A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over GF(p)

Yuan Ma[1,2](✉), Zongbin Liu[1], Wuqiong Pan[1,2], and Jiwu Jing[1]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
{yma,zbliu,wqpan,jing}@lois.cn

Abstract. Elliptic curve cryptography (ECC) is preferred for high-speed applications due to the lower computational complexity compared with other public-key cryptographic schemes. As the basic arithmetic, the modular multiplication is the most time-consuming operation in public-key cryptosystems. The existing high-radix Montgomery multipliers performed a single Montgomery multiplication either in approximately $2n$ clock cycles, or approximately n cycles but with a very low frequency, where n is the number of words. In this paper, we first design a novel Montgomery multiplier by combining a quotient pipelining Montgomery multiplication algorithm with a parallel array design. The parallel design with one-way carry propagation can determine the quotients in one clock cycle, thus one Montgomery multiplication can be completed in approximately n clock cycles. Meanwhile, by the quotient pipelining technique applied in digital signal processing (DSP) blocks, our multiplier works in a high frequency. We also implement an ECC processor for generic curves over GF(p) using the novel multiplier on FPGAs. To the best of our knowledge, our processor is the fastest among the existing ECC implementations over GF(p).

Keywords: FPGA · Montgomery multiplier · DSP · High-speed · ECC

1 Introduction

Elliptic curve cryptography has captured more and more attention since the introduction by Koblitz [8] and Miller [12] in 1985. Compared with RSA or discrete logarithm schemes over finite fields, ECC uses a much shorter key to achieve an equivalent level of security. Therefore, ECC processors are preferred for high-speed applications owing to the lower computational complexity and other nice properties such as less storage and power consumption. Hardware accelerators

This work is supported by National Natural Science Foundation of China grant 70890084/G021102 and 61003274, Strategy Pilot Project of Chinese Academy of Sciences sub-project XDA06010702, and National High Technology Research and Development Program of China (863 Program, No. 2013AA01A214 and 2012AA013104).

are the most appropriate solution for the high-performance implementations with acceptable resource and power consumption. Among them, field-programmable gate arrays (FPGAs) are well-suited for this application due to their reconfigurability and versatility.

Point multiplication dominates the overall performance of the elliptic curve cryptographic processors. Efficient implementations of point multiplication can be separated into three distinct layers [6]: the finite field arithmetic, the elliptic curve point addition and doubling and the scalar multiplication. The fundamental finite field arithmetic is the basis of all the others. Finite field arithmetic over GF(p) consists of modular multiplications, modular additions/subtractions and modular inversions. By choosing an alternative representation, called the projective representation, for the coordinates of the points, the time-consuming finite field inversions can be eliminated almost completely. This leaves the modular multiplication to be the most critical operation in ECC implementations over GF(p). One of the widely used algorithms for efficient modular multiplications is the Montgomery algorithm which was proposed by Peter L. Montgomery [16] in 1985.

Hardware implementations of the Montgomery algorithm have been studied for several decades. From the perspective of the radix, the Montgomery multiplication implementations can be divided into two categories: radix-2 based [7,21] and high-radix based [1,2,9,11,17,19,20,22,23]. Compared with the former one, the latter, which can significantly reduce the required clock cycles, are preferred for high-speed applications.

For high-radix Montgomery multiplication, the determination of quotients is critical for speeding up the modular multiplication. For simplifying the quotient calculation, Walter et al. [3,23] presented a method that shifted up of modulus and multiplicand, and proposed a systolic array architecture. Following the similar ideas, Orup presented an alternative to systolic architecture [18], to perform high-radix modular multiplication. He introduced a rewritten high-radix Montgomery algorithm with quotient pipelining and gave an example of a non-systolic (or parallel) architecture, but the design is characterized by low frequency due to global broadcast signals. In order to improve the frequency, DSP blocks widely dedicated in modern FPGAs have been employed for high-speed modular multiplications since Suzuki's work [19] was presented. However, as a summary, the existing high-radix Montgomery multipliers perform a single Montgomery multiplication for n-word multiplicand either in approximately $2n$ clock cycles, or approximately n cycles but with a low frequency.

To design a high-speed ECC processor, our primary goal is to propose a new Montgomery multiplication architecture which is able to simultaneously process one Montgomery multiplication within approximately n clock cycles and improve the working frequency to a high level.

Key Insights and Techniques. One key insight is that the parallel array architecture with one-way carry propagation can efficiently weaken the data dependency for calculating quotients, yielding that the quotients can be determined in a *single* clock cycle. Another key insight is that a high working frequency can be achieved by employing quotient pipelining inside DSP blocks. Based on these insights, our Montgomery multiplication design is centered on the novel techniques: combining the parallel array design and the quotient pipelining inside DSP blocks.

We also implement an ECC processor for generic curves over GF(p) using the novel multiplier on FPGAs. Due to the pipeline characteristic of the multiplier, we reschedule the operations in elliptic curve arithmetic by overlapping successive Montgomery multiplications to further reduce the number of operation cycles. Additionally, side-channel analysis (SCA) resistance is considered in our design. Experimental results indicate that our ECC processor can perform a 256-bit point multiplication in 0.38 ms at 291 MHz on Xilinx Virtex-5 FPGA.

Our Contributions. In summary, the main contributions of this work are as follows.

– We develop a novel architecture for Montgomery multiplication. As far as we know, it is the first Montgomery multiplier that combining the parallel array design and the quotient pipelining using DSP blocks.
– We design and implement our ECC processor on modern FPGAs using the novel Montgomery multiplier. To the best of our knowledge, our ECC processor is the fastest among the existing hardware implementations over GF(p).

