.e. bit sizes 64, 128, 256, 512, and 1024. The proposed architectures of MMM have been designed in Verilog hardware description language and synthesis has been done in Xilinx ISE Design Suite 14.1 on different devices (Virtex-6, Virtex-7). Table VII demonstrates the implementation results for two-parts, four-parts, and eight-parts splitting MMM in an FPGA device. Table VII shows that the eight-part splitting architectures utilize fewer DSP blocks for the same operand length compared to two-part splitting and four-part splitting.

Table VII: Montgomery Multiplier with Two-Part and Four-Part Splitting Multiplier

| Size | LUT | DSPs | SR | F | CC | Period | Time | TP |
|------|-----|------|----|---|----|--------|------|----|
| bits | ns  | ns   | GOPS |
| MM with KA Two-Part Splitting Technique | 64 | 810 | 12 | 912 | 109.31 | 29 | 9.15 | 265.30 | 0.004 |
| 128 | 2135 | 36 | 1996 | 79.05 | 29 | 12.65 | 366.84 | 0.003 |
| 256 | 6680 | 168 | 3980 | 59.87 | 29 | 16.70 | 484.36 | 0.002 |
| 512 | 24782 | 693 | 7971 | 43.70 | 29 | 22.89 | 663.69 | 0.002 |
| MM with KA Four-Part Splitting Technique | 64 | 1200 | 10 | 868 | 277.66 | 29 | 3.60 | 104.44 | 0.010 |
| 128 | 2709 | 40 | 2715 | 109.17 | 29 | 9.16 | 265.65 | 0.004 |
| 256 | 7103 | 120 | 6003 | 78.99 | 29 | 12.66 | 367.16 | 0.003 |
| 512 | 24278 | 560 | 11095 | 53.66 | 29 | 18.64 | 540.43 | 0.002 |
| MM with KA Two-Part Splitting Technique | 64 | 810 | 12 | 912 | 123.22 | 29 | 8.12 | 235.36 | 0.004 |
| 128 | 2135 | 36 | 1996 | 88.12 | 29 | 11.35 | 329.11 | 0.003 |
| 256 | 6680 | 168 | 3980 | 66.23 | 29 | 15.10 | 437.85 | 0.002 |
| 512 | 24782 | 693 | 7971 | 48.64 | 29 | 20.56 | 596.20 | 0.002 |
| 1024 | 92328 | 2394 | 16162 | 33.84 | 29 | 29.55 | 856.87 | 0.001 |
Another concept was being presented to implement the architecture is observed until the specified size of the multiplier is cascading unsigned m bits utilization of the carry chain of 64 bit and multiplication of nine clock cycles to generate the result. The implemented results show that the time increased in the eight-part splitting method. Table VIII shows that the performance comparison with other design architectures on the same platform. The proposed design of 256-bit MMM architectures runs at frequency 95.87 MHz and utilized fourteen clock cycles to compute the result. The proposed design consumes 192 DSP blocks. The other proposed design for 256-bit architectures runs at frequency 78.985 MHz and utilized twenty-nine clock cycles to generate the result. The proposed design consumes 120 DSP blocks. Author Mondal and his associate authors in 2012 utilized the School-Book architectures to compute the MMM. It put up the concept to consume 16 64x64 bit soft cores architecture operated at 102 MHz frequency. Their resources regarding hardware architecture are consumed double as compare to proposed design. 29% less frequency is achieved here [25].

A concept for the multiplier of BN curves was presented by authors in 2013. They provided the special prime number for implementations of BN curves and utilized the non-standard division for adjustment of the FPGA Digital Signal Processor (DSP) block [26]. This architecture achieves 208 MHz frequency. However, they are only flexible with BN curves. This is the main drawback of this architecture. They utilized 50% more DSP block as compared to the proposed architecture. Author Javed and associates in 2014 elaborated the concept with the addition of 512-bit and multiplication of 256-bit utilization of the carry chain of 64-bit with the soft-core multiplier. They utilized the School-Book multiplier in Montgomery modular multiplier and succeed to achieve 188 MHz frequency. They utilized 200% more DSP blocks compared to the proposed architecture [27].

A group of researchers advanced the concept of an implementation scheme by using IP cores of the FPGA with the addition of 512-bit and 256-bit multiplication to achieve 50% better results as compared to a standard implementation. They improved the low frequency and high latency in this architecture. In this architecture, the authors have achieved a 40 MHz frequency, which is too low for high-speed applications. The drawback of this architecture is observed in the School-Book multiplier in MMM, which utilized too much area [28].

