High Performance Architecture of Elliptic Curve Scalar Multiplication

Bijan Ansari and M. Anwar Hasan
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
{bansari, ahasan}@uwaterloo.ca

Abstract

A high performance architecture of elliptic curve scalar multiplication over finite field $GF(2^m)$ is proposed. A pseudo-pipelined 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 $T_{AND} + (\log_2 w)T_{XOR}$, where T_{AND} and T_{XOR} are delays due to two-input AND and XOR gates respectively.

Index Terms

Scalar multiplication, elliptic curves, finite fields

I. 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]–[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. One way to achieve parallel and pipelined scalar multiplication is to decompose ECADD and ECDBL operations into FF operations, which results 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] to achieve parallelism and/or pipelining in the scalar multiplication operation. In [4], finite field operations are optimized for single instruction multiple data (SIMD) architecture. In [2], such a grouping has been used for obtaining pipelining and systolic operation. In [1], the sequence of operations is divided into a collection of uniform (similar) atomic blocks, where each block consists of a series of finite field 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 have been used in [5] as a low cost solution to achieve immunity against simple power analysis attacks (SPA) on the scalar multiplication.

Grouping of finite field operations is a key factor in the implementation of parallel and/or pipelined algorithms. Among the finite field 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 generally becomes comparable to that of addition [3], [6], [7]. In practice, FF squaring, like FF addition, can be performed in one clock cycle. Therefore a different approach for optimization is needed.

Finite field multiplication is the bottleneck of scalar multiplication, specially using projective coordinates. Most high speed elliptic curve processors (ECP) in $GF(2^m)$ use a word serial (WS) finite field multiplier, either in direct form [3], [8] or in the Karatsuba form [3], [9], [10]. Assuming the field size of 2^m elements and the word size of w bits a typical, WS multiplication algorithm is 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 $M(m-1)\lceil m/w \rceil$ clock cycles, where M is the number of FF multiplications in one iteration of the scalar

TABLE I

TYPICAL NUMBER OF CLOCK CYCLES OF BASIC FINITE FIELD OPERATIONS

Design	m	Multiplication	Addition	Squaring
[8]	167	7	3	3
[3]	163	7 to 12	3	3
[9]	233	9	≥ 2	2 (est.)
[7]	163	7	3	2

multiplication loop. However, this level of performance has not been reported in the literature yet and the main reasons are the followings:

1) In hardware implementation of WS multipliers, a few extra clock cycles are spent on loading inputs and unloading outputs [3], [7], [8]. This leads to a total of $\lceil m/w \rceil + c$ clock cycles in practice. Typically, the value of c is 3 [3], [7], [8] for elliptic curves that are of practical interest. For other finite field arithmetic units, like adder and squarer, extra clock cycles are spent to transfer data to and from memory/register file as well. Table I compares the execution times of these operations in terms of clock cycles as reported in various articles.
2) For high speed hardware implementation of operations of $GF(2^m)$ the execution time of addition and squaring are comparable to that of multiplication and may not be ignored (Table I).
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 has to stay idle until the other unit releases the bus. This could lead to a large number of idle cycles for the processing units [7].
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 depends 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 finite field 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 finite field multiplier to become idle 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 pseudo-pipeline WS finite field 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], [7]. A decrease in the number of clock cycles from 7 to 4 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 access relevant variables in parallel with finite field computations.

The organization of the remainder of this report is as follows. Section II briefly reviews the Montgomery scalar multiplication algorithm. There are data dependencies in the steps of the algorithm and hence the latter cannot be readily executed in pipelined fashion as desired. In Section III, we develop a pipelined version of the scalar multiplication scheme. In Section IV we explain an architecture of a finite field multiplier suitable for the proposed scalar multiplication scheme. In this section, implementation issues are also considered and some results for ASIC and FPGA implementations are presented. Finally, concluding remarks are given section V.

II. Review of the Montgomery Scalar Multiplication

Points on an elliptic curve E, defined over a finite field $GF(q)$, 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 $GF(q)$, the

number of points in $E(GF(q))$ is called the order of E over $GF(q)$, denoted by $\#E(GF(q))$. For cryptographic applications $\#E(GF(q)) = rh$ where r is prime and h is a small integer and P and Q have order r. Scalars such as k are random integers where $1 < k < r - 1$. Since $r \approx q$, the binary representation of $k = \sum_{i=0}^{n-1} k_i 2^i$ has n bits where $n \approx m = \lceil \log_2 q \rceil$. Scalar multiplication is the most dominant computation part of elliptic curve cryptography. More on this can be found in [11], [12].