Structure. The rest of this paper is organized as follows. Section 2 presents the related work for high-speed ECC implementations and high-radix Montgomery multiplications. Section 3 describes a processing method for pipelined implementation, and a high-speed architecture is proposed in Sect. 4. Section 5 gives implementation results and detailed comparisons with other designs. Section 6 concludes the paper.

2 Related Work

2.1 High-Speed ECC Implementations over GF(p)

Among the high-speed ECC hardware implementations over GF(p), the architectures in [5] and [4] are the fastest two. For a 256-bit point multiplication they reached latency of 0.49 ms and 0.68 ms in modern FPGAs Virtex-4 and Stratix II, respectively. The architectures in [5] are designed for NIST primes using fast reduction. By forcing the DSP blocks to run at their maximum frequency (487 MHz), the architectures reach a very low latency for one point multiplication. Nevertheless, due to the characteristics of dual clock and the complex control inside DSP blocks, the architecture can be only implemented in FPGA platforms. Furthermore, due to the restriction on primes, the application scenario of [5] is limited in NIST prime fields. For generic curves over GF(p), [4] combines residue number systems (RNS) and Montgomery reduction for ECC implementation. The design achieves 6-stage parallelism and high frequency with a large number of DSP blocks, resulting in the fastest ECC implementation for generic curves. In addition, the design in [4] is resistant to SCA.

As far as we know, the fastest ECC implementation based on Montgomery multiplication was presented in [11], which was much slower than the above two designs. The main reason is that the frequency is driven down to a low level although the number of cycles for a single multiplication is approximately n. In an earlier FPGA device Virtex-2 Pro, the latency for a 256-bit point multiplication is 2.27 ms without the SCA resistance, and degrades to 2.35 ms to resist SCA.

2.2 High-Radix Montgomery Multiplication

Up to now, for speeding up high-radix Montgomery multiplications, a wealth of methods have been proposed either to reduce the number of processing cycles or to shorten the critical path in the implementations.

The systolic array architecture seems to be the best solution for modular multiplications with very long integers. Eldridge and Walter performed a shift up of the multiplicand to speed up modular multiplication [3], and Walter designed a systolic array architecture with a throughput of one modular multiplication every clock cycle and a latency of $2n + 2$ cycles for n-word multiplicands [23]. Suzuki introduced a Montgomery multiplication architecture based on DSP48, which is a dedicated DSP unit in modern FPGAs [19]. In order to achieve scalability and high performance, complex control signals and dual clocks were involved in the design. However, the average number of processing cycles per multiplication are approximately $2n$ at least. In fact, this is a common barrier in the high-radix Montgomery algorithm implementations: the quotient is hard to generate in a single clock cycle. This barrier also exists in other systolic high-radix designs [9,22].

On the contrary, some non-systolic array architectures were proposed for speeding up the process of quotient determination, but the clock frequency is a concern. Orup introduced a rewritten high-radix Montgomery algorithm with quotient pipelining and gave an example of a non-systolic architecture [18]. Another high-speed parallel design was proposed by Mentens [11], where the multiplication result was written in a special carry-save form to shorten the long computational path. The approach was able to process a single n-word Montgomery multiplication in approximately n clock cycles. But the maximum frequency was reduced to a very low level, because too many arithmetic operations have to be completed within one clock cycle. Similar drawbacks in frequency can also be found in [2,17].

3 Processing Method

In this section, we propose a processing method for pipelined implementation by employing DSP blocks.

3.1 Pipelined Montgomery Algorithm

Montgomery multiplication [16] is a method for performing modular multiplication without the need to perform division by the modulus. A high-radix version of Montgomery's algorithm with quotient pipelining [18] is given as Algorithm 1. The algorithm provides a way to apply pipelining techniques in Montgomery modular multiplication to shorten the critical path.

Algorithm 1. Modular Multiplication with Quotient Pipelining [18]

```
Input:
    A modulus $M > 2$ with $\gcd(M, 2) = 1$ and positive integers $k$,$n$ such that
    $4\tilde{M} < 2^{kn} = R$, where $\tilde{M}$ is given by $\tilde{M} = (\bar{M} \bmod 2^{k(d+1)})M$ and integer $d \geq 0$
    is a delay parameter.
    Integer $R^{-1}$, where $2^{kn}R^{-1} \bmod M = 1$
    Integer $\bar{M}, M'$, where $(-M\bar{M}) \bmod 2^{k(d+1)} = 1$, $M' = (\tilde{M} + 1) \text{ div } 2^{k(d+1)}$
    Integer multiplicand $A$, where $0 \leq A \leq 2\tilde{M}$
    Integer multiplier $B = \sum_{i=0}^{n+d}(2^k)^i b_i$, where digit $b_i \in \{0, 1, \ldots, 2^k - 1\}$ for $0 \leq i <$
    $n$, $b_i = 0$ for $i \geq n$ and $0 \leq B \leq 2\tilde{M}$.
Output:
    Integer $S_{n+d+2}$ where $S_{n+d+2} \equiv ABR^{-1} \pmod{M}$ and $0 \leq S_{n+d+2} < 2\tilde{M}$
1: $S_0 = 0; q_{-d} = 0; q_{-d+1} = 0; \ldots; q_{-1} = 0;$
2: for $i = 0$ to $n + d - 1$ do
3:     $q_i = S_i \bmod 2^k;$
4:     $S_{i+1} = S_i \text{ div } 2^k + q_{i-d}M' + b_iA;$
5: end for
6: $S_{n+d+2} = 2^{kd}S_{n+d+1} + \sum_{j=0}^{d+1} q_{n+j+1}2^{kj}$
```

The calculation of the right q_i is crucial for Montgomery multiplication, and it is the most time-consuming operation for hardware implementations. In order to improve the maximum frequency, a straightforward method is to divide the computational path into α stages for pipelining. The processing clock cycles, however, increase by a factor of α, since q_i is generated every α clock cycles. That is to say, the pipeline does not work due to data dependency. The main idea of Algorithm 1 is using the preset values $q_{-d} = 0, q_{-d+1} = 0, \ldots, q_{-1} = 0$ to replace q_1 to start the pipeline in the first d clock cycles. Then, in the $(d+1)^{th}$ cycle, the right value q_1 is generated and fed into the calculation of the next round. After that, one q_i is generated per clock cycle in pipelining mode. Compared to the traditional Montgomery algorithm, the cost of Algorithm 1 is a few extra iteration cycles, additional pre-processing and a wider range of the final result.

