High-Performance Architecture of Elliptic Curve Scalar Multiplication

Bijan Ansari and M. Anwar Hasan, *Senior Member*, *IEEE*

Abstract—A high-performance architecture of elliptic curve scalar multiplication based on the Montgomery ladder method over finite field $GF(2^m)$ is proposed. A pseudopipelined word-serial finite field multiplier, with word size w, suitable for the scalar multiplication is also developed. Implemented in hardware, this system performs a scalar multiplication in approximately $6\lceil m/w \rceil (m-1)$ clock cycles, and the gate delay in the critical path is equal to $T_{AND} + \lceil \log_2(w/k) \rceil T_{XOR}$, where T_{AND} and T_{XOR} are delays due to two-input AND and XOR gates, respectively, and $1 \leq k \ll w$ is used to shorten the critical path.

Index Terms—Elliptic curves, Finite Fields, scalar multiplication.

1 Introduction

Elliptic curve scalar multiplication kP, where k is an integer and P is a point on the curve, is a fundamental operation in elliptic curve cryptosystems. In the recent past, a number of hardware architectures have been proposed in the literature to speed up this operation, for example, see [1], [2], and [3]. Among them, parallel and pipeline structures have emerged as the most promising ones for high-performance systems.

Elliptic curve scalar multiplication is normally performed by repeating point addition (ECADD) and doubling (ECDBL) operations over the curve in some special way. ECADD and ECDBL operations in turn rely on finite field (FF) operations such as addition/subtraction, multiplication, and inversion. Elliptic curve scalar multiplication is quite different from field multiplication. One way to achieve parallel and pipelined scalar multiplication is to decompose ECADD and ECDBL operations into FF operations, which result in a sequence of FF addition, subtraction, squaring, multiplication, and inversion operations. Proper grouping of these field operations reveals new possibilities for optimization. This idea is used in [1], [2], [4], and [5] to achieve parallelism and/or pipelining in the scalar multiplication operation. In [5], a number of multipliers are used in parallel. In [4], FF operations are optimized for the single-instruction multiple-data (SIMD) architecture. In [2], such a grouping has been used for obtaining pipelining and systolic operations. In [1], the sequence of operations is divided into a collection of uniform (similar) atomic blocks, where each block consists of a number of FF operations. This leads to a pipelined algorithm, in which two blocks of operations run in parallel and consequently require a double-sized hardware. The idea of atomic blocks has been used in [6] as a low-cost solution to achieve immunity against simple power analysis (SPA) attacks on the scalar multiplication.

Elliptic curve processors (ECPs) may be seen as arithmetic processors, where instead of integer/floating-point arithmetic, FF arithmetic is performed. Therefore, all traditional architectural optimizations such as runtime parallelism detection, jump or branch prediction, and pipelining are applicable. Unlike general-purpose processors, ECPs do not execute a random sequence of computations. Rather, they execute a specific sequence of FF computations, known as scalar multiplication, and once the sequence is selected and optimized for execution, there is no need for change. Runtime detection of parallelism has been employed in [3] and [7]. In general, this approach imposes complexity overhead. However, it is effective when the execution of multiple algorithms is required. Since scalar multiplication algorithms can be optimized offline, VLIW architectures could be an alternative to the superscaling approach employed in [7].

Grouping of FF operations using uniform blocks with comparable computational complexity is a key factor in the implementation of parallel and/or pipelined algorithms. Among the FF operations, the execution time of a squaring operation varies considerably depending on the type of fields—prime or extension. For field $GF(p)$, where p is prime, the complexity of squaring is comparable to the complexity of multiplication. This approach has been used in [1] to create the uniform grouping and atomic blocks. However, in binary extension field $GF(2^m)$, when the irreducible polynomial defining the field is known in advance, the complexity of squaring is significantly lower than that of multiplication and, under certain situations, becomes comparable to that of addition [3], [8], [9]. In practice, FF squaring, like FF addition, can be performed in one clock cycle. For example, in the prime field, a multiplication or a squaring operation may be grouped with an addition, while in the extension field, a multiplication should be grouped with an addition or squaring to

- *B. Ansari is with the Electrical Engineering Department, University of California, Los Angeles, CA 90095. E-mail: ansari@ucla.edu.*
- *M.A. Hasan is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: ahasan@uwaterloo.ca.*

Manuscript received 12 Nov. 2007; revised 12 Mar. 2008; accepted 27 Mar. 2008; published online 25 July 2008.
Recommended for acceptance by R. Steinwandt, W. Geiselmann, and Ç.K. Koç. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-11-0586.
Digital Object Identifier no. 10.1109/TC.2008.133.

TABLE 1
Typical Number of Clock Cycles of Basic FF Operations

Design	m	Multiplication	Addition	Squaring
[10]	167	7	3	3
[3]	163	7 to 12	3	3
[11]	233	9	≥ 2	2 (est.)
[9]	163	7	3	2

construct computation blocks with the same complexity. In fact, the pipelining scheme introduced in [1] would not be possible/efficient over $GF(2^m)$ with known irreducible polynomial. Therefore, depending on the complexity of FF squaring, a different approach to the grouping is needed for elliptic curve computations.

FF multiplication is the bottleneck of scalar multiplication, especially using projective coordinates. Most high-speed ECPs in $GF(2^m)$ use a word-serial (WS) FF multiplier, either in direct form [3], [10] or in the Karatsuba form [3], [11], [12]. Assuming the field size of 2^m elements and the word size of w bits, a typical WS multiplication algorithm is normally performed in $\lceil m/w \rceil$ iterations. It is also common in the literature to ignore the execution time of FF addition (and, sometimes, FF squaring) compared to the execution time of FF multiplication. Simple analyses show that scalar multiplication is achievable in $N(m-1)\lceil m/w \rceil$ clock cycles, where N is the number of FF multiplications in one iteration of the scalar multiplication loop. However, this level of performance has not been reported in the literature yet, and the main four reasons appear to be the following:

1. In the hardware implementation of WS FF multipliers, a few extra clock cycles are spent on loading inputs and unloading outputs [3], [9], [10]. In practice, this leads to a total of $\lceil m/w \rceil + c$ clock cycles for one multiplication using a WS multiplier. Typically, the value of c is 3 for elliptic curves that are of practical interest [3], [9], [10]. For other FF arithmetic units, like the adder and squarer, extra clock cycles are spent to transfer data to and from memory/register file as well. Table 1 compares the execution times of these operations in terms of clock cycles as reported in various articles.
2. For the high-speed hardware implementation of operations of $GF(2^m)$, the computation times of addition and squaring in terms of the number of clock cycles are comparable to that of multiplication (see Table 1) and may not be ignored.
3. In a typical processor architecture, the computation units are connected to the memory/register file or to each other by a common bus. If two units require data at the same time, one of these wait or has to stay idle until the other unit releases the bus. This could lead to a large number of idle cycles for the computation units [9].
4. The FF multiplier, which occupies the bulk of hardware in a high-performance design, is not used efficiently. In some cases, the inputs of one FF multiplication depend on the output of the previous FF operation. Therefore, the FF multiplier, if implemented in pipelined form, stays idle while waiting for the next input. This is specifically true in two consecutive iterations of the scalar multiplication loop.

This work proposes an architecture/scheme for elliptic curve scalar multiplication over binary extension field $GF(2^m)$ that alleviates the above-mentioned problems. In this scheme, the output of one field multiplication operation is not used as an input to the next multiplication operation; rather, the underlying FF operations of the scalar multiplication are divided into two streams (addition/squaring and multiplication) that are executed in parallel, and simultaneous loading of operands to the multiplier and adder/squarer is permitted. The proposed scalar multiplication scheme achieves better performance by preventing the FF multiplier to become idle anytime during the entire $m-1$ iterations of the scalar multiplication loop.

