Algorithm-Hardware Co-design of Ultra-high Radix based High Throughput Modular Multiplier

Hao Xiao\(^1\), Yuxuan Liu\(^1\), Zhenmin Li\(^{1, a}\), and Guangzhu Liu\(^1\)

Abstract This paper presents an algorithm-hardware co-design of ultra-high radix modular multiplier for high throughput modular multiplication. First, to speed up the modular multiplication, we exploit an ultra-high radix interleaved modular multiplication algorithm with a novel segmented reduction method, which reduces the number of iterations and pre-computations. Then, to further improve the throughput of the modular multiplication, we design a highly parallel modular multiplier architecture. Finally, we implement and verify the modular multiplier using the Xilinx Virtex-7 FPGA. Experimental results show it can perform a 256-bit modular multiplication in 0.56 µs with the throughput rate of up to 4999.7 Mbps.

Key words: Modular multiplication, high throughput, ultra-high radix

Classification: Integrated circuits (logic)

1. Introduction

Modular multiplication plays an important role in public-key cryptographic algorithms, such as RSA [1] and Elliptic Curve Cryptography (ECC) [2, 3]. Both RSA and ECC performs lots of modular multiplications, which are complex and time-consuming. Therefore, the design and speed-up of modular multiplications are significant to implement high-performance public-key cryptosystems.

According to the different methods of reduction, the modular multiplication can be divided into three categories including pseudo Mersenne primes, Montgomery [11] and interleaved modular multiplication [12]. In [13, 14, 15, 16, 17], the pseudo Mersenne primes algorithm is adopted, where the reduction with the product of two input integers can be performed using a few shifts and additions. In [14] and [15], Toom-Cook and Karatsuba algorithms are further adopted to reduce the complexity of the multiplication, and thus, improve the performance and throughput for a single modular multiplication. However, this algorithm can only deal with the modular multiplication of specific numbers (e.g. NIST recommended primes), which lacks flexibility and is not applied in generic scenarios. The Montgomery modular multiplication algorithm is widely used in large integer modular multiplication [18, 19, 20, 21]. It calculates

\[ c = ab \mod M \]

where \( a, b, c \) and \( r \) are all \( n \) bits number, and \( r \) is an integer power of 2, so reduction can be realized by shift and subtraction. However, this approach still requires further processing on the operands or the multiplication results for removing the \( r^{-1} \) factor of the output product. The idea of the interleaved modular multiplication algorithm [12] is to multiply the first operand with the second operand bit-wise and accumulate the product to the intermediate result iteratively. And the intermediate result is reduced with respect to the modulus. Compared with the Montgomery modular multiplication, this algorithm does not need the non-trivial processing for removing the \( r^{-1} \) factor, which facilitates the hardware implementation. Therefore, this paper focuses on the design of high-throughput modular multipliers using the interleaved modular multiplication algorithm.

In the interleaved modular multiplication algorithm, the size of radix is an essential parameter, where higher radix may reduce the number of iterations, and thus, improve the performance and throughput for a single modular multiplication. The authors in [22, 23] adopt radix-2 interleaved modular multiplication, which reuses the reduction circuit in each iteration for saving logic resources. However, for a \( n \)-bit modular multiplication, this approach needs \( n \) iterations, resulting in a huge number of computational cycles. Leveraging the pre-computation method proposed in [24], radix-4 interleaved modular multiplier architectures are implemented [25, 26, 27], which reduces the number of iterations by half. Although higher radix modular multiplier requires fewer iterations, it causes the pre-computation to increase exponentially. Moreover, the pre-computation needs to be carried out as long as the operand \( A \) or modulus \( M \) updates. When modular multiplications are calculated simultaneously, there are many pre-computations need to be updated, which causes an increased workload. Thus, to alleviate the workload of pre-computation, [28] and [29] carry out the modular multiplication by cascading pre-computed look up tables (LUTs) to gradually reduce the bit width of the product of two input integers. Compared with [25, 26, 27], the pre-computation of this method only depends on the modulus \( M \), and thus, it is beneficial to carry out multiple modular multiplications in scenarios with constant modulus. However, there is a crossover part caused by the carry of the addition in the process of gradually calculating the reduction, which requires
additional pre-computation. Therefore, to address the problems above, this paper proposes an algorithm-hardware co-design of ultra-high radix based high throughput modular multiplication. First, to speed up the modular multiplication, an ultra-high radix interleaved modular multiplication algorithm is proposed by taking different reduction methods for the high bits and the low bits of the intermediate result respectively. Moreover, to further improve the throughput of the modular multiplication, a highly parallel modular multiplier architecture is designed, which can deal with multiple modular multiplication operations simultaneously. Additionally, to reduce the storage resources consumed by pre-computation, a segmented reduction method is proposed, which can reduce the storage requirement caused by the carry of the addition in the sliding window (SW) method [28]. Finally, the proposed design is implemented and verified using the Xilinx Virtex-7 FPGA. Experimental results show that, based on a radix-224 design, the throughput of the 256-bit modular multiplication can achieve up to 4999.7 Mbps.