3.2 DSP Blocks in FPGAs

Dedicated multiplier units in older FPGAs have been adopted in the high-radix Montgomery multiplication implementations for years. In modern FPGAs, such as Xilinx Virtex-4 and later FPGAs, instead of traditional multiplier units, arithmetic hardcore accelerators - DSP blocks have been embedded. DSP blocks can

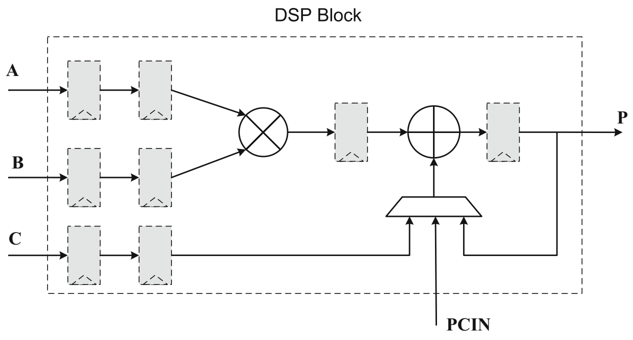

Fig. 1. Generic structure of DSP blocks in modern FPGAs

be programmed to perform multiplication, addition, and subtraction of integers in a more flexible and advanced fashion.

Figure 1 shows the generic DSP block structure in modern FPGAs. By using different data paths, DSP blocks can operate on external inputs A, B, C as well as on feedback values from P or results $PCIN$ from a neighboring DSP block. Notice that all the registers, labeled in gray in Figure 1, can be added or bypassed to control the pipeline stages, which is helpful to implement the pipelined Montgomery algorithm. Here, for the sake of the brevity and portability of the design, we do not engage dual clock and complex control signals like [5,19] which force DSP blocks to work in the maximum frequency.

3.3 Processing Method for Pipelined Implementation

According to the features of DSP resources, the processing method for pipelined implementation is presented in Algorithm 2. From Algorithm 1, we observe that M' is a pre-calculated integer, and the bit length m of $M' = \sum_{i=0}^{m-1}(2^k)^i m'_i$ equals that of modulus M, and the last statement in Algorithm 1 is just a left shift of S_{n+d+1} where the d last quotient digits are shifted in from the right. Here, the radix is set to 2^{16} and the delay parameter $d = 3$, yielding that $n \leq m + d + 2 = m + 5$. The remaining inputs appearing in Algorithm 1 are omitted.

Now we explain the consistency between Algorithms 1 and 2. There are three phases in Algorithm 2: *Phase 0* for initialization, *Phase 1* for iteration and *Phase 2* for the final addition. The initialization should be executed before each multiplication begins. In *Phase 1*, a four-stage pipeline is introduced in order to utilize the DSP blocks, so the total of the surrounding loops becomes $n+6$ from $n+3$. The inner loop from 0 to $n-1$ represents the operations of n Processing Elements (PEs). In the pipeline, referring to Algorithm 1, we can see that *Stage 1* to *Stage 3* are used to calculate $w_i = q_{i_d}M' + b_iA$, and *Stage 4* is used to calculate $(S_i \bmod 2^k + w_i)$. Here, $(S_i \bmod 2^k)$ is divided into two parts: $c_{(i+3,j)}$ inside the PE itself and $s_{(i+3,j+1)}$ from the neighboring higher PE. The delay

Algorithm 2. Processing Method for Pipelined Implementation

```
Input:
    radix $2^k = 2^{16}$, delay parameter $d = 3$
    $M' = \sum_{i=0}^{m-1}(2^k)^i m'_i$ , $A = \sum_{i=0}^{n-1}(2^k)^i a_i$, $B = \sum_{i=0}^{n+d}(2^k)^i b_i$
Output:
    Integer $S_{n+5}$ where $S_{n+5} \equiv ABR^{-1} \pmod M$ and $0 \le S_{n+5} < 2\tilde{M}$
    /* Phase 0: Initialization */
 1: for $j = 0$ to $n-1$ do
 2:     $u_{(0,j)} = 32'b0, v_{(0,j)} = 32'b0$;
 3:     $w_{(0,j)} = 33'b0$;
 4:     $s_{(0,j)} = 16'b0, c_{(0,j)} = 17'b0$;
 5: end for
    /* Phase 1 */
 6: for $i = 0$ to $n+6$ do
 7:     $q_{i-3} = s_{(i,0)}$;
 8:     for $j = 0$ to $n-1$ do
 9:         Stage 1: $u_{(i+1,j)} = q_{i-3}m'_j$;
10:         Stage 2: $v_{(i+1,j)} = a_j b_i$;
11:         Stage 3: $w_{(i+1,j)} = u_{(i,j)} + v_{(i,j)}$;
12:         Stage 4: $\{c_{(i+1,j)}, s_{(i+1,j)}\} = w_{(i,j)} + c_{(i,j)} + s_{(i,j+1)}$ ;
13:     end for
14: end for
    /* Phase 2 */
15: $S_{n+4} = \sum_{j=0}^{n-4}(2^{16})^j\{c_{(n+7,j)}, s_{(n+7,j)}\}$;
16: $S_{n+5} = \{S_{n+4}, q_{n+3}, q_{n+2}, q_{n+1}\}$;
```