We demonstrate the effectiveness of the scalar multiplication scheme by applying it to a classical processor architecture, which uses a pseudopipelined WS FF multiplier. This multiplier computes one multiplication every $\lceil m/w \rceil$ clock cycles instead of $\lceil m/w \rceil + c$, where $\lceil m/w \rceil = 4$ and $c = 3$, as reported in the literature for practical applications [3], [9]. A decrease in the number of clock cycles from seven to four is extremely useful and comes without any significant cost, since the hardware added for pipelining is negligible compared to the rest of the multiplier. This multiplier enables us to access relevant variables in parallel with FF computations.

The organization of the remainder of this article is described as follows: Section 2 briefly reviews the Montgomery scalar multiplication algorithm, which has been used in a number of hardware implementations [3], [5], [10], [13], [14], [15]. There are data dependencies in the steps of the algorithm, and hence, the latter cannot be readily executed in a pipelined fashion as desired. In Section 3, we develop a pipelined version of the scalar multiplication scheme. In Section 4, we adapt an architecture of an FF multiplier suitable for the proposed scalar multiplication scheme. In this section, implementation issues are also considered, and some results are presented. Finally, concluding remarks are given in Section 5.

2 Review of the Montgomery Scalar Multiplication

Points on an elliptic curve E, defined over a finite field $GF(2^m)$, along with a special point called infinity and a group operation known as point addition, form a commutative finite group. If P is a point on the curve E and k is a positive integer, computing

$$kP = \underbrace{P + P + P + \cdots + P}_{k \quad times}$$

is called scalar multiplication. The result of scalar multiplication is another point Q on the curve E. It is normally expressed as $Q = kP$. If E is an elliptic curve defined over $GF(2^m)$, the number of points in $E(GF(2^m))$ is called the order of E over $GF(2^m)$, denoted by $\#E(GF(2^m))$. For cryptographic applications, $\#E(GF(2^m)) = rh$, where r is a large prime and h is a small integer, and P and Q have order r. Scalars such as k are random integers, where

$1 \le k \le r-1$. Since $r \approx 2^m$, the binary representation of $k = \sum_{i=0}^{n-1} k_i 2^i$ has n bits, where $k_i \in \{0,1\}$, and $n \approx m$. Scalar multiplication is the most dominant computation part of elliptic curve cryptography (ECC). More on this can be found in [16] and [17].

Points on elliptic curves can be represented using affine or projective coordinates. Each point operation, namely, point doubling or addition, requires $I + 2M + S$ cycles in affine coordinates [17]. The extended euclidean [3] and Itoh-Tsujii [18] algorithms for FF inversion over $GF(2^m)$ requires $2m$ and $(m-1)S+(\lfloor \log_2(m-1) \rfloor + h(m-1)-1)M$ clock cycles, respectively, where $h(j)$ is the number of nonzero bits in the binary representation of the integer j. Using projective coordinates, only one inversion at the end of scalar multiplication is required at the expense of extra multiplications in the point operation formulas. If the complexity of inversion over the underlying field is significantly higher than that of multiplication, then it may be advantageous to represent points using projective coordinates. This is the case for the processors in Table 1. The affine coordinate representation could be suitable for architectures that employ serial multipliers. For two interesting ECP implementations using affine coordinates, the reader is referred to [2] and [13].

Algorithm 1 shows the projective version of the Montgomery scalar multiplication scheme for nonsupersingular elliptic curves over binary fields as it was introduced in [19]. In this algorithm, $\texttt{Madd}(X_1, Z_1, X_2, X_2)$, $\texttt{Mdouble}(X_1, Z_1)$, and $\texttt{Mxy}(X_1, Z_1, X_2, X_2)$ are functions for point addition, point doubling, and conversion of projective coordinates to affine coordinates. The computations involved in these functions can be found in the Appendix. The reader is referred to [16] and [19] for a detailed explanation.

Algorithm 1. Montgomery scalar multiplication in projective coordinates
Input: A point $P = (x, y) \in E$, an integer $k > 0$, $k = 2^{n-1} + \sum_{i=0}^{n-2} k_i 2^i$, $k_i \in \{0,1\}$
Output: $Q = kP = (x_k, y_k)$

1: $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$ {compute P and $2P$}
2: **if** ($k = 0$ **or** $x = 0$) **then**
3: $x \leftarrow 0$, $y \leftarrow 0$ {This part is provided to have a complete presentation of algorithm. In practice, the algorithm inputs must be checked for validity before entering the scalar multiplication unit.}
4: **stop**
5: **end if**
6: **for** $i = n-2$ **to** 0 **do**
7: **if** $k_i = 1$ **then**
8: $(X_1, Z_1) \leftarrow \texttt{Madd}(X_1, Z_1, X_2, Z_2)$, $(X_2, Z_2) \leftarrow \texttt{Mdouble}(X_2, Z_2)$
9: **else**
10: $(X_2, Z_2) \leftarrow \texttt{Madd}(X_2, Z_2, X_1, Z_1)$, $(X_1, Z_1) \leftarrow \texttt{Mdouble}(X_1, Z_1)$
11: **end if**
12: **end for**
13: $Q \leftarrow \texttt{Mxy}(X_1, Z_1, X_2, Z_2)$
14: **return** Q

This algorithm has been used in several high-speed ECC implementations [3], [10], [20]. For a straightforward implementation in hardware, it may take as many as $(m-1)(6M + 3A + 5S) + (10M + 7A + 4S + I)$ clock cycles, where M, A, S, and I are the number of clock cycles required for multiplication, addition, squaring, and inversion, respectively, in the underlying FF, and m is the dimension of the binary extension field $GF(2^m)$.

3 Architecture for Scalar Multiplication

Since the FF multiplier is the bottleneck of scalar multiplication, it requires special consideration for realizing high-performance architecture for scalar multiplication. One of our goals is to utilize the multiplier in such a way so that it effectively becomes the sole component that determines the time duration of each pass of the loop in the scalar multiplication algorithm. Our other goal is to keep the multiplier core working during the entire time of the loop of the algorithm including the transition from one iteration to the next iteration.

3.1 Merging of Two Execution Paths

In order to keep the algorithm uniform and suitable for hardware implementation, we can merge the two k_i dependent execution paths of Algorithm 1. It is sufficient to swap X_1 with X_2 and Z_1 with Z_2 before computation and swap them back afterward if k_i equals to one, as it is shown in Algorithm 2.[1]

In hardware, when an indexing mechanism is utilized to access variables X_1, X_2, Z_1, and Z_2, swapping can be easily performed by exchanging the address lines to these registers or by an equivalent mechanism. Such swapping can be done on the fly using combinational logic only. A swap signal can be generated using the state of k_{i-1} and k_i. It can then be applied to the address logic of the register file.

We assume fixed irreducible polynomial, which implies that the complexity of squaring is equal to that of addition [8]. It is also assumed that the multiplication takes longer than addition and squaring, which makes the critical path of the scalar multiplication operation dependent only on the FF multiplication. The case of generic polynomials, i.e., unnamed curves or scalable processors and multiple fixed polynomials are discussed at the end of this section and the next section, respectively.