In the rest of the paper, Section 2 gives a brief introduction of the interleaved modular multiplication algorithm. Section 3 demonstrates the proposed hardware architecture for the modular multiplier. Implementation results and conclusions are given in section 4 and 5, respectively.

2. Algorithm

2.1 Radix-2 Interleaved Modular Multiplication

Let $A$, $B$ and $M$ are three $n$-bit integers, and $A, B \in [0, M - 1]$. $A$ is represented in radix $r_A$ as $A = \sum_{i=0}^{n_{omega_A}} A_i \times r_A^i$, where $r_A = 2^{\omega_A}$, and $n_{omega_A} = \lfloor n/\omega_A \rfloor$. Here, $\omega_A$ is the digit length, and $A_i$ is the $i$th digit where $A_i \in [0, r_A - 1]$ and $n_{omega_A}$ is the number of digits of $A$ in radix $r_A$ representation. The $i$th digit $A_i$ can be calculated with Eq. (1).

\[
A_i = (A >> (\omega_A \times i)) \mod r_A. \quad (1)
\]

Integer $A$ can be represented by Eq. (2). Then, the product of $A$ and $B$ could be represented by Eq. (3). Thus, a modular multiplication can be calculated using Eq. (4).

\[
A = \sum_{i=0}^{n_{omega_A}-1} (A_i \times r_A^i). \quad (2)
\]

\[
C = A \times B = \sum_{i=0}^{n_{omega_A}-1} (A_i \times B \times r_A^i). \quad (3)
\]

\[
C \equiv A \times B \mod M
\]

\[
\equiv \sum_{i=0}^{n_{omega_A}-1} (A_i \times B \times r_A^i) \mod M \quad (4)
\]

\[
\equiv \sum_{i=0}^{n_{omega_A}-1} ((A_i \times B \times r_A^i) \mod M) \mod M.
\]

Algorithm 1 Classical Interleaved Modular Multiplication.

Input: $A$, $B$, $M$, where $A = \sum_{i=0}^{n-1} A_i \times 2^i$ and $A, B \in [0, M - 1]$

Output: $C = A \times B \mod M$

1: $C \leftarrow 0$
2: for $A_i$ from $A_{n-1}$ to $A_0$ do
3: \hspace{1em} $C' \leftarrow 2 \times C$
4: \hspace{1em} $P \leftarrow C' + A_i \times B;$
5: \hspace{1em} if $P \geq 2M$ then
6: \hspace{2em} $C \leftarrow P - 2M$
7: \hspace{1em} else if $P \geq M$ then
8: \hspace{2em} $C \leftarrow P - M$
9: \hspace{1em} else
10: \hspace{2em} $C \leftarrow P$
11: end if
12: end for
13: return $C$

The idea of the interleaved modular multiplication is to allow multiplication and reduction arithmetic to overlap, and then, reduce the sum of the multiplication and reduction results within the range $[0, M-1]$. Algorithm 1 demonstrates the classical radix-2 interleaved modular multiplication [12], where $r_A = 2$. It starts scanning the multiplier $A$ from the most significant bit to the least significant bit. At the beginning of each iteration, the intermediate result $C$ is doubled to perform the iterative addition of the successive partial products at Line 4 and 5. Because the sum $P$ with a width of $n + 2$ satisfies $0 \leq P < 3M$, comparisons and subtractions (Line 5 to 10) are required to keep the intermediate result $C \in [0, M - 1]$. And after $n$ iterations, $C$ becomes the final result of the modular multiplication.