is caused by the pipeline. In *Stage 4*, S_i is represented by $s_{(i+3,j)}$ and $c_{(i+3,j)}$ in a redundant form, where $s_{(i+3,j)}$ represents the lower k bits and $c_{(i+3,j)}$ the $k+1$ carry bits. Note that the carry bits from lower PEs are not transferred to higher PEs, because this interconnection would increase the data dependency for calculating q_i implying that q_i cannot be generated per clock cycle. Therefore, except for q_{i-3}, the transfer of $s_{(i,j+1)}$ in *Stage 4* is the only interconnection among the PEs, ensuring that q_{i-3} can be generated per cycle. The carry bits from lower PEs to higher PEs which are saved in $c_{(i,j)}$ are processed in *Phase 2*. In brief, the goal of *Phase 1* is to generate the right quotient per clock cycle for running the iteration regardless of the representation of S_i, while by simple additions *Phase 2* transforms the redundant representation to non-redundant representation of the final value S_{n+4}. The detailed hardware architecture is presented in the next section.

4 Proposed Architecture

4.1 Montgomery Multiplier

Processing Element. The Processing Elements, each of which processes k-bit block data, form the modular multiplication array. As the input $A \le 2\tilde{M}$, n PEs are needed to compose the array. Figure 2 provides the structure of the j^{th} PE.

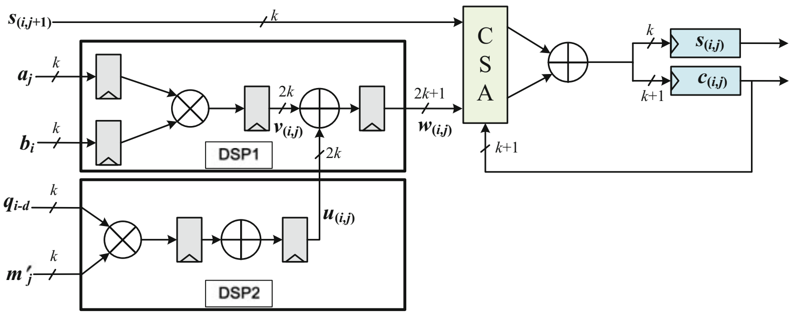

Fig. 2. The structure of the j^{th} PE

In the first three pipeline stages, the arithmetic operations $q_{i-d}M' + b_iA$, are located in the two DSP blocks named DSP1 and DSP2. In order to achieve high frequency, two stage registers are inserted in DSP2 which calculates the multiplication of q_{i-d} and m'_j. Accordingly, another stage of registers are added in DSP1 in order to wait for the multiplication result $u_{(i,j)}$ from DSP2. The addition of $u_{(i,j)}$ and $v_{(i,j)}$ is performed by DSP1 as shown in Fig. 2. In the fourth stage, the three-number addition $w_{(i,j)} + c_{(i,j)} + s_{(i,j+1)}$ is performed by using the carry-save adder (CSA). In FPGAs, CSA can be implemented by one-stage look-up tables (LUTs) and one carry propagate adder (CPA). Because the computational path between the DSP registers is shorter than the CSA, the critical path only includes three-number addition, i.e. CSA. Therefore, in this way, the PE can work in a high frequency due to the very short critical path.

Parallel PE Array. n PEs named PE_0 to PE_{n-1} have been connected to form a parallel array which performs *Phase 1* in Algorithm 2, as shown in Fig. 3. The quotient is generated from PE_0 and fed to the following PEs. Especially, in PE_m to PE_{n-1}, DSP2 and q_{i-d} are no more required since m'_j equals to zero for these PEs. The PE outputs $c_{(i,j)}$ are omitted in Fig. 3, as they only need to be outputted when the iteration in *Phase 1* finishes. Unlike the high-radix systolic array [9] and the design in [19], the PE array works in parallel for the iteration, resulting in the consumed clock cycles for transferring the values from lower PEs to higer PEs are saved.

Now we analyze the performance of the PE array. According to Algorithm 2, the number of iteration rounds of *Phase 1* is $n + 7$. Together with the one clock cycle for initialization, the processing cycles of the PE array are $n+8$. Regarding the consumed hardware resources, $n + m$ DSP blocks are required for forming the PE array.

Although the frequency may decrease due to the global signals and large bus width, fortunately we find that these factors do not have a serious impact on the hardware performance, owing to the small bit size (256 or smaller) of the operands of ECC. The impact has been verified in our experiment as shown in Sect. 5.1.

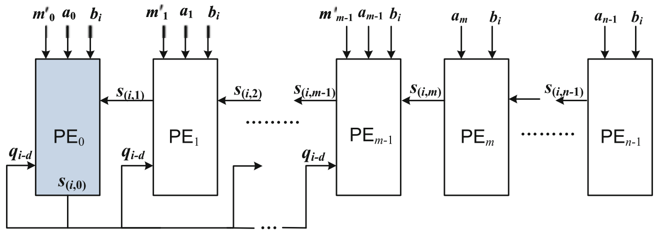

Fig. 3. The structure of the parallel PE array

The outputs of PEs are the redundant representation of the final result S_{n+4}. So some addition operations (cf. *Phase 2*) have to be performed to get the non-redundant representation before it can be used as input for a new multiplication. Here we use another circuit module - redundant number adder (RNA) to implement the operation of *Phase 2*. Actually, the PE array and RNA that work in an alternative form can be pipelined for processing independent multiplications which are inherent in the elliptic curve point calculation algorithms. Therefore, the average number of processing clock cycles for one multiplication are only $(n + 8)$ in our ECC processor.