3.2 Parallel Execution

If the FF operations required for each `Madd(.)` and `Mdouble(.)` as defined in the Appendix are performed in sequence, then each pass of the main loop of Algorithm 2 will require about $6M + 3A + 5S$ clock cycles.

Algorithm 2. Scalar multiplication algorithm with uniform addressing
Input: A point $P = (x, y) \in E$, an integer $k > 0$, $k = 2^{n-1} + \sum_{i=0}^{n-2} k_i 2^i$, $k_i \in \{0,1\}$
Output: $Q = kP = (x_k, y_k)$

1: $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
2: **if** ($k = 0$ **or** $x = 0$) **then**
3: $Q \leftarrow \mathcal{O}$
4: **stop**

1. The same result may be obtained by expressing the computation part in the following way: $(X_{\overline{k_i}+1}, Z_{\overline{k_i}+1}) \leftarrow \texttt{Madd}(X_1, Z_1, X_2, Z_2)$, $(X_1, Z_1) \leftarrow \texttt{Mdouble}(X_{k_i+1}, Z_{k_i+1})$, where $\overline{k_i}$ is the logical inversion of k_i.

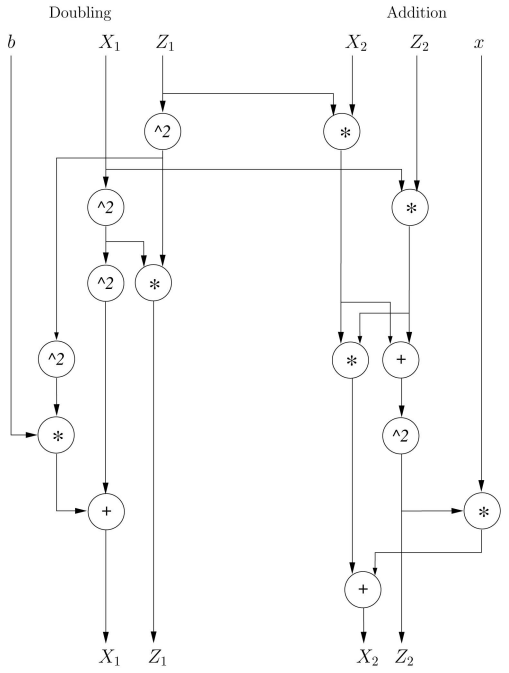

Fig. 1. Parallel execution of multiplication and addition/squaring in one iteration.

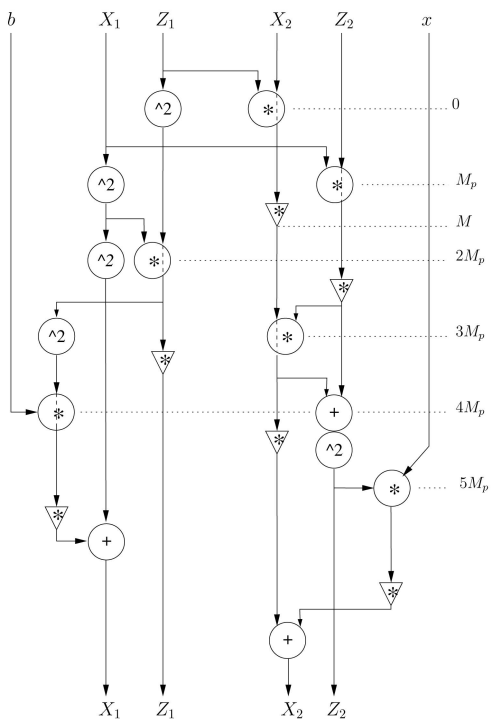

Fig. 2. Parallel with no idle cycle in the middle of the iteration.

5: **end if**
6: **if** $k_{n-2} = 1$ **then**
7: $\texttt{Swap}(X_1, X_2)$, $\texttt{Swap}(Z_1, Z_2)$
8: **end if**
9: **for** $i = n - 2$ **to** 0 **do**
10: $(X_2, Z_2) \leftarrow \texttt{Madd}(X_1, Z_1, X_2, Z_2)$,
$(X_1, Z_1) \leftarrow \texttt{Mdouble}(X_1, Z_1)$
11: **if** ($i \neq 0$ **and** $k_i \neq k_{i-1}$) **or** ($i = 0$ **and** $k_i = 1$) **then**
12: $\texttt{Swap}(X_1, X_2)$, $\texttt{Swap}(Z_1, Z_2)$
13: **end if**
14: **end for**
15: $Q \leftarrow \texttt{Mxy}(X_1, Z_1, X_2, Z_2)$
16: **return** Q

Fig. 1 depicts the flowgraph of one pass of the loop of the scalar multiplication algorithm in which each multiplication is performed in parallel with an addition and/or a squaring. It is assumed that multiplication takes longer than addition and squaring, which makes the critical path of the scalar multiplication operation dependent only on the FF multiplication. Using this technique, the execution time for one pass in the scalar multiplication loop is equal to $6M + A$.

Fig. 1 is based on the assumption that only one multiplier is available. Should two multipliers of the same specification be available, then two multiplications may be grouped along with addition and/or squaring and reduce the execution time to $3M + A$ clock cycles [21]. It is also possible to utilize more multipliers in parallel [5].

3.3 Data Dependency at Transitions of Iterations

Let $M = M_p + c$ be the number of clock cycles needed for an FF multiplication, where M_p ($= \lceil m/w \rceil$ for a WS multiplier) is the number of clock cycles needed to compute the product, and c is the total clock cycles needed to load the input and unload the product from the multiplier.

If not properly designed, during loading and unloading of operands that require c clock cycles, the multiplier core becomes idle. In the flowgraph shown in Fig. 1, performance can be improved if the idle period can be reduced. In order to prevent the multiplier core to become idle, a new set of operands need to be fed to the multiplier at every M_p clock cycles. At the same time, one needs to make certain that the next multiplication is not dependent on the output of current one because the results will be ready only after $M_p + c$ clock cycles.

The flowgraph of scalar multiplication in Fig. 2 assumes a multiplier with a computation time of M_p clock cycles and a total multiplication time of $M = M_p + c$ clock cycle. Each ⊛ (circle) in the flowgraph corresponds to the start of an FF multiplication. Vertically below each circle, there is a triangle to indicate the end of the multiplication that originated at the circle. The distance between a circle and the corresponding triangle is $M = M_p + c$ clock cycles. The minimum time difference between two consecutive circles (or two consecutive triangles) is the operation rate M_p of the multiplier. The result of the multiplication cannot be used before the triangle in the flowgraph, which is $M = M_p + c$ clock cycles after the corresponding circle. We assume that $M_p > A$ and $M_p > S$ to allow addition and squaring to be performed in parallel with multiplication. In the flowgraph, the multiplier receives its operands regularly and at equal intervals, which is M_p clock cycles. Using this scheme, the total execution time is reduced to $5M_p + M + A$ clock cycles compared to $6M + A$ clock cycles in the flowgraph in Fig. 1.