Algorithm 1 shows the Montgomery [13] scalar multiplication scheme for non-supersingular elliptic curves over binary fields as it was introduced in [14]. In this algorithm `Madd(`X_1`,`Z_1`,`X_2`,`X_2`)`, `Mdouble(`X_1`,`Z_1`)` and `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 computation involved in these functions can be found in the appendix. The reader is referred to [11], [14] for 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: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$ {calculate P and $2P$}
2: **if** ($k = 0$ **or** $x = 0$) **then**
3: $\quad x \leftarrow 0, y \leftarrow 0$
4: $\quad$ **stop**
5: **end if**
6: **for** $i = n - 2$ **to** 0 **do**
7: $\quad$ **if** $k_i = 1$ **then**
8: $\quad\quad (X_1, Z_1) \leftarrow$`Madd(`$X_1, Z_1, X_2, Z_2$`)`, $(X_2, Z_2) \leftarrow$`Mdouble(`X_2, Z_2`)`
9: $\quad$ **else**
10: $\quad\quad (X_2, Z_2) \leftarrow$`Madd(`$X_2, Z_2, X_1, Z_1$`)`, $(X_1, Z_1) \leftarrow$`Mdouble(`X_1, Z_1`)`
11: $\quad$ **end if**
12: **end for**
13: $Q \leftarrow$`Mxy(`X_1, Z_1, X_2, Z_2`)`
14: **return** Q

This algorithm has been used in several high speed ECC implementations [3], [8], [15]. For a straight-forward 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 finite field and m is the dimension of the binary extension field $GF(2^m)$.

III. Architecture for Scalar Multiplication

Since finite field multiplier is the bottle neck of scalar multiplication, it requires special consideration for realizing a high performance architecture for scalar multiplication. Consider a word serial finite field multiplier. It can be divided into two functional units: the multiplication *core* and the input/output buffers. When data is being loaded to the input buffer or the result is unloaded from the output buffer, the multiplier core is essentially idle. 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. This can be achieved by performing a field addition and a squaring in parallel with a field multiplication. For this the combined execution time for the addition and squaring is assumed to be less than or equal to that of multiplication. Since the multiplier is a finite state machine and performs the multiplication in a certain number of clock cycles, the multiplier should be fed with data in equal pace. This is addressed in Sections III-B and III-C.

Our another 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. This means when one multiplication is performed, data for the next multiplication should be available to the multiplier on time. Additionally, the end of one iteration in the scalar multiplication loop be tied properly to the start of the next iteration. This needs to be done carefully since the next iteration uses the result of the previous iteration (Section III-D).

A. *Merging of Two Execution Paths*

In Algorithm 1, depending on the value of k_i, either line 8 or line 10 is executed. The operations are the same in both paths, but the inputs and the outputs of `Madd(.)`and `Mdouble(.)` functions are different. In order to keep the algorithm uniform and suitable for pipelining we merge the two k_i dependent execution paths in Algorithm 1. Since point addition is commutative,

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

the inputs to `Madd(.)` function affectively remain the same. The output variable however depends on the value of k_i. In the case of `Mdouble(.)`, input and output variables depend on k_i. It is sufficient to swap X_1, with X_2 and Z_1 with Z_2 before and after any calculation, if k_i equals to one. Doing so, the input to `Mdouble(.)` remains the same but the output goes to X_2, Z_2 instead. Input and output variables of `Mdouble(.)` are changed to X_2 and Z_2 accordingly. After calculation, the variables need to be swapped back to their original states. If two consecutive bits are one, then a pair of swapping can be eliminated. This is shown in Algorithm 2.

In hardware, when 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. Swapping does not take any clock cycles. A swap signal can be generated using the current state of i and k_i. It can then be applied to the address logic of the register file.