Redundant Number Adder. The outputs of PEs should be added in an interleaved form to obtain the final result S_{n+4} by the RNA. The redundant number addition process is shown in Algorithm 3. For simplicity we rename $s_{n+7,j}$ and $c_{n+7,j}$ as s_j and c_j, respectively. Notice that there is a 1-bit overlap between c_j and c_{j+1} due to the propagation of the carry bit. Obviously, s_0 can be directly outputted. We rewrite the s_j and c_j to form three long integers SS,CL and CH in Algorithm 3, where $c_j[k-1:0]$ and $c_j[k]$ are the lowest k bits and the highest bit of c_j, respectively. Before being stored into the registers for addition, the three integers are converted to two integers by using CSA within one clock cycle. Then the addition of the two integers can be performed by a l-bit CPA in $\lceil\frac{(n-3)k}{l}\rceil$ clock cycles. For balancing the processing cycles and working frequency, the path delay of l-bit CPA is configured close to the addition of three numbers in PE. In our design, the width l is set to a value between $3k$ and $4k$.

4.2 ECC Processor Architecture

The architecture of the ECC processor based on our Montgomery modular multiplier is described in Fig. 4, where the Dual Port RAM is used to store arithmetic data. By reading a pre-configured program ROM, the Finite State Machine (FSM) controls the modular multiplier and the modular adder/subtracter (ModAdd/Sub), as well as the RAM state. Note that the widths of the data interfaces among the Dual Port RAM and the arithmetic units are all kn bits due to the parallelism of the multiplier.

Algorithm 3. Redundant number addition

```
Input:
    $s_j = s_{(n+7,j)}$, $c_j = c_{(n+7,j)}$, $j \in [0, n-4]$
Output:
    $S = \sum_{j=0}^{n-4} (2^k)^j \{c_j, s_j\}$
    /*Forming three integers*/
 1: $SS = \sum_{j=1}^{n-4} (2^k)^j s_j$
 2: $CL = \sum_{j=0}^{n-4} (2^k)^j c_j[k-1:0]$
 3: $CH = \sum_{j=0}^{n-4} (2^k)^{j+1} c_j[k]$
    /*CSA*/
 4: $X = SS \oplus CL \oplus CH$
 5: $C = (SS \& CL) | (SS \& CH) | (CL \& CH)$
    /*$l$-bit CPA*/
 6: carry $= 1'b0$
 7: for $i = 0$ to $\lceil \frac{(n-3)k}{l} \rceil - 1$ do
 8:     $\{\text{carry}, S_i\} = X_i + C_i + \text{carry}$,
        where $S_i, X_i, C_i$ represent the $i$th $l$-bit block of $S, X, C$, respectively.
 9: end for
```

Modular Adder/Subtracter. In elliptic curve arithmetic, modular additions and subtractions are interspersed among the modular multiplication arithmetic. According to Algorithm 1, for the inputs in the range of $[0, 2\tilde{M}]$ the final result S_{n+d+2} will be reduced to the range of $[0, 2\tilde{M}]$.

In our design, ModAdd/Sub performs actually straightforward integer addition/subtraction without modular reduction. As an alternative, the modular reduction is performed by the Montgomery multiplication with an expanded R. After a careful observation and scheduling, the results of ModAdd/Sub are restricted to the range of $(0, 8\tilde{M})$, as shown in Appendix A, where the squaring is treated as the generic multiplication with two identical multiplicands. The range of $(0, 8M)$ is determined by the rescheduling of elliptic curve arithmetic. For example, for calculating $8(x \times y)$ where $x, y < 2\tilde{M}$, the process is rescheduled as $(4x) \times (2y)$ to narrow the range of the result. In this case, parameter R should be expanded to $R > 64\tilde{M}$ to guarantee that for inputs in the range of $(0, 8\tilde{M})$ the result of Montgomery multiplication S still satisfies: $S < 2\tilde{M}$. The proof is omitted here.

For $A + B \bmod \tilde{M}$, the range of the addition result is $(0, 8\tilde{M})$ due to the calculation of $4x$ where $x \in (0, 2\tilde{M})$ is an output of the multiplier. Therefore, the modular addition is simplified to the integer addition $A + B$, as shown in Eq. (1). For $A - B \bmod \tilde{M}$, the range of the subtrahend B is $(0, 4\tilde{M})$ after specific scheduling, so $4\tilde{M}$ should be added to ensure that the result is positive, as shown in Eq. (2). Especially, for calculating $x - (y - z)$ where $x, y, z < 2\tilde{M}$, the process is rescheduled as $(x - y) + z \rightarrow (x - y + 4\tilde{M}) + z \in (0, 8\tilde{M})$.

$$A + B \bmod M \rightarrow A + B \in (0, 8\tilde{M}) \tag{1}$$

$$A - B \bmod M \rightarrow A - B + 4\tilde{M} \in (0, 8\tilde{M}) \tag{2}$$

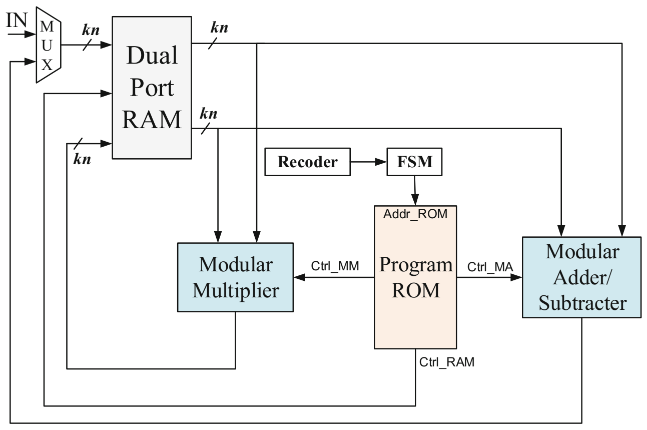

Fig. 4. The architecture of the ECC processor

Point Doubling and Addition. The point doubling and addition are implemented in Jacobian projective coordinates, under which the successive multiplications can be performed independently. The process of point doubling and addition with specific scheduling is presented in Appendix A. In the process, the dependencies of adjacent multiplications are avoided to fully exploit the multiplier, and the range of the modular addition/subtraction output satisfies the required conditions. After the above optimizations, completing one point doubling operation needs the processing cycles of 8 multiplications, and completing one point addition operation needs the processing cycles of 16 multiplications and 2 subtractions/additions.