Near the bottom of the flowgraph in Fig. 2, the adder needs to wait until the multiplier computes the field multiplication and the output of the adder would be the next input to the multiplier, which starts on the next

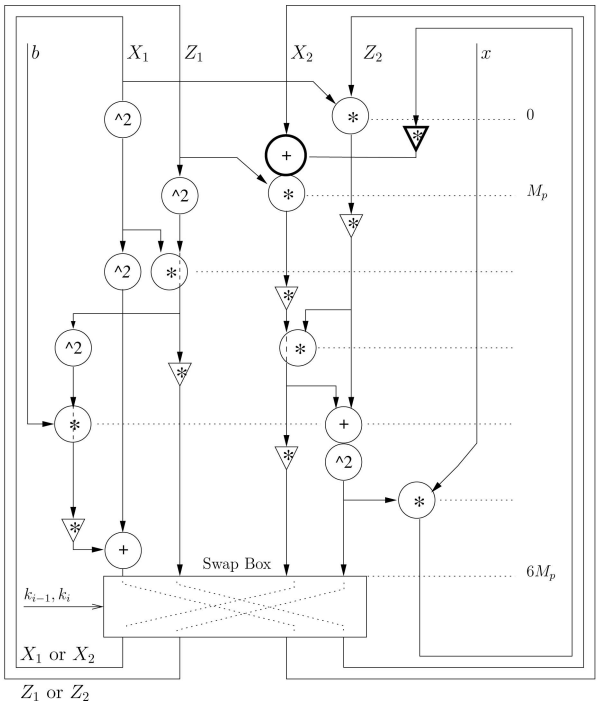

Fig. 3. Parallel with no idle cycle in entire scalar multiplication loop.

iteration. This causes a delay of $(M - M_p + A)$ clock cycles per iteration, which in turn translates into an overall delay of $(m-1)(M - M_p + A)$ clock cycles in the scalar multiplication operation. This delay can be eliminated as follows.

3.4 Resolving Data Dependency at Iteration Transitions

Based on `Madd(.)` in the Appendix, the first multiplication in an iteration of the loop can be either $X_1 * Z_2$ or $X_2 * Z_1$. We observe that Z_1, Z_2, and X_1 are ready before X_2 becomes available in the last FF multiplication in the flowgraph in Fig. 2. This suggests that the next iteration can be started by $X_1 * Z_2$ operation and before X_2 becomes available.

If $k_i = 0$, we may start the next iteration by the $X_1 * Z_2$ operation. However, if $k_i = 1$, the variables are swapped as in Algorithm 2; X_2 in the current cycle goes to X_1 in the next cycle. Therefore, we should start the next cycle with the $X_2 * Z_1$ operation, which is actually a $X_1 * Z_2$ operation. In the new arrangement, the first multiplication in the loop will depend on k_i. The complete loop is shown in Fig. 3.

A switch at the end (or start) of the flowgraph swaps registers properly. Swapping does not take extra clock cycles since the logic is rather simple and can be realized by combinational logic (Algorithm 2). The next iteration starts before the end of the last multiplication. One addition operation, from the end of the previous iteration and termination of one multiplication appears at the start of the next iteration. This is highlighted in Fig. 3 with a boldfaced circle and triangle. As shown, each iteration takes $6M_p$ clock cycles, and the multiplier does not become idle.

The scheme can be implemented by a multiplier that has a computational time long enough to allow an addition or squaring to be performed in parallel with the multiplication. An FF multiplier suitable for this scheme is proposed in Section 4.2.

Table 2 summarizes and compares the speedups of scalar multiplication operation as mapped on to the flowgraphs in Figs. 1, 2, and 3. For FF operations indicated in the flowgraph, high-speed architectures similar to [3], [9], [10], and [11] are assumed. In these architectures, one typically has $A = S = 3$ clock cycles, $M_p = \lceil m/w \rceil$ clock cycles, and $M = M_p + 3 = 7$ clock cycles. The first row in Table 2 serves as a basis of comparison and corresponds to a straightforward hardware implementation of Algorithm 1. As an example of speedups, the bottom row of the table indicates that a scalar multiplication using Fig. 3 is almost 2.75 times faster than that using a straightforward implementation of Algorithm 1.

As shown in Fig. 3, each multiplication is paired with one squaring. If unnamed curves are used, another multiplier may be employed in parallel with the original multiplier to perform the squaring operation. In the flowgraph, some multiplications take their input from a squarer. The number of cycles in the flowgraph will be affected, unless squaring is performed in M_p or less cycles. The critical path of the hardware is not affected as all multiplications and squarings are performed in parallel.

4 Implementation

The number of clock cycles by itself is not an accurate measure of performance of the system, since the clock rates may vary considerably. Therefore, an implementation is carried out to verify the performance of the system.

4.1 Implemented Architecture

Traditional ECPs are based on an instruction set that allows them to execute different scalar multiplication schemes [3], [9], [10]. The proposed scalar multiplication scheme is highly optimized toward the execution of the Montgomery ladder in projective coordinates [19], [22], [23]. Therefore, it is implemented in the form of a programmable state machine. Fig. 4 shows the basic architecture of the execution unit, which consists of a squaring/addition unit, an FF multiplier, a dual-port $8 \times m$ bit register file, control unit (not shown), and an address swapping logic. An FF addition or squaring, an FF multiplication, and a load/

TABLE 2
Performance with Various Enhancement Methods Assuming that $\lceil m/w \rceil = 4$, $A = S = 3$ Clock Cycles

Method	#Clks in one iteration	#Clks in $(m-1)$ iterations	Relative speed-ups
Straight-forward (Alg. 1)	$6M + 3A + 5S$	$66(m-1)$	1.00
Parallel addition/squaring (Fig. 1)	$6M + A$	$45(m-1)$	1.47
No idle cycle for the FF multiplier (Fig. 2)	$5M_p + M + A$	$30(m-1)$	2.13
No idle cycle in the entire operation (Fig. 3)	$6M_p$	$24(m-1)$	2.75

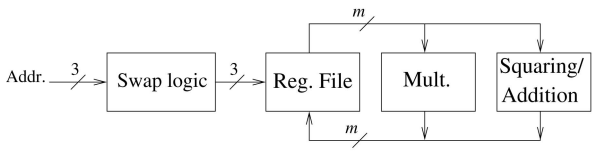

Fig. 4. Implemented architecture.

save operation from/to the register file can be performed in parallel. A squaring, for example, is performed in three clock cycles—one for each of the following operands: loading the accumulator with data from the register file, squaring, and, finally, saving the result in the register file. The squarer can perform multiple squarings without storing the data in the register file. This property is used in the Itoh-Tsujii inversion algorithm at the end of the scalar multiplication, where a^{2^i} operation is required. The multiplier and the squaring/addition unit can be loaded with the same data at the same clock cycle to prevent redundant data transfer on the data bus. The register file is used to store the result at the end of each iteration, and finally, the scalar k is stored in a specific registered (not shown) and is provided to the control logic bit by bit at each iteration.

4.2 Pseudopipelined WS Finite Field Multiplier with Short Critical Path

A WS multiplication algorithm is applicable to any algebraic ring and essentially offers a time-space tradeoff. It has been employed in [3], [9], [10], and [11] over $GF(2^m)$. If the word size of w bits is used, the space complexity of the multiplier is $O(wm)$, the critical path is $T_{AND} + \lceil \log_2 w \rceil + T_{mod}$, and multiplication will take $\lceil m/w \rceil + 3$ clock cycles, where T_{mod} is modular reduction time, and loading operands and unloading the result take two and one clock cycles, respectively.