Another concept was being presented to implement the hardware processor for ECC. Full block MMM with the 256-bit integer multiplier (IM) is intended to 16-bit cascading unsigned multipliers and this method is sustained until the specified size of the multiplier is

| Design | Device | Size | Area | f | CC | Perio d | Time | TP |
|--------|--------|------|------|---|----|---------|------|----|
| Proposed KAM | | | | | | | | |
| Proposed KA Four-Part Splitting Montgomery Multiplier | | | | | | | | |
| Proposed SB Four-Part Splitting Montgomery Multiplier | | | | | | | | |
| Proposed SB Four-Part Splitting Technique | | | | | | | | |
| Proposed SB Four-Part Splitting Technique | | | | | | | | |
achieved. Fast carry look-ahead adders are required for the addition of modular multiplication. The drawback of this architecture in terms of time is the long duration for synthesis comparatively to our purposed architecture [29]. Another design concept was put forward by researchers when FPGA working of high frequency and routing delay is increased in large multipliers. Decreasing the routing delay is achieved by dividing the operands into small parts and by using the deep pipeline stages. The number of pipeline stages and time for the Montgomery modular multiplier is not mentioned in this paper. The drawback of this architecture is utilized more than fifty clock cycles [30]. Our purposed architecture utilized only twenty-nine clock cycles for one Montgomery Modular Multiplier. The design concept of the Karatsuba algorithm by dividing the operand into two chunks and use in the MMM algorithm with higher radix was introduced. This architecture uses the dedicated blocks of the multiplier. The drawback of this architecture is increasing the space of the hardware [31].

In another design concept, the Karatsuba algorithm was divided into 4 levels to use in MMM. Using the splitting method operand is divided into two parts. The divided part is again divided into two other parts in the reappearance style until the divided parts are length matches with the DSP blocks of an FPGA. This design utilized the LUTs on FPGA rather than the dedicated multipliers [32]. They utilized the same number of clock cycles as in our purposed architecture. The results are better in terms of time. Our purposed architecture is better in terms of utilization of hardware resources and time delay.

We also discussed the result of MMM with bit-wise implementation. In modern FPGA dedicated multipliers are not used in bitwise implementations, it uses only standard FPGA logic. The utilization of dedicated multipliers on FPGA is faster than the standard design-based FPGA. The purposed architecture consumes the 256-bit interleaved modular multiplier. In this architecture, they achieve a 96 MHz frequency [33]. The results of our purposed architecture are 4 times better in terms of time for MMM and throughput.

A design concept of LUT used on the hardware implementation rather than the FPGA dedicated multipliers was presented. Radix-4 is based on a modular multiplier with the serial interleaved [34]. In Table VIII, it is shown that our purposed architecture is 4 times better as compared to the design of Author Javeed who presented the similar design concept of Parallel interleaved modular multiplier implementation of hardware architecture. According to the architecture, an operand is working on four parallel processing elements to complete the dedicated task according to the algorithm [35]. The results of our purposed architecture are 2.5 times better in terms of time and throughput. Figure 6 shows the performance of several modular multipliers on the Virtex-6 FPGA platform. The proposed design has a higher throughput rate and comparable area and maximum frequencies.

![Performance Comparison](image)

**Figure 6: Performance Comparison of Several Multipliers**

VIII. CONCLUSION

In the hardware implementation of public-key cryptographic algorithms, Montgomery Modular Multiplier plays a vital role. This paper provides a full-word implementation of MMM which enhances the execution speed of Elliptic-Curve Cryptography (ECC) and RSA cryptographic algorithms on hardware. We have utilized the Karatsuba–Ofman and School-Book algorithm to calculate the 64–1024 bits. Multiplications are done by using Xilinx FPGA devices. In this work, we exploit the efficiency of the Karatsuba-Ofman and School-Book techniques to divide the operands into different sizes according to Xilinx FPGA devices. The proposed design is evaluated on computational time, area consumption, throughput and it will significantly surpass the state of the art.

**Acknowledgment**

First of all, we are thankful to almighty Allah, who gives us enough strength to fulfill this research work.

**Authors Contributions**

All the authors contributed equally to this manuscript.

**Conflict of Interest**

There is no conflict of interest between all the authors.

**Data Availability Statement**

All the data regarding this research is available in this article.

**Funding**

This research received no funding.

**References**

[1] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644-654.

[2] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.

[3] Koblitz, N. (1987). Elliptic curve cryptosystems. Mathematics of computation, 48(177), 203-209.
Khalid Javed et al.