B. Parallel Execution

If the finite field 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. There are ways (see for example [3]) to improve the performance by using parallel operations. To this end, one can simply use one multiplier, one adder, and one squaring unit. Figure 1 depicts the flow graph of scalar multiplication algorithm in which each multiplication is performed in parallel with an addition and/or with a squaring. We assume that multiplication takes longer than addition and squaring. This enables us to make the critical path of the scalar multiplication operation dependents only on the *finite field multiplication*. Using this algorithm the execution time for one iteration in the scalar multiplication loop is equal to $6M + A$. In Fig. 1 the dashed line shows the critical path of the algorithm, which is dependent on M as long as $M > A$ and $M > S$.

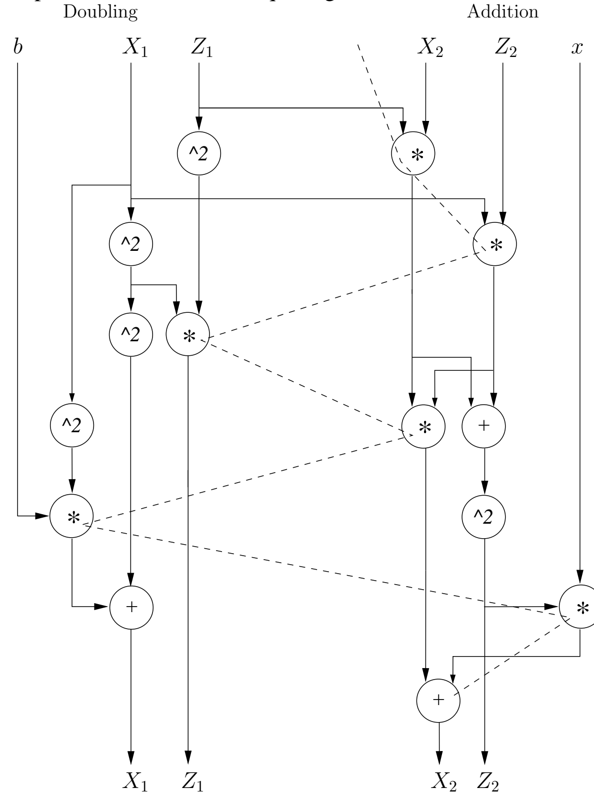
Algorithm 2 is very flexible. If enough hardware resources are available two multiplications can be performed in parallel. This is shown in Fig. 2. This architecture can reduce the execution time to $3M + A$ clock cycles, assuming the input operands to the multipliers are available at appropriate clock cycle. This architecture however, almost doubles the hardware size and is not considered further in this work.

C. Data Dependency at Transitions of Iterations

Let $M = M_p + c$ be the number of clock cycles needed for a finite field multiplication, where $M_p (= \lceil m/w \rceil$ for WS multiplier) is the number of clock cycles needed to calculate the result and c is the total clock cycles needed to load the input and unload the result from the multiplier. We call c as the *idle* time of the multiplier core. In the flowgraph shown in Fig. 1 performance can be improved if another multiplication can start while the multiplier core is in the idle state.

In order to prevent the multiplier to become idle new operands need to be fed to the multiplier at the rate of M_p. However, for an idle free operation one needs to make certain that the next multiplication is not dependent on the current one. The flowgraph of scalar multiplication in Fig. 3