In this section, a pseudopipelined WS FF multiplier is introduced, which uses the pipeline to reduce the critical path and can be used in the proposed scalar multiplication scheme. A polynomial basis representation of the field and a fixed irreducible polynomial are assumed. For multiplying $A(x) \times B(x) (\mathrm{mod} f(x))$, input $A(x)$ is divided into $\lceil m/w \rceil$ words, i.e., $A(x) = A_{\lceil m/w \rceil - 1} x^{(\lceil m/w \rceil - 1)w} + \ldots + A_2 x^{2w} + A_1 x^w + A_0$, where A_i's are each a polynomial of degree $\leq w-1$ (degree of $A_{\lceil m/w \rceil - 1}$ might be less than the degree of other A_i's). Assume that a multiplication starts in clock cycle i. Then, input $B(x)$ is loaded in register S in cycle i (see Fig. 5). In cycle $i+1$, $B(x)$ is copied onto register T, $A(x)$ is loaded into S, and $A_{\lceil m/w \rceil - 1}$ of $A(x)$ is loaded into register t. Similar to the way $A(x)$ is divided, the content of t is divided into k words $A_j^{(k-1)}, A_j^{(k-2)}, \ldots, A_j^{(0)}$, where $j = \lceil m/w \rceil - 1$ in cycle $i+1$.

In cycle $i+2$, the k-bit parallel polynomial multipliers are used to compute $m \times w/k$ bit multiplications $A_{\lceil m/w \rceil - 1}^{k-1} B(x), \ldots, A_{\lceil m/w \rceil - 1}^{1} B(x), A_{\lceil m/w \rceil - 1}^{0} B(x)$, as shown in Fig. 5. Each of these $m + w/k - 1$ bit product polynomials is buffered in $D_{k-1}, \ldots, D_1, D_0$ registers for one cycle and then forwarded to the next stage, where they are properly accumulated to create $A_{\lceil m/w \rceil - 1} B(x)$. The output of $D_{k-1}, \ldots, D_1, D_0$ and P are shifted properly, added together, reduced mod $f(x)$, and stored in P in the next cycle. The overall multiplication has a computation time of $3 + \lceil m/w \rceil$ clock cycles, and a new pair of inputs can be fed at every $\lceil m/w \rceil$ cycles.

4.2.1 The Critical Path

The critical path of the multiplier is $max(\mathrm{path}_1, \mathrm{path}_2)$, where path_1 is the delay of the combinational logic between T and D_i registers, and likewise, path_2 is the delay between D_i and P registers.

Between T and D_i, a w/k-bit by m-bit polynomial multiplication over $GF(2)$ is performed; therefore, $\mathrm{path}_1 = T_{AND} + \lceil \log_2 w/k \rceil T_{XOR}$. In path_2, first $k+1$ $(m+w-1)$-bit polynomials are added together, and then, the result is reduced mod $f(x)$. Therefore, $\mathrm{path}_2 = \lceil \log_2 (k+1) \rceil T_{XOR} + T_{mod}$, where T_{mod} is the delay of modular reduction.

It has been shown that the reduction operation modulo an r-term irreducible polynomial $f(x)$ requires $(r-1)(m-1)$ bit operations, when two m-term polynomials are multiplied. The critical path of the reduction depends on the nonzero terms of $f(x)$. For trinomials $f(x) = x^m + x^k + 1$, where $1 < k < m/2$, the critical path is $T_{mod} = \lceil \log_2 r \rceil = 2T_{XOR}$, and $2m-2$ gates are required [8].

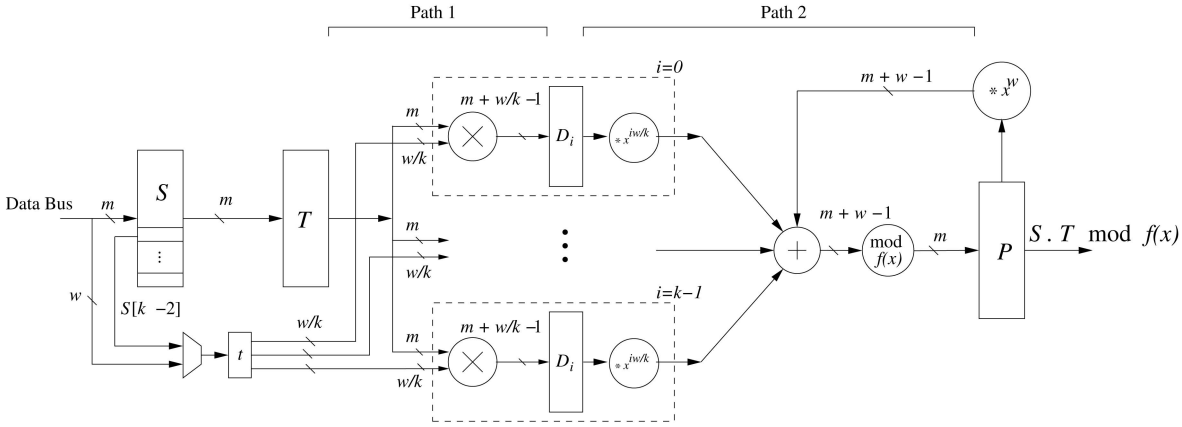

Fig. 5. Pseudopipelined FF multiplier.

TABLE 3
k versus Critical Paths

k	path_1	path_2
1	$T_{AND} + 6T_{XOR}$	$3T_{XOR}$
2	$T_{AND} + 5T_{XOR}$	$3T_{XOR}$
3	$T_{AND} + 4T_{XOR}$	$4T_{XOR}$

Delay calculation for pentanomial irreducible polynomials in the general case is complicated. In our cases, a w-term polynomial is multiplied by an m-term polynomial, and NIST irreducible polynomials are used. For pentanomial $f(x) = x^m + x^{m_3} + x^{m_2} + x^{m_1} + 1$, where $m - m_3 > w$, the reduction requires the maximum of $4(w-1)$ additions over $GF(2)$, and the critical path is $T_{mod} = \lceil \log_2 r \rceil = 3T_{XOR}$. Similar results apply for trinomials.

The variable k is used to equalize path_1 and path_2 in order to reduce the overall critical path at the expense of increased number of D_i registers. It may be viewed as a parameter that transfers parts of the computation from one stage of the pipeline to the next stage. For $k = 1$, the critical path is the same as the critical path of the multiplier used in [3]. Consider implementation over $GF(2^{163})$ of the NIST ECC standard, where $f(x) = X^{163} + x^7 + x^6 + x^3 + 1$, and $\lceil m/w \rceil = 4$. Critical paths for different values of k are given in Table 3. Since the addition and reduction operation are mixed together by the synthesizer, $\text{path}_2 = \lceil \log_2(r + k + 1) \rceil$. Note that either path_1 or path_2 can be set as the critical path. For $k = 3$, the total scalar multiplication time is reduced by almost $2(m-1)T_{XOR}$, compared to the conventional WS multiplier used in [3]. In the deep-submicron CMOS technology, interconnections and the layout congestion are the major cause of the delay, that is not taken into account here. Therefore, finding the optimum k may require a trial-and-error approach.

4.2.2 Operation

The operation of the multiplier for the special case of $\lceil m/w \rceil = 4$ and $k = 2$ is presented in Table 4. It shows register contents for two consecutive multiplications, namely, $A \times B$ and $U \times V$, assuming a "cold" start. In the table, operands A and U are split as $A = A_3x^{3w} + A_2x^{2w} + A_1x^w + A_0$ and similarly $U = U_3x^{3w} + U_2x^{2w} + U_1x^w + U_0$. One can see from the table that each multiplication takes seven cycles. The multiplier has a pipeline rate of four, i.e. after every four clock cycles, a new set of input operands can start entering the multiplier.

The key to the fast execution of scalar multiplication is to perform loading of the operands and unloading the results from the computation units in parallel with the FF computations, namely, addition, multiplication, and squaring. As an example, consider the execution of $Z_1 = Z_1 * X_1$ in parallel with squaring $X_1 = X_1^2$ and $Z_2 = Z_1 + X_2$ and the start of another multiplication $X_2 \times Z_2$.