2.2 Ultra-high Radix Interleaved Modular Multiplication

Algorithm 2 shows the proposed high radix interleaved modular multiplication. In each iteration, the width of the partial product $D$ (Line 3) and the sum $P$ (Line 4) is $r_A + n$ bit and $r_A + n + 1$ bit, respectively. The intermediate result $P$ is divide into high part ($P_H$) and low part ($P_L$), which are shown in Fig. 1. The $P_H$ part, whose width is $d = r_A + 1$ bit, is further represented by three parts $P_1$, $P_2$, and $P_3$ using Eq. (5), where $l = \lfloor d/3 \rfloor$. Assume a fixed modulus is used for modular multiplication. $P_1$ and $P_2$ with a width of $l$ bits satisfy Eq. (6) and (7) respectively. $P_3$ with a width of $l + 1$ bits satisfies Eq. (8). For each $P_1$, $P_2$, and $P_3$, their reduction results are unique, which can be calculated in advance and stored in the LUT. Using this approach, the reduction results of $P_1$, $P_2$, and $P_3$ can be obtained by the Step 6 at the same time. The number of bytes $q$ of storage resources consumed by LUT can be calculated by Eq. (9), where $k$ is the bit width of the input number. The reduction results of remaining $P_L$ with a width of $n$ can be calculated using Eq. (10). For Step 12, the sum of the four reduction results is calculated, which satisfies the relationship $0 \leq F < 4M$. The final step of Algorithm 2 takes three comparisons and subtractions to keep
Algorithm 2 Proposed Interleaved Modular Multiplication.

Input: $A$, $B$, $M$, where $A = \sum_{i=0}^{n} A_i \times r_a^i$ and $A, B \in [0, M - 1]$
Output: $C = A \times B \mod M$
1: $C \leftarrow 0$
2: for $i$ from $n_{\omega_A} - 1$ to 0 do
3: $D \leftarrow A_i \times B$
4: $P \leftarrow C \times r_a + D$
5: Split $P$ into $P_L, P_1, P_2, P_3$
6: $P_1' \leftarrow \text{LUT}(P_1)$; $P_2' \leftarrow \text{LUT}(P_2)$; $P_3' \leftarrow \text{LUT}(P_3)$;
7: if $P_L \geq M$ then
8: $P_L' \leftarrow P_L - M$
9: else
10: $P_L' \leftarrow P_L$
11: end if
12: $P' \leftarrow (P_L' + P_1') + (P_2' + P_3')$
13: if $P' \geq 3M$ then
14: $C \leftarrow P' - 3M$
15: else if $P' \geq 2M$ then
16: $C \leftarrow P' - 2M$
17: else if $P \geq M$ then
18: $C \leftarrow P' - M$
19: else
20: $C \leftarrow P'$
21: end if
22: end for
23: return $C$

intermediate result $C \in [0, M - 1]$. And after $n_{\omega_A}$ iterations, $C$ becomes the final result of the modular multiplication.

Fig. 1: Format of $P$.

$$P_H \equiv (P_3 \cdot 2^{n+2l} + P_2 \cdot 2^{n+l} + P_1 \cdot 2^n) \mod M$$
$$\equiv (P_3 \cdot 2^{n+2l} \mod M + P_2 \cdot 2^{n+l} \mod M + P_1 \cdot 2^n \mod M) \mod M$$

$$2^n \leq P_1 < 2^r a^t + l.$$  \hspace{1cm} (6)

$$2^{n+l} \leq P_2 < 2^{r a^t + 2l}.$$  \hspace{1cm} (7)

$$2^{n+2l} \leq P_3 < 2^{r a^t + d}.$$  \hspace{1cm} (8)

$$q = 2^k \times n \div 8.$$  \hspace{1cm} (9)

$$P_L' = \begin{cases} P_L - M & \text{if } P_L \geq M \\ P_L & \text{else} \end{cases}.$$  \hspace{1cm} (10)