SCA Resistance. Considering the SCA resistance and the efficiency, we combine randomized Jacobian coordinates method and a window method [13] against differential power analysis (DPA) and simple power analysis (SPA), respectively. The randomization technique transforms the base point $(x, y, 1)$ of projective coordinates to (r^2x, r^3y, r) with a random number $r \neq 0$. The window method in [13] based on a special recoding algorithm makes minimum information leak in the computation time, and it is efficient under Jacobian coordinates with a pre-computed table. A more efficient method was presented in [14], and a security enhanced method, which avoided a fixed table and achieved comparative efficiency, was proposed in [15]. For computing point multiplication, the window-based method [13] requires $2^{w-1} + tw$ point doublings and $2^{w-1} - 1 + t$ point additions, where w is the window size and t is the number of words after recoding. The pre-computing time has been taken into account, and the base point is not assumed to be fixed. The pre-computed table with $2^w - 1$ points can be easily implemented by block RAMs which are abundant in modern FPGAs, and the cost is acceptable for our design. Note that the randomization technique causes no impact on the area and little decrease in the speed, as the randomization is executed only twice or once [13].

5 Implementation and Comparison

5.1 Hardware Implementation

Our ECC processor for 256-bit curves named *ECC-256p* is implemented on Xilinx Virtex-4 (XC4VLX100-12FF1148) and Virtex-5 (XC5LX110T-3FF1136) FPGA devices. In order to keep the length of the critical path as expected and simultaneously achieve a high efficiency, the addition width is set to 54 for RNA and ModAdd/Sub, the path delay of which is shorter than that of three number addition. Therefore, as expected, the critical path of *ECC-256p* is the addition of three 32-bit number in the PE. The Montgomery modular multiplier can complete one operation in $n + 14$ clock cycles that consists of $n + 8$ cycles for the PE array and 6 cycles for the RNA, and the former is the average number of clock cycles for ECC point calculation. For the window-based algorithm of point multiplications, the window size w is set to 4, and the maximum t after recoding is 64 for 256-bit prime fields. In this case, one point multiplication requires 264 doublings and 71 additions at the cost of a pre-computed table with 15 points.

Table 1. Clock cycles for *ECC-256p* under Jacobian projective coordinates

Operation	*ECC-256p*
MUL	35 (average 29)
ADD/SUB	7
Point Doubling (Jacobian)	232
Point Addition (Jacobian)	484
Inversion (Fermat)	13685
Point Multiplication (Window)	109297

The number of clock cycles for the operations are shown in Table 1, and Post and Route (PAR) results on Virtex-4 and Virtex-5 are given in Table 2. In our results, the final inversion at the end of the scalar multiplication is taken into account. We use Fermats little theorem to compute the inversion. According to Table 2, *ECC-256p* can process one point multiplication in 109297 cycles under 250 MHz and 291 MHz frequency, meaning that the operation can be completed within 0.44 ms and 0.38 ms on Virtex-4 and Virtex-5, respectively. Note that the amounts of consumed hardware resource are different in the two devices, since the Virtex-5 resource units, such as slice, LUT and BRAM, have larger capacity. In particular, each slice in Virtex-5 contains four LUTs and flip-flops, while the number is two in Virtex-4 slice. Therefore, the total occupied slices are significantly reduced when the design is implemented on Virtex-5.

Table 2. PAR results of *ECC-256p* on Virtex-4 and Virtex-5

	Virtex-4	Virtex-5
Slices	4655	1725
LUTs	5740 (4-input)	4177 (6-input)
Flip-flops	4876	4792
DSP blocks	37	37
BRAMs	11 (18 Kb)	10 (36 Kb)
Frequency (Delay)	250 MHz (0.44 ms)	291 MHz (0.38 ms)

5.2 Performance Comparison and Discussion

The comparison results are shown in Table 3, where the first three works support generic elliptic curves, while the last two only support NIST curves. In addition, our work and [4,11] are SCA resistant, while the others are not. We have labeled these differences in Table 3.

As far as we know, the fastest ECC processor for generic curves is [4], which uses RNS representations to speed up the computation. Substantial hardware resources (96 DSP blocks and 9177 ALM) in Stratix II FPGA are used for the implementation. In fact, Stratix II and Virtex-4 are at the same level, since the process nodes of the two devices are both 90 nm. Assuming that a Stratix II ALM and a Virtex-4 Slice are equivalent, our processor saves more than half resources compared with [4]. In the aspect of speed, our design is faster than [4] by more than 40 % from the perspective of implementation results. However, note that employing different SCA protections makes the performance quite different. In [4], Montgomery ladder which is a time-hungry technique against SPA and error-injection attacks was engaged. As the speed is the main concern, in our design it is not optimal (or even a waste) to adopt those countermeasures such as indistinguishable point addition formulae and Montgomery ladder, because the point doubling operation is nearly twice faster than the addition under Jacobian coordinates. In addition, our design has great advantages in area over [4]. Therefore, we use the window-based method which is a type of resource-consuming but efficient countermeasure against SPA. In brief, for generic curves over GF(p), our work provides an efficient alternative to achieve a higher speed and competitive security with a much more concise design.