1: $S \leftarrow X_1$, $ACC \leftarrow X_1$ {*load register S and the accumulator simultaneously*}
2: $T \leftarrow S$, $S \leftarrow Z_1$, $ACC \leftarrow ACC^2$ {*The squaring is per formed in one cycle.*}
3: $ACC \leftarrow Z_1$, $X_1 \leftarrow ACC$ {*Load ACC with the next operand* (Z_1 *is still intact*). *Save the result in ACC* (*changing* X_1 *will not affect the multiplication*). *Multiplier is still busy*}
4: $ACC \leftarrow ACC + X_2$
5: $S \leftarrow X_2$, $Z_2 \leftarrow ACC$ {*Another multiplication can start here, however the result of* $Z_1 * X_1$ *is not ready yet*}
6: $S \leftarrow Z_2$
7: $P \leftarrow Z_1 * X_1$ {*The result of multiplication is ready.*}

4.2.3 Multiple Fixed Irreducible Polynomials

Expansion to multiple fixed irreducible polynomials is straightforward. The modular reduction operation in Fig. 5 needs to be replaced with multiple mod modules and be selected using a multiplexer appropriately, and the length of registers need to be expanded to fit the elements of the field with the largest dimension. The equation for the critical path remains the same; it increases in logarithm scale as the underlying field's dimension increases. This arrangement is used in [3].

4.3 Timing

The timing diagram for the flowgraph in Fig. 3 is represented in Fig. 6. It shows the application of the pseudopipelined WS multiplier in parallel with a squarer and an adder. The smallest $\lceil m/w \rceil$ that allows parallel execution with addition and squaring and fits in our FPGA turns out to be four. Consequently, one iteration takes

TABLE 4
State Diagram of the Pseudopipelined FF Multiplier

Cycle	S	T	t	D_0	D_1	P
1	B					
2	A	B	A_3			
3	A	B	A_2	$B \times {A_3}^{(0)}$	$B \times {A_3}^{(1)}$	
4	A	B	A_1	$B \times {A_2}^{(0)}$	$B \times {A_2}^{(1)}$	$B \times A_3 \pmod{f(x)}$
5	V	B	A_0	$B \times {A_1}^{(0)}$	$B \times {A_1}^{(1)}$	$(B \times A_3)x^w + B \times A_2 \pmod{f(x)}$
6	U	V	U_3	$B \times {A_0}^{(0)}$	$B \times {A_0}^{(1)}$	$((B \times A_3)x^w + B \times A_2)x^w + B \times A_1 \pmod{f(x)}$
7	U	V	U_2	$V \times {U_3}^{(0)}$	$V \times {U_3}^{(1)}$	$(((B \times A_3)x^w + B \times A_2)x^w + B \times A_1)x^w + B \times A_0 \pmod{f(x)}$ End of $A \times B \pmod{f(x)}$
8	U	V	U_1	$V \times {U_2}^{(0)}$	$V \times {U_2}^{(1)}$	$V \times U_3 \pmod{f(x)}$
9		V	U_0	$V \times {U_1}^{(0)}$	$V \times {U_1}^{(1)}$	$(V \times U_3)x^w + V \times U_2 \pmod{f(x)}$
10				$V \times {U_0}^{(0)}$	$V \times {U_0}^{(1)}$	$((V \times U_3)x^w + V \times U_2)x^w + V \times U_1 \pmod{f(x)}$
11						$(((V \times U_3)x^w + V \times U_2)x^w + V \times U_1)x^w + V \times U_0 \pmod{f(x)}$ End of $U \times V \pmod{f(x)}$

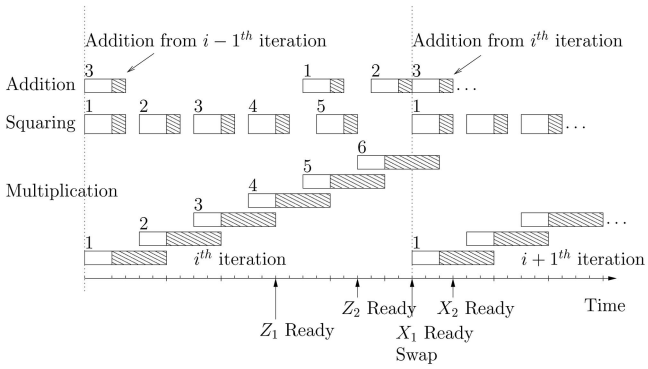

Fig. 6. Timing of scalar multiplication with no idle cycle in the entire loop.

24 clock cycles. The last multiplication and addition of the ith iteration continues to the $(i+1)$th iteration, and hence, the execution of scalar multiplication is performed without an idle cycle in the multiplication.

4.4 Implementation Results

Results of the FPGA synthesis, place and route (P&R) for Xilinx XC4VLX200 using various fields are shown in Table 5. These results are improvements over our previous results reported in [21]. The improvements are due to the use of different synthesizer and the removal of extra memory and zero detection circuits used in [21] to support different scalar multiplication algorithms in ASIC implementation. The number of flip-flops in the table partially includes the flip-flops used to implement the $8 \times m$ bit register file. The synthesizer utilizes unused flip-flops in some slices to implement the register file as well. Note that the FF multiplier is a large combinational circuit, and many flip-flops in the allocated slices remain unused.

Table 6 shows the synthesis results for $\lceil m/w \rceil \in \{4, 8, 16, 32\}$ over $GF(2^{163})$. The proposed scheme is efficient when the runtime of the multiplication is comparable to that of addition/squaring. If the multiplication operation is too slow, i.e., m/w is large, parallel operations and pseudopipelining might not be very effective. On the other hand, if m/w is small, the space complexity of the multiplier becomes much higher than that of adders/squarers. Therefore, it might be beneficial to use an algorithm with multiple on-the-fly adders/squarers.

TABLE 5
Synthesis, Place, and Route Results for Virtex4:XC4VLX200

m	Slices	Clk (MHz)	FF	LUT	Time [a] (μS)
79	1472	224	701	2784	9
113	2391	201	1117	4430	14
163	4080	197	1502	7719	21
233	7130	190	2219	13396	31
283	10893	139	3603	19972	51
409	21956	137	4013	36237	74
571	34892	107	6445	66594	133

[a]Estimated time for a scalar multiplication $\approx$ number of clock cycles $\times$ clock period $= 25(m-1) \times$ clock period.

TABLE 6
$\lceil \frac{m}{w} \rceil$ versus Slices over $GF(2^{163})$ for Virtex 2:XC2V2000

$\lceil \frac{m}{w} \rceil$	Slices
4	4192
8	2860
16	2323
32	2192

4.5 Comparison

The implementation results on XC2V2000 using Synplify Pro are reflected in Table 7, where $k = 1$ and for the bottom two rows, the conventional quadratic multiplication and the Karatsuba algorithms have been used in the bit parallel multiplier in Fig. 5. For the Karatsuba version over $GF(2^{163})$, the 164×41 bit multiplication in Fig. 5 is first decomposed into 40×40 bit multiplication, and then, the Karatsuba algorithm is applied three times along with one 5-bit quadratic multiplication. More efficient schemes can be developed [24], [25]. Tables 7 and 5 indicate that using two different FPGAs of the same family can result in a large speed difference while the difference in area is small.