Fig. 1. Parallel execution of multiplication and addition/squaring

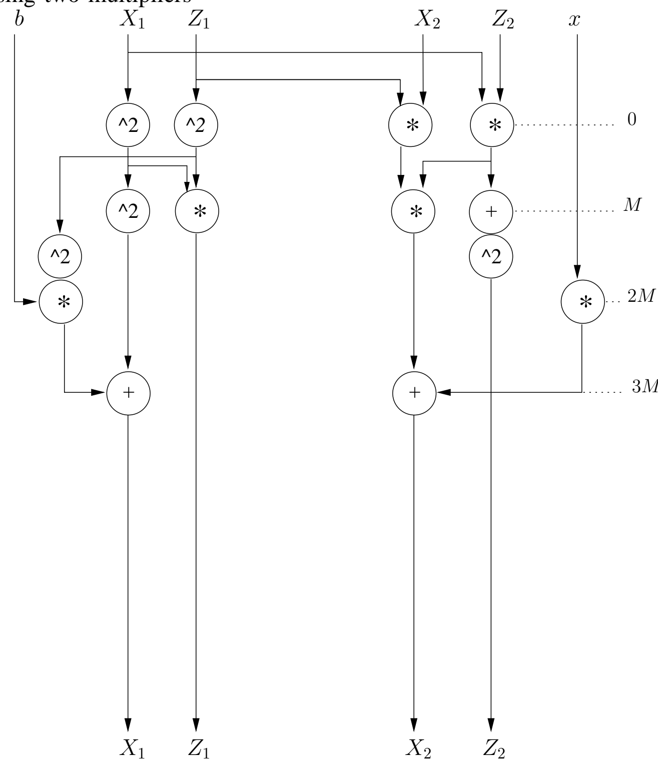


assumes a multiplier with a computation time of M_p clock cycles and a total multiplication time of M clock cycles, where $M = M_p + c$.

In Fig. 3, each $\circledast$ (circle) corresponds to the start of a finite field multiplication. Vertically below each circle, there is a triangle to indicate the end of the multiplication that originated at the circle. 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 event in the flow graph. We assume that $M_p > A$ and $M_p > S$. In the flowgraph the multiplier receives its operands regularly and at equal intervals. Using this scheme the total execution time equals to $5M_p + M + A$ clock cycles.

The dashed lines through the circles in Fig. 3 indicate the time up to which operands at the input of multiplier are intact and may be used for other operations. Modification to the register happens at the triangle event. Near the bottom of flowgraph of Fig. 3, the adder needs to wait until the multiplier computes the field multiplication and the output of the adder would be the

Fig. 2. Parallel execution using two multipliers

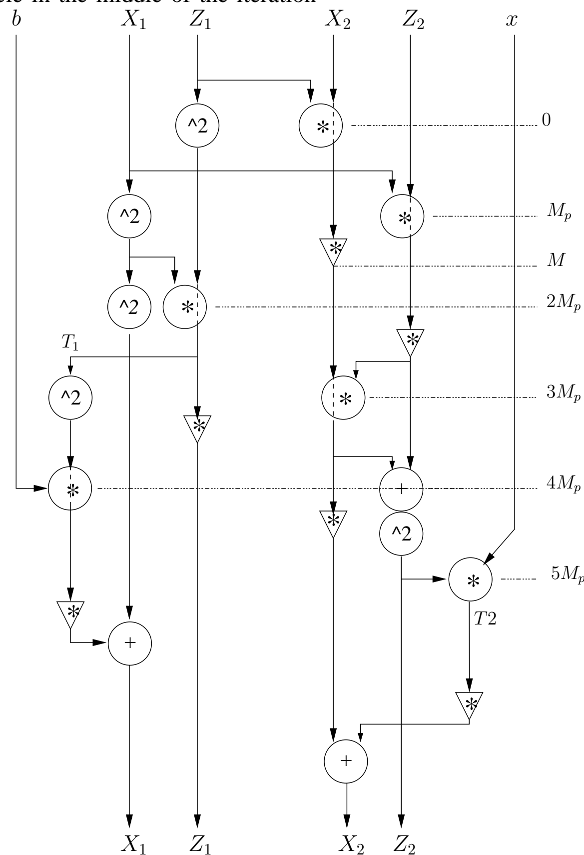


next set of inputs to the multiplier. This causes a delay of $(M - M_p + A)$ clock cycles per iteration, which in turn translates into an overall delay of $(m-1)(M - M_p + A)$ clock cycles in the scalar multiplication operation. This problem can be eliminated as follows.

D. Resolving Data Dependency at Iteration Transitions

As shown in Fig 1, the first multiplication in the next iteration 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 the triangle event in the last finite field multiplication in the flowgraph Fig. 3. If $k_i = 0$, we may start the next iteration at the triangle event by the $X_1 * Z_2$ operation. If $k_i = 1$ the variables are swapped; 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. 4. A switch box is added at the end (or start) of the flowgraph which swaps the registers properly. It does not take extra clock cycles since the logic is simple and can be done by combinational logic. One addition