The designs in [10,11] are both based on the classic Montgomery algorithm, and implemented in earlier FPGAs Virtex-2 Pro, which did not supported DSP blocks yet. To our best knowledge, the architecture [11] is the fastest among the implementations based on the Montgomery multiplication for generic curves. In [11], the multiplication result was written in a special carry-save form to shorten the long computational path. But the maximum frequency was reduced to a very low level. As the targeted platform of our design is more advanced than that of [11], it is necessary to explain that our frequency is higher than [11] by a large margin from the aspect of the critical path. The critical path of [11] is composed of one adder, two 16-bit multipliers and some stage LUTs for 6-2 CSA, whereas

Table 3. Hardware performance comparison of this work and other ECC cores

	Curve	Device	Size (DSP)	Frequency (MHz)	Delay (ms)	SCA res.
Our work	256 any	Virtex-5	1725 Slices (37 DSPs)	291	0.38	Yes
	256 any	Virtex-4	4655 Slices (37 DSPs)	250	0.44	Yes
[4]	256 any	Stratix II	9177 ALM (96 DSPs)	157	0.68	Yes
[11]	256 any	Virtex-2 Pro	3529 Slices (36 MULTs)	67	2.35	Yes
[10]	256 any	Virtex-2 Pro	15755 Slices (256 MULTs)	39.5	3.84	No
[5]	256 NIST	Virtex-4	1715 Slices (32 DSPs)	487	0.49	No
[17]	192 NIST	Virtex-E	5708 Slices	40	3	No

the critical path in our design is only one stage LUTs for 3-2 CSA and one 32-bit adder. As a result, owing to the quotient pipelining technique applied in DSP blocks, the critical path is shortened significantly in our design.

The architecture described in [5] is the fastest FPGA implementation of elliptic curve point multiplication over GF(p), but with restrictions on primes. It computes point multiplication over NIST curves which are widely used and standardized in practice. It is a dual clock design, and shifts all the field arithmetic operations into DSP blocks, thus the design occupies a small area and runs at a high speed (487 MHz) on Virtex-4. Our design extends the application to support generic curves at a higher speed, and our architecture is not limited in FPGA platforms. In fact, our architecture can be easily transferred to application specific integrated circuits (ASICs) by replacing the multiplier cores, i.e. DSP blocks with excellent pipelined multiplier IP cores. It will be more flexible on ASICs to configure the delay parameter and the radix to maximize the hardware performance. Furthermore, notice that the drawbacks of the pipelined Montgomery algorithm, i.e. the wider range and additional iteration cycles mentioned in Sect. 2.1, can be eliminated for commonly used pseudo Mersenne primes. Taking NIST prime P-256 $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ as an example, the least 96 significant bits are all ‘1’, so the parameter $\bar{M}$ equals 1 in Algorithm 1 for k, d satisfying $2^{k(d+1)} \leq 2^{96}$ and then $\tilde{M}$ is reduced to $\tilde{M} = M$. In this case, the range of pre-computed parameters are corresponding to the width in the traditional Montgomery algorithm. Therefore, if our architecture is designed for P-256, the performance will be further improved.

6 Conclusion and Future Work

This paper presents a high-speed elliptic curve cryptographic processor for generic curves over GF(p) based on our novel Montgomery multiplier. We combine the quotient pipelining Montgomery multiplication algorithm with our new parallel array design, resulting in our multiplier completes one single Montgomery multiplication in approximately n clock cycles and also works in a high frequency. Also, employing the multiplier, we implement the ECC processor for scalar multiplications on modern FPGAs. Experimental results indicate that the design is faster than other existing ECC implementations over GF(p) on FPGAs. From the comparison results, we can see that pipelined Montgomery based scheme is a better choice than the classic Montgomery based and RNS based ones in terms of speed or consumed resources for ECC implementations. In future work, we will implement the architecture in more advanced FPGAs such as Virtex-6 and Virtex-7, and transfer it to ASIC platforms.

Acknowledgements. The authors would like to acknowledge the contributions of Doctor Zhan Wang, Jingqiang Lin, and Chenyang Tu for useful discussions. The authors also would like to thank Professor Tanja Lange from Technische Universiteit Eindhoven in the Netherlands for helpful proofreading and comments. Finally, we are grateful to the anonymous reviewers for their invaluable suggestions and comments to improve the quality and fairness of this paper.

A Rescheduling of Point Addition and Doubling in Jacobian Projective Coordinates

Given the Weierstrass equation of an elliptic curve $E : y^2 = x^3 + ax + b$ defined over GF(p), the projective point $(X : Y : Z)$, $Z \neq 0$ corresponds to the affine