In Table 7, a number of high-speed ECPs are compared with the proposed one. The hardware in [5] uses normal bases. Four FF multipliers are utilized in parallel, and addition and squarings are performed in series with multiplications. Each multiplication is performed in $5 + \lceil m/p \rceil$ clock cycles, where p is the degree of parallelism and may be considered as equivalent to w. The maximum p, where the design has been implemented and can fit in a Xilinx XC2V6000 FPGA, is 16. This design is using a composite field that is, however, avoided in some cryptographic systems for security reasons [16]. On the other hand, more efficient schemes are available for FF multiplication over composite fields. The scalar multiplier in [1] is not included in the table above. This is because the scalar multiplier in [1] uses a multiplier for squaring, which increases the total number of multiplications, but it is necessary for the creation of atomic blocks. It also uses two multipliers, which means a larger hardware. Considering an implementation in [1] for a 160-bit scalar (i.e., $m = 160$) and window-based NAF with a window size of four, an elliptic curve scalar multiplication over $GF(2^m)$ takes $1152(M + A + A + 0) \approx 87(m-1)$ clock cycles (the cost of multiplication, addition, and additive inversion is assumed to be six, three, and zero clock cycles, respectively).

In the scalar multiplication algorithm, a conversion from projective to affine coordinates, which includes an FF inversion, is required at the end of the loop (i.e., `Mxy(.)`). Using the Itoh-Tsujii algorithm, it takes $(m-1) + M(\lfloor \log_2(m-1) \rfloor + h(m-1) - 1)$ clock cycles for an inversion. For parameters pertaining to ECC applications, the field-multiplication-related latency term, i.e., $M\lceil \log_2(m-1) \rceil + h(m-1) - 1)$ is smaller than $(m-1)$ and is almost negligible compared to the large number of clock cycles required for a complete scalar multiplication less the conversion. In order to give a simple expression, the field multiplication latency in the inversion is neglected in the total number of clock cycles for a scalar multiplication, as shown in the fourth column in Table 7. If the extended euclidean algorithm is used, an inversion takes $2m$ clock cycles [3]. It can be concluded from the

TABLE 7
Performance of the Scalar Multipliers

Design	m	w	#Clk for kP Cycle	Clk MHz	kP μS	Xilinx FPGA	HW resource LUT, FF	Efficiency Throughput/#LUTs
[10][a] (2000)	167	42	$47(m-1)$ (est.)	76	210	XCV400E	3002, 1769 $\times$ 4	0.40
[3][b] (2002)	163	41	$57(m-1)$ (est.)	66.5	143	XC2V2000	19508, 6442	0.36
[9] (2003)	163	41	$93(m-1)$ (est.)	66	233	XC2V2000	10017, 1930	0.43
[11][c](2004)	233	-	$44(m-1)$ (est.)	100	132	XC2V6000	19440, 16970	0.39
[5][d] (2005)	162	16	$35(m-1)$	54	110	XC2V6000	34000, 3400 (est.)	0.27
This work[e]	163	41	$25(m-1)$	100 (128 P&R)	41	XC2V2000	7559, 2059 (4192 slices)	3.23
This work[f]	163	41	$25(m-1)$	100 (122 P&R)	41	XC2V2000	6095, 2398 (3416 slices)	4.00

[a]In [10], the maximum value of w is 16. For our comparison, we have scaled it up to $w = 42$.
[b]The large size is due to its scalability.
[c]For [11], we have assumed NAF representation for scalar k.
[d]See text.
[e]Using conventional quadratic complexity algorithm for 163×41 bit multiplication.
[f]Using the Karatsuba algorithm for 40×40 bit multiplication.

table that the proposed scheme outperforms other systems in terms of the number of clock cycles, as well as the critical path.

The last column in the table shows the algorithmic efficiency defined as throughput/area. It would be more accurate to use throughput/#slices, but slice counts were not reported by the authors of other designs. Therefore, we have used throughput/#LUTs.

4.6 Comments

- *Security against SPA attack.* The Montgomery algorithm is considered to be inherently more resistant against SPA and timing attacks. This is because the computation cost does not depend on the specific bit of the scalar k. For each bit of the scalar, one point addition and one doubling are performed. The proposed scheme has two different execution paths depending on the current bit of the scalar k. Both execution paths have the same complexity and take the same number of clock cycles. If the attacker is not able to separate swapping operation in the whole process, it is expected that the new scheme has the same level of resistance against SPA attacks.
- *Simple swapping.* In order to keep the swapping operation simple, parameters X_1, X_2, Z_1, and Z_2 are stored in a register file. Therefore, this operation is performed by merely swapping the address information, this does not take any additional clock cycles. A simple scheme for swapping is shown in Fig. 7, which can be considered equivalent to the switching of maximum two gates. The swap signal is generated by the condition in Algorithm 2.
- *Modular construction.* Fig. 3 can be used to derive a straightforward architecture. The basic building block is composed of an adder, a squarer, a multiplier, and a set of registers that hold the output of these units and a data path control unit. The output of the registers goes to the data path control unit. It arranges the data for the next round of arithmetic operation. It takes M_p clock cycles for data to be processed in the basic building block. One can cascade m building blocks to construct a pipeline system that outputs the result of one scalar multiplication every M_p clock cycles with a latency of mM_p clock cycles.
- *Designing for future FPGAs and multiple multipliers.* A WS multiplier with $m/w = 1$, i.e., a fully bit-parallel multiplier, occupies almost the same area as four multipliers with $m/w = 4$ while it performs the multiplication in one clock cycle instead of four clock cycles. This suggests that one large multiplier may be used instead of multiple parallel multipliers. Using multiple multipliers, as used in [5], has the disadvantage that we might not always be able to use all multipliers simultaneously or apply a pseudopipelining technique, which could result in higher time-space complexity. If $m/w = 1$, based on the flowgraph in Fig. 3, a scalar multiplication may be performed in about $6(m-1)$ clock cycles. The problem is that for a large m, such design does not fit in a single FPGA chip that is presently available commercially.

5 Conclusion

A high-performance scalar multiplication scheme based on the Montgomery scalar multiplication algorithm has been

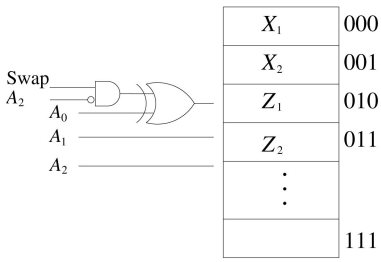

Fig. 7. Swapping mechanism.

proposed. The proposed scheme has been optimized to be dependent only to the number of FF multiplications. It hides the multiplier's overhead and prevents a pipeline multiplier from stalling.

A pseudopipelined WS multiplier suitable for the scheme has been introduced. The underlying FF multiplier performs loading and unloading of operands for the next operation while it performs the multiplication, and hence, a field multiplication takes $\lceil m/w \rceil$ clock cycles, where w is the word size. The gate delay in the critical path of the multiplier is $T_{AND} + \lceil \log_2(w/k) \rceil T_{XOR}$, which is smaller than that of conventional WS multipliers.

The proposed scheme performs a scalar multiplication in $25(m-1)$ clock cycles, which is approximately 2.75 times faster than a straightforward implementation and 1.6 times faster than the best implementations reported in this category in the open literature.