Taking 256-bit modular multiplication as an example (the operands of modular multiplication in this paper are all 256 bits). An ultra-high radix $r_a = 2^{24}$ is adopted in this paper, so that the reduction can be calculated only by $n_{\omega_A} = \lceil n / \omega_A \rceil = 11$ iterations. For each iteration, the bit width of $P_H$ is $\omega_A + 1 = 25$. The significant bits of $P_1, P_2$ are $l$, where $l = \lfloor \omega_A / 3 \rfloor = 8$. And the significant bit of $P_3$ is $l + 1 = 9$. The storage resources consumed by LUT for $P_1, P_2$ and $P_3$ are 8KB, 8KB and 16KB respectively, a total of 32KB.

3. Hardware Implementation

3.1 Architecture Overview

Fig. 2 shows the block diagram of the proposed hardware architecture, which can support multiple modular multiplication calculations simultaneously. It consists of a multiply and add (MA) array, a reduction unit, a feedback loop and three data buffers. After input, multiple pairs of $A$ and $B$ can be stored in buffer $A$ and buffer $B$ respectively, and they are inputted into the MA array in a pipelined manner. The MA array is composed of multiple MA units. For each MA unit, the port $a$ with the same width of the selected radix is cascaded together. Each $A_i$ are inputted to MA0 in a serial manner, and then, passed to the next MA through the port $a$. The port $b$ with a width of 16 bits in each MA unit is independent and can obtain data from buffer B at the same time. The output of MA unit is divided into two parts: 1) as the intermediate result of the partial product, it is inputted into the partial product buffer; 2) the carry part is inputted to the next level MA unit through the port $carry$. When the partial products of multiple pairs of $A_i$ and $B$ are calculated, these partial products will be read from the partial product buffer and added with the results feedback from the reduction unit (the result initially turns to 0 when the first feedback result is returned). And the sums are inputted into the reduction unit for next iteration. For $n$-bit operands, $q = \lceil n / 16 \rceil$ MA units and $n_{\omega_A} = \lceil n / \omega_A \rceil$ iterations are required.
3.2 Calculation of Partial Products

The partial product calculation of the architecture for processing two 256-bit modular multiplication is shown in Fig. 3 (a) and (b). Taking radix $r_A = 2^{24}$ for example, there are $q = \lceil n/16 \rceil = 16$ MA units. Thus, the data in buffers A and B are further processed. As shown in Fig. 3 (a), $x$ is the serial number of the operands. $A$ is divided into $\lceil n/\omega_A \rceil = 11$ parts, and each serial number is represented by $y$. Similarly, $B$ is divided into $\lceil n/16 \rceil = 16$ parts. As shown in Fig. 3 (b), at the beginning of each iteration, $A_{1,y}$ and $B_{1,0}$ are read from the buffers and inputted into MA0. In the next clock, MA1 gets $A_{1,y}$ from MA0 and reads $B_{1,1}$ from buffer B. At the same time, MA0 reads $A_{2,y}$ and $B_{2,0}$ again from the buffer A and B. The intermediate results of the partial products calculated by each MA unit are pipelined into the next level MA unit and the partial product buffer respectively.

3.3 Segment Reduction Method

As shown in Fig. 2, the reduction part consists of a reduction unit and a feedback loop, which works in a completely pipelined manner. At each iteration, the multiple partial products stored in the partial product buffer are read out and added them one by one to the results returned by the feedback loop, and then input the sums into the reduction unit for the next iteration.