[4] Khan, S., Javeed, K., & Shah, Y. A. (2018). High-speed FPGA implementation of full-word multiplier for ECC applications. Microprocessors and Microsystems, 62, 91-101.

[5] San, I., & At, N. (2014). Improving the computational efficiency of modular operations for embedded systems. Journal of Systems Architecture, 60(5), 440-451.

[6] Mondal, A., Ghosh, S., Das, A., & Chowdhury, D. R. (2012). Efficient FPGA implementation of Montgomery multiplier using DSP blocks. In Progress in VLSI Design and Test (pp. 370-372). Springer, Berlin, Heidelberg.

[7] Brinci, R., Khmiri, W., Mbarek, M., Rabab, A. B., Bouallegue, A., & Chekir, F. (2015). Efficient Multiplier for pairings over Barreto-Naehrig Curves on Virtex-6 FPGA. IACR Cryptol. ePrint Arch., 2015, 5.

[8] Kuang, S. R., Wu, K. Y., & Lu, R. Y. (2015). Low-cost high-performance VLSI architecture for Montgomery modular multiplication. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 24(2), 434-443.

[9] Rezai, A., & Keshavari, P. (2014). High-throughput modular multiplication and exponentiation algorithms using multibit-scan-multibit-shift technique. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 23(9), 1710-1719.

[10] Javeed, K., & Wang, X. (2014, July). Efficient Montgomery multiplier for pairing and elliptic curve-based cryptography. In 2014 9th International Symposium on Communication Systems, Networks & Digital Sign (CSNDSP) (pp. 255-260). IEEE.

[11] Yang, Y., Wu, C., Li, Z., & Yang, J. (2016). Efficient FPGA implementation of modular multiplication based on the Montgomery algorithm. Microprocessors and Microsystems, 47, 209-215.

[12] McIvor, C. J., McLoone, M., & McCanny, J. V. (2006). Hardware Elliptic Curve Cryptographic Processor Over nm GF (p). IEEE Transactions on Circuits and Systems I: Regular Papers, 53(9), 1946-1957.

[13] Morales-Sandoval, M., & Diaz-Perez, A. (2016). Scalable GF (p) Montgomery multiplier based on a digit-digit computation approach. IET Computers & Digital Techniques, 10(3), 102-109.

[14] Gong, Y., & Li, S. (2010, March). High-throughput FPGA implementation of 256-bit Montgomery modular multiplier. In 2010 Second International Workshop on Education Technology and Computer Science (Vol. 3, pp. 173-176). IEEE.

[15] Ghosh, S., Verbaaswede, I., & Roychowdhury, D. (2012, May). Core-based architecture to speed up optimal ate pairing on the FPGA platform. In International Conference on Pairing-Based Cryptography (pp. 141-159). Springer, Berlin, Heidelberg.

[16] Chow, G. C., Eguro, K., Luk, W., & Leong, P. (2010, August). A Karatsuba-based Montgomery multiplier. In 2010 International Conference on Field Programmable Logic and Applications (pp. 434-437). IEEE.

[17] San, I., & At, N. (2014). Improving the computational efficiency of modular operations for embedded systems. Journal of Systems Architecture, 60(5), 440-451.

[18] Yan, X., Wu, G., Wu, D., Zheng, F., & Xie, X. (2013, December). An implementation of Montgomery modular multiplication on FPGAs. In 2013 International Conference on Information Science and Cloud Computing (pp. 32-38). IEEE.

[19] Javeed, K., Wang, X., & Scott, M. (2017). High-performance hardware support for elliptic curve cryptography over a general prime field. Microprocessors and Microsystems, 51, 331-342.

[20] Javeed, K., & Wang, X. (2017). Low latency flexible FPGA implementation of point multiplication on elliptic curves over GF (p). International Journal of Circuit Theory and Applications, 45(2), 214-228.

[21] Ors, S. B., Batina, L., Preneel, B., & Vandewalle, J. (2003, June). Hardware implementation of an elliptic curve processor over GF (p). In Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors. ASAP 2003 (pp. 433-443). IEEE.

[22] Javeed, K., Wang, X., & Scott, M. (2015, September). Serial and parallel interleaved modular multipliers on the FPGA platform. In 2015 25th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-4). IEEE.

[23] Karatsuba, A. A., & Ofman, Y. P. (1962). Multiplication of many digital numbers by automatic computers. In Doklady Akademii Nauk (Vol. 145, No. 2, pp. 293-294). Russian Academy of Sciences.