APPENDIX

FUNCTIONS IN ALGORITHM 1

Assume that E is a nonsupersingular elliptic curve over $GF(2^m)$ defined as $y^2 + xy = x^3 + ax^2 + b$ and $P = (x, y) \in E(GF(2^m))$. Functions `Madd(.)`, `Mdouble(.)`, and `Mxy(.)` in Algorithm 1 are defined as follows [19]: In these functions, global variables x and y are the coordinates of the original point P, which are fixed during the computation of kP, and x_k and y_k are the coordinates of $Q = kP$.

function `Mdouble` (**input** X_1, **input** Z_1)
{ See (8) in [19]
$X \leftarrow X_1^4 + b \cdot Z_1^4$
$Z \leftarrow Z_1^2 \cdot X_1^2$
return (X, Z)
}
function `Madd` (**input** X_1, **input** Z_1, **input** X_2, **input** Z_2)
{ See (9) in [19]
$Z \leftarrow (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2$
$X \leftarrow x \cdot Z + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1)$
return (X, Z)
}
function `Mxy` (**input** X_1, **input** Z_1, **input** X_2, **input** Z_2)
{
$x_k = X_1/Z_1$
$y_k = (x + x_k)[(y + x^2) + (X_2/Z_2 + x)(X_1/Z_1 + x)] \times (1/x) + y$
return (x_k, y_k)
}

ACKNOWLEDGMENTS

The first author wishes to thank Prof. Ingrid Verbauwhede of the University of California, Los Angeles, for her support. This work was supported in part by US National Science Foundation grant CCF-0541472 awarded to Dr. Verbauwhede and by NSERC through grants awarded to Dr. Hasan. Parts of this work were done when the first author was with the Department of Electrical and Computer Engineering, University of Waterloo.

REFERENCES

[1] P.K. Mishra, "Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems," *IEEE Trans. Computers,* vol. 55, no. 8, pp. 1000-1010, Aug. 2006.
[2] A.K. Daneshbeh and M.A. Hasan, "Area Efficient High Speed Elliptic Curve Cryptoprocessor for Random Curves," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC '04),* vol. 2, pp. 588-593, 2004.
[3] N. Gura, S.C. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila, "An End-to-End Systems Approach to Elliptic Curve Cryptography," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '02),* B.S. Kaliski, Ç.K. Koç, and C. Paar, eds., pp. 349-365, 2002.
[4] T. Izu and T. Takagi, "Fast Elliptic Curve Multiplications with SIMD Operations," *Proc. Fourth Int'l Conf. Information and Comm. Security (ICICS '02),* R.H. Deng, S. Qing, F. Bao, and J. Zhou, eds., pp. 217-230, 2002.
[5] R.C. Cheung, N.J. Telle, W. Luk, and P.Y. Vjeung, "Customizable Elliptic Curve Cryptosystems," *IEEE Trans. VLSI Systems,* vol. 13, no. 9, pp. 1048-1059, 2005.
[6] B. Chevallier-Mames, M. Ciet, and M. Joye, "Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity," *IEEE Trans. Computers,* vol. 53, no. 6, pp. 760-768, June 2004.
[7] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede, "Superscalar Coprocessor for High-Speed Curve-Based Cryptography," *Proc. Eighth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '06),* L. Goubin and M. Matsui, eds., pp. 415-429, 2006.
[8] H. Wu, "Bit-Parallel Finite Field Multiplier and Squarer Using Polynomial Basis," *IEEE Trans. Computers,* vol. 51, no. 7, pp. 750-758, July 2002.
[9] J. Lutz and M.A. Hasan, "High Performance FPGA Based Elliptic Curve Cryptographic Co-Processor," *Proc. Int'l Conf. Information Technology: Coding and Computing (ITCC '04),* vol. 2, pp. 486-492, 2004.
[10] G. Orlando and C. Paar, "A High Performance Reconfigurable Elliptic Curve Processor for GF(2^m)," *Proc. Second Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '00),* Ç.K. Koç and C. Paar, eds., pp. 41-56, 2000.
[11] C. Grabbe, M. Bednara, J. von zur Gathen, J. Shokrollahi, and J. Teich, "A High Performance VLIW Processor for Finite Field Arithmetic," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* pp. 189-194, 2003.
[12] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n)," *Proc. Fourth Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '02),* B.S. Kaliski, Ç.K. Koç, and C. Paar, eds., pp. 381-399, 2002.
[13] C. Huang, J. Lai, J. Ren, and Q. Zhang, "Scalable Elliptic Curve Encryption Processor for Portable Application," *Proc. Fifth Int'l Conf. ASIC,* vol. 2, pp. 1312-1316, Oct. 2003.
[14] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor," *Proc. Eighth IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '00),* pp. 68-76, 2000.
[15] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms," *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2002.
[16] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curves Cryptography.* Springer, 2003.
[17] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography.* Cambridge Univ. Press, 2002.
[18] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases," *Information and Computation,* vol. 78, no. 3, pp. 171-177, 1988.
[19] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation," *Proc. First Int'l Workshop Cryptographic Hardware and Embedded Systems (CHES '99),* Ç.K. Koç and C. Paar, eds., pp. 316-327, 1999.
[20] A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *IEEE Trans. Computers,* vol. 52, no. 4, pp. 449-460, Apr. 2003.
[21] B. Ansari and M.A. Hasan, "High Performance Architecture of Elliptic Curve Scalar Multiplication," technical report, Univ. of Waterloo, http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-01.pdf, Jan. 2006.

[22] P.L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Math. of Computation,* vol. 48, pp. 243-264, 1987.
[23] D. Catalano, R. Cramer, I. Damgard, G.D. Crescenzo, D. Pointcheval, and T. Takagi, *Contemporary Cryptology.* Birkhäuser Basel, 2005.
[24] P.L. Montgomery, "Five, Six, and Seven-Term Karatsuba-Like Formulae," *IEEE Trans. Computers,* vol. 54, no. 3, pp. 362-369, Mar. 2005.
[25] A. Weimerskirch and C. Paar, *Generalizations of the Karatsuba Algorithm for Efficient Implementations,* Cryptology ePrint Archive, Report 2006/224, http://eprint.iacr.org/, 2006.

Bijan Ansari received the BS degree from Isfahan University of Technology, Isfahan, Iran, in 1989 and the MSc degree in electrical engineering from the University of Windsor, Canada in 2004. He was a PhD student at the University of Waterloo, Canada, from 2004 to 2006 and is currently a PhD candidate in the Electrical Engineering Department at the University of California, Los Angeles. He has worked as a hardware and embedded software engineer and has served as a lecturer for several educational missions for the United Nations since 1990. His research interests include finite field arithmetic and VLSI architectures and algorithms for cryptography.

M. Anwar Hasan received the BSc degree in electrical and electronic engineering and the MSc degree in computer engineering from the Bangladesh University of Engineering and Technology in 1986 and 1988, respectively, and the PhD degree in electrical engineering from the University of Victoria in 1992. From April to December of 1992, he was a postdoctoral fellow at the University of Victoria. In January 1993, he became an assistant professor in the Department of Electrical and Computer Engineering, University of Waterloo. He was promoted to the rank of associate professor with tenure in 1998 and then to a full professor in 2002. At the University of Waterloo, he is also a member of the Centre for Applied Cryptographic Research, the Centre for Wireless Communications, and the VLSI Research Group. He is a recipient of the Raihan Memorial Gold Medal. At the University of Victoria, he received the President's Research Scholarship four times. At the University of Waterloo, he received a Faculty of Engineering Distinguished Performance Award in 2000 and an Outstanding Performance Award in 2004. He has served on the program and executive committees of several conferences. From 2000 to 2004, he was an associate editor of the *IEEE Transactions of Computers*. He is a licensed professional engineer of Ontario. He is a senior member of the IEEE.