Fig. 4 shows the hardware structure of the reduction unit using the proposed segmented reduction method in section 2.2. The reduction unit consists of four pipeline stage. In the first stage, the intermediate result $P$ is divided into four parts, where $P_3$, $P_2$ and $P_1$ are the high $d = \omega_A + 1$ bits of $P$, and $P_L$ is the lower $n$ bits of $P$. All reductions of $P_3$, $P_2$ and $P_1$ are calculated in advance and stored in the LUTs. The reduction of $P_L$ is calculated after a comparison and subtraction. The next two levels calculate the sum of the four parallel reduction results. In the last stage, the reduction of the sum is calculated after three comparisons and a subtraction to keep the result $C \in [0, M - 1]$. The critical path of the reduction part contains a series of addition and comparison which is split into multiple cycles to complete. For our 256-bit design, 2 clock cycles are required for iteration loop and reduction Stage 1-3, and 3 clock cycles for Stage 4, a total of 11 clock cycles. Thus, it can calculate 11 reductions in a pipelined manner. Although a deeper pipeline can accommodate more parallel calculations, it will increase the latency of each single calculation.

Table I illustrates the storage resources consumed by the pre-computation of 256-bit modular multiplication with radix $2^{24}$. According to Section 2.2, $P$ and $P_H$ in each iteration satisfy the relationship $0 \leq P < 2^{281}$, $0 \leq P_H < 2^{25}$ respectively. There are $2^{25}$ reduction results need to be pre-computed, which requires 512KB storage space. Through our experiments, the SW method [28] can reduce storage resources of the pre-computation to 48KB, which is equivalent to a reduction of 90.6%. In this method, it performs the reduction operation on the higher part of $P$ and adds the result to the rest of $P$ repeatedly. However, the carry of addition requires additional pre-computation. The proposed segmented reduction method solves this problem, so the storage resources of the pre-computation are further reduced to 32KB, which is equivalent to a reduction of 33.3%.

3.4 Timing of Pipelined Modular Multiplication

As illustrated above, the proposed architecture breaks each 256-bit modular multiplication into 11 iterations, during which 11 different 256-bit modular multiplications can be carried out in parallel for improving the throughput. Figure 5 details the timing for calculating 256-bit modular multiplications. During the 1st iteration, the MA array calculates the products of $A_{x,10}$ and $B_x$, and the resulting partial products ($D_{1,10}$, ..., $D_{11,10}$) are output from $t_1 + t_3$ (t3 is the output delay of each MA unit). Meanwhile, the reduction unit starts and outputs the reduction results ($C_{1,10}$, ..., $C_{11,10}$) from

| Table II: Breakdown of cycles for calculating a 256-bit modular multiplication |
|-------------------------------|---|---|---|---|---|
| Clock Cycles  | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ |
|----------------|------|------|------|------|------|
| Total          | 16   | 5    | 3    | 11   | 151  |
In this paper, the proposed ultra-high radix modular multiplier is implemented and verified on the Xilinx Virtex-7 FPGA using Vivado 2018.3. As illustrated in section 3, the proposed modular multiplier adopts radix $r_A = 2^4$ and supports up to 11 256-bit modular multiplications working in parallel. The experimental results show the latency for calculating one 256-bit modular multiplication is 0.56 μs and its throughput rate can reach up to 4999.7 Mbps.

Table III compares the performance of this design and various 256-bit interleaved modular multiplication implementations based on FPGA. To speed up the modular multiplication, both [23] and [30] use the radix-2 interleaved modular multiplication algorithm which consumes 257 clock cycles. However, thanks to the proposed ultra-high radix method, this design reduces the number of iterations from 256 to 11, leading to the computation latency reduced by 66.7% and 61.4% compared with [23] and [30] respectively. [27] designs two radix-4 interleaved modular multipliers through the pre-computation method [24]. As compared with [27],

thanks to the proposed segmented reduction method, our design does not need to update the pre-computation for a fixed modulus. In addition, due to the proposed pipelined parallel architecture, it can calculate up to 11 modular multiplications in parallel. Experimental results show its throughput rate can reach up to 4999.7 Mbps and its latency is reduced by approximately 56.9% and 27.3% than the two designs in [27] respectively. Table III further compares the efficiency of these implementations in terms of area time product (AT) and throughput per area (TR/A). Comparing with [23], our design requires $2.0 \times$ AT1 but offers $5.6 \times$ TR/A1. As compared with [30], although the AT2 is increased by $1.3 \times$, our design improves the TR/A2 by $3.7 \times$. Comparing with [27], our design improves the TR/A2 by $9.9 \times$ and $7.8 \times$, respectively, with a similar AT2 performance.