Table 4. Scheduling process of point addition and doubling

	Point Addition		Point Doubling	
Step	MUL	ADD/SUB	MUL	ADD/SUB
1	$L_1 = Z_2 \times Z_2$		$L_1 = Z_1 \times Z_1$	$L_2 = Y_1 + Y_1$
2	$L_2 = Z_1 \times Z_1$		$L_3 = L_2 \times L_2$	$L_4 = X_1 + L_1$
				$L_5 = X_1 - L_1$
3	$\lambda_1 = X_1 \times L_1$		$L_{13} = L_4 \times L_5$	$L_6 = X_1 + X_1$
4	$\lambda_2 = X_2 \times L_2$		$\lambda_2 = L_3 \times L_6$	$\lambda_1 = 3L_{13}$
5	$L_3 = Y_1 \times Z_2$	$\lambda_3 = \lambda_1 - \lambda_2$	$L_9 = \lambda_1 \times \lambda_1$	$L_7 = \lambda_2/2$
		$\lambda_7 = \lambda_1 + \lambda_2$		$L_8 = \lambda_2 + L_7$
6	$L_4 = Z_1 \times Y_2$		$L_{10} = L_3 \times L_3$	$L_{11} = L_8 - L_9$
7	$\lambda_4 = L_1 \times L_3$		$L_{12} = L_{11} \times \lambda_1$	$\lambda_3 = L_{10}/2$
				$X_3 = L_9 - \lambda_2$
8	$\lambda_5 = L_2 \times L_4$		$Z_3 = L_2 \times Z_1$	$Y_3 = L_{12} - \lambda_3$
9	$L_5 = Z_1 \times Z_2$	$\lambda_6 = \lambda_4 - \lambda_5$		
		$\lambda_8 = \lambda_4 + \lambda_5$		
10	$L_6 = \lambda_3 \times \lambda_3$			
11	$L_7 = \lambda_6 \times \lambda_6$	$L_8 = \lambda_7 + \lambda_7$		
12	$L_9 = L_6 \times L_8$			
13	$L_{10} = \lambda_3 \times L_6$	$L_{11} = L_9/2$		
		$L_{12} = L_{11} + L_9$		
		$L_{13} = L_7 + L_7$		
14	$Z_3 = \lambda_3 \times L_5$	$X_3 = L_7 - L_{11}$		
		$L_{15} = L_{12} - L_{13}$		
15	$L_{14} = \lambda_8 \times L_{10}$			
16	$L_{16} = L_{15} \times \lambda_6$			
17		$Y_3 = L_{16} - L_{14}$		
18		$Y_3 = Y_3/2$		

point $(X/Z^2, Y/Z^3)$ in Jacobian projective coordinates. Here we assume that the elliptic curve $y^2 = x^3 + ax + b$ has $a = -3$ without much loss of generality. Given two points $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$ in Jacobian coordinates, sum $P_3 = P_1 + P_2$. The point addition calculation process for $P_1 \neq P_2$ and point doubling calculation process for $P_1 = P_2$ are scheduled as given in Table 4.

References

1. Blum, T., Paar, C.: High-radix montgomery modular exponentiation on reconfigurable hardware. IEEE Trans. Comput. **50**(7), 759–764 (2001)
2. Daly, A., Marnane, W.P., Kerins, T., Popovici, E.M.: An FPGA implementation of a GF(p) ALU for encryption processors. Microprocess. Microsyst. **28**(5–6), 253–260 (2004)
3. Eldridge, S.E., Walter, C.D.: Hardware implementation of montgomery's modular multiplication algorithm. IEEE Trans. Comput. **42**(6), 693–699 (1993)
4. Guillermin, N.: A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 48–64. Springer, Heidelberg (2010)
5. Güneysu, T., Paar, Ch.: Ultra high performance ECC over NIST primes on commercial FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 62–78. Springer, Heidelberg (2008)
6. Hankerson, D., Vanstone, S., Menezes, A.J.: Guide to elliptic curve cryptography. Springer, New York (2004)
7. Huang, M., Gaj, K., Kwon, S., El-Ghazawi, T.: An optimized hardware architecture for the montgomery multiplication algorithm. In: Cramer, R. (ed.) PKC 2008. LNCS, vol. 4939, pp. 214–228. Springer, Heidelberg (2008)
8. Koblitz, N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987)
9. McIvor, C., McLoone, M., McCanny, J.V.: High-radix systolic modular multiplication on reconfigurable hardware. In: Brebner, G.J., Chakraborty, S., Wong, W.F. (eds) FPT 2005, pp. 13–18. IEEE (2005)
10. McIvor, C.J., McLoone, M., McCanny, J.V.: Hardware elliptic curve cryptographic processor over GF(p). IEEE Trans. Circ. Syst. I: Regul. Pap. **53**(9), 1946–1957 (2006)
11. Mentens, N.: Secure and efficient coprocessor design for cryptographic applications on FPGAs. Ph.D. thesis, Katholieke Universiteit Leuven (2007)
12. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
13. Möller, B.: Securing elliptic curve point multiplication against side-channel attacks. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 324–334. Springer, Heidelberg (2001)
14. Möller, B.: Securing elliptic curve point multiplication against side-channel attacks, addendum: Efficiency improvement. http://pdf.aminer.org/000/452/864/securing_elliptic_curve_point_multiplication_against_side_channel_attacks.pdf (2001)
15. Möller, B.: Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 402–413. Springer, Heidelberg (2002)
16. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)

17. Orlando, G., Paar, C.: A scalable GF(p) elliptic curve processor architecture for programmable hardware. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 348–363. Springer, Heidelberg (2001)
18. Orup, H.: Simplifying quotient determination in high-radix modular multiplication. In: IEEE Symposium on Computer Arithmetic, pp. 193–199 (1995)
19. Suzuki, D.: How to maximize the potential of FPGA resources for modular exponentiation. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 272–288. Springer, Heidelberg (2007)
20. Tang, S.H., Tsui, K.S., Leong, P.H.W.: Modular exponentiation using parallel multipliers. In: FPT 2003, pp. 52–59. IEEE (2003)
21. Tenca, A.F., Koç, Ç.K.: A scalable architecture for montgomery multiplication. In: Koç, Ç.K., Paar, Ch. (eds.) CHES 1999. LNCS, vol. 1717, pp. 94–108. Springer, Heidelberg (1999)
22. Tenca, A.F., Todorov, G., Koç, Ç.K.: High-radix design of a scalable modular multiplier. In: Koç, Ç.K., Naccache, D., Paar, Ch. (eds.) CHES 2001. LNCS, vol. 2162, pp. 185–201. Springer, Heidelberg (2001)
23. Walter, C.D.: Systolic modular multiplication. IEEE Trans. Comput. **42**(3), 376–378 (1993)