Fig. 3. Parallel with no idle cycle in the middle of the iteration

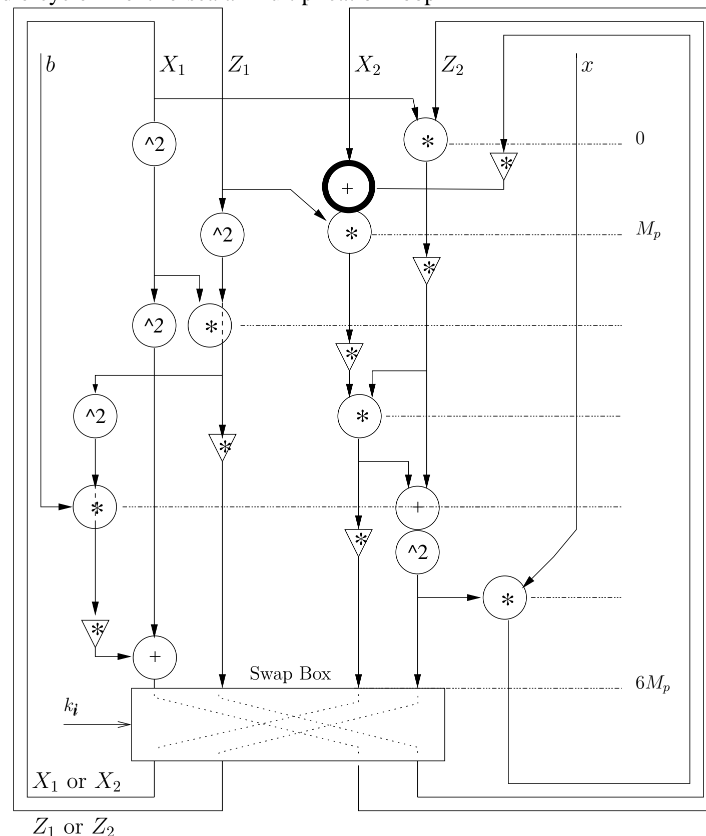


operation, from the end of the previous iteration appears at the start of the next iteration. This is highlighted in Fig. 4 with a bold faced circle. As it is shown, each iteration takes $6M_p$ clock cycles and the multiplier does not become idle.

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. A finite field multiplier suitable for this scheme is proposed in section IV-B.

Table II summarizes and compares the speed of scalar multiplication operation as mapped on to the flowgraph of Fig. 1, Fig. 3 and Fig. 4. For finite field operations indicated in the flowgraph, high speed architectures similar to [3], [7]–[9] are assumed. In these architectures one typically has $A = S = 3$ and $M = \lceil m/w \rceil + 3 = 7$. The first row in Table II serves as a basis of comparison and corresponds to a straight-forward hardware implementation of Algorithm 1.

Fig. 4. Parallel with no idle cycle in entire scalar multiplication loop



IV. IMPLEMENTATION

The number of clock cycles by itself does not show the speed of the system, since the clock rate may vary considerably. Therefore an implementation is carried out to verify the performance of the system. Traditional elliptic curve processors are based on an instruction set which allows them to execute different scalar multiplication schemes [3], [7], [8].

A. Implemented Architecture

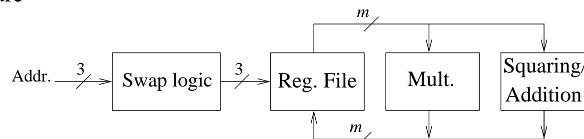
The proposed scalar multiplication scheme is highly optimized toward the execution of the Montgomery ladder in projective coordinates. Therefore, it is implemented in the form of a state machine. Figure 5 shows the basic architecture of the execution unit, which consists of a squaring/addition unit, a finite field multiplier, a dual port $8 \times m$ bit register file and an address swapping logic. An FF addition or squaring, an FF multiplication and a load/save operation

TABLE II

SUMMARY, ASSUMING $\lceil m/w \rceil = 4, A = S = 3$

Method	#Clks in one iteration	#Clks in $(m-1)$ iterations	Speed
Straight-forward (Alg. 2)	$6M + 3A + 5S$	$66(m-1)$	1.00
Parallel addition/squaring (Fig. 1)	$6M + A$	$45(m-1)$	1.47
No idle cycle for the FF multiplier (Fig. 3)	$5M_p + M + A$	$30(m-1)$	2.13
No idle cycle in the entire operation (Fig. 4)	$6M_p$	$24(m-1)$	2.75

Fig. 5. Implemented Architecture