5. Conclusion

In this paper, an algorithm-hardware co-design of ultra-high radix modular multiplier is proposed for high throughput modular multiplication. In terms of the algorithm, we exploit an ultra-high radix interleaved modular multiplication algorithm with the proposed reduction method. Moreover, in terms of hardware, thanks to the proposed segmented reduction method, the pre-computation only depends on the modulus, and the parallel calculation of modular multiplication is realized through a pipelined hardware structure, which significantly increases the throughput. Finally, the proposed modular multiplier has been implemented and verified on FPGA. Experimental results confirm our modular multiplier
performs efficiently with low processing latency, high parallelism and fewer pre-computation requirements, making it a viable option for high throughputs modular multiplication.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China 61974039, 61834006, and the Aeronautical Science Foundation of China 2018ZCP4003.

References

[1] R. L. Rivest, et al.: “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM 21 (1978) 120 (DOI: 10.1145/359340.359342).
[2] K. Neal: “Elliptic curve cryptosystems,” Math. Comput. 48 (1987) 203 (DOI: 10.1090/S0025-5718-1987-0866109-5).
[3] V. S. Miller: “Use of elliptic curves in cryptography,” Lect. Notes Comput. Sci. (1985) 417 (DOI: 10.1007/3-540-39799-X_31).
[4] T. Numata, et al.: “Circuit simulation model for ultimately-scaled ballistic nanowire MOSFETs,” IEICE Electron. Express 10 (2013) 20120906 (DOI: 10.1587/elex.20120906).
[5] A. Rezai and P. Keshavarzi: “High-Throughput Modular Multiplication and Exponentiation Algorithms Using Multibit-Scan–Multibit-Shift Technique,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 23 (2015) 1710 (DOI: 10.1109/TVLSI.2014.2355854).
[6] N. Nedjah, et al.: “Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems,” IET Comput. Digital Tech. 6 (2012) 290 (DOI: 10.1049/iet-cdt.2011.0074).
[7] J. Wei, et al.: “A low-time-complexity and secure dual-field scalar multiplication based on co-Z protected NAF,” IEICE Electron. Express 11 (2014) 20140361 (DOI: 10.1587/elex.11.20140361).
[8] X. Han, et al.: “A high-performance elliptic curve cryptographic coprocessor with side channel analysis countermeasures for smart IC card,” IEICE Electron. Express 12 (2015) 20150470 (DOI: 0.1587/elex.12.20150470).
[9] T. F. Al-Somani: “Very efficient point multiplication on Koblitz curves,” IEICE Electron. Express 13 (2016) 20160044 (DOI: 10.1587/elex.13.20160044).
[10] M. Mehrabi, et al.: “Elliptic Curve Cryptography Point Multiplication Core for Hardware Security Module,” IEEE Trans. Comput. 69 (2020) 1707 (DOI: 10.1109/TC.2020.3013266).
[11] L. M. Peter: “Modular multiplication without trial division,” Math. Comput. 44 (1985) 519 (DOI: 10.1090/S0025-5718-1985-0777282-X).
[12] G. R. Blakely: “A Computer Algorithm for Calculating the Product AB modulo M,” IEEE Trans. Comput. C-32 (1983) 497 (DOI: 10.1109/TBC.1983.1676262).
[13] M. Knežević, et al.: “Low-Latency ECDSA Signature Verification—A Road Toward Safer Traffic,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 24 (2016) 3257 (DOI: 10.1109/TVLSI.2016.2557965).
[14] J. Ding, et al.: “High-Speed ECC Processor Over NIST Prime Fields Applied With Toom–Cook Multiplication,” IEEE Trans. Circuits Syst. Regul. Pap. 66 (2019) 1003 (DOI: 10.1109/TC.2018.2878598).
[15] J. Ding and S. Li: “A Reconfigurable High-Speed ECC Processor Over NIST Primes,” IEEE Int. Conf. Trust, Secur.
Priv. Comput. Commun. (2017) 1064 (DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.533).
[16] P. Choi, et al.: “Low-Complexity Elliptic Curve Cryptography Processor Based on Configurable Partial Modular Reduction Over NIST Prime Fields,” IEEE Trans. Circuits Syst. Express Briefs 65 (2018) 1703 (DOI: 10.1109/TCSII.2017.2756680).
[17] H. Marzouqi, et al.: “A High-Speed FPGA Implementation of an RSD-Based ECC Processor,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 24 (2016) 151 (DOI: 10.1109/TVLSI.2015.2391274).
[18] G. Gallin and A. Tisserand: “Generation of Finely-Pipelined GF(P) Multipliers for Flexible Curve Based Cryptography on FPGAs,” IEEE Trans. Comput. 68 (2019) 1612 (DOI: 10.1109/TC.2019.2920352).
[19] B. Koziel, et al.: “Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves,” IEEE Trans. Circuits Syst. Regul. Pap. 64 (2017) 86 (DOI: 10.1109/TC.2016.2611561).
[20] D. Mukhopadhyay and D. B. Roy: “Revisiting FPGA Implementation of Montgomery Multiplier in Redundant Number System for Efficient ECC Application in GF(p),” Int. Conf. Field Program. Log. Appl. (2018) 323 (DOI: 10.1109/FPL.2018.0061).
[21] D. B. Roy and D. Mukhopadhyay: “High-Speed Implementation of ECC Scalar Multiplication in GF(p) for Generic Montgomery Curves,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 27 (2019) 1587 (DOI: 10.1109/TVLSI.2019.2905899).
[22] S. Ghosh, et al.: “Petrel: Power and Timing Attack Resistant Elliptic Curve Scalar Multiplier Based on Programmable GF(p) Arithmetic Unit,” IEEE Trans. Circuits Syst. Regul. Pap. 58 (2011) 1798 (DOI: 10.1109/TCSII.2010.2103190).
[23] M. S. Hossain, et al.: “High-performance elliptic curve cryptography processor over NIST prime fields,” IET Comput. Digital Tech. 11 (2017) 33 (DOI: 10.1049/iet-cdt.2016.0033).
[24] V. Buminov and M. Schlimmler: “Area and time efficient modular multiplication of large integers,” Proc. Int. Conf. Appl. Spec. Syst. Arcitec. and Process. (2003) 400 (DOI: 10.1109/ASAP.2003.1212863).
[25] Z. Liu, et al.: “An Efficient and Flexible Hardware Implementation of the Dual-Field Elliptic Curve Cryptographic Processor,” IEEE Trans. Ind. Electron. 64 (2017) 2353 (DOI: 10.1109/TIE.2016.2625241).
[26] J. Lee, et al.: “Efficient Power-Analysis-Resistant Dual-Field Elliptic Curve Cryptographic Processor Using Heterogeneous Dual-Processing-Element Architecture,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 22 (2014) 49 (DOI: 10.1109/TVLSI.2013.2237930).
[27] K. Javed, et al.: “Serial and parallel interleaved modular multipliers on FPGA platform,” Int. Conf. Field Program. Log. Appl. (2015) 1 (DOI: 10.1109/FPL.2015.7293986).
[28] S. Sinha Roy, et al.: “FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data,” Proc. IEEE Int. Symp. High Perform. Comput. Archit. (2019) 387 (DOI: 10.1109/HPCA.2019.00052).
[29] E. Öztürk: “Design and Implementation of a Low-Latency Modular Multiplication Algorithm,” IEEE Trans. Circuits Syst. Regul. Pap. 67 (2020) 1902 (DOI: 10.1109/TC.2020.2966755).
[30] M. D. M. Islam, et al.: “Area-Time Efficient Hardware Implementation of Modular Multiplication for Elliptic Curve Cryptography,” IEEE Access 8 (2020) 73898 (DOI: 10.1109/ACCESS.2020.2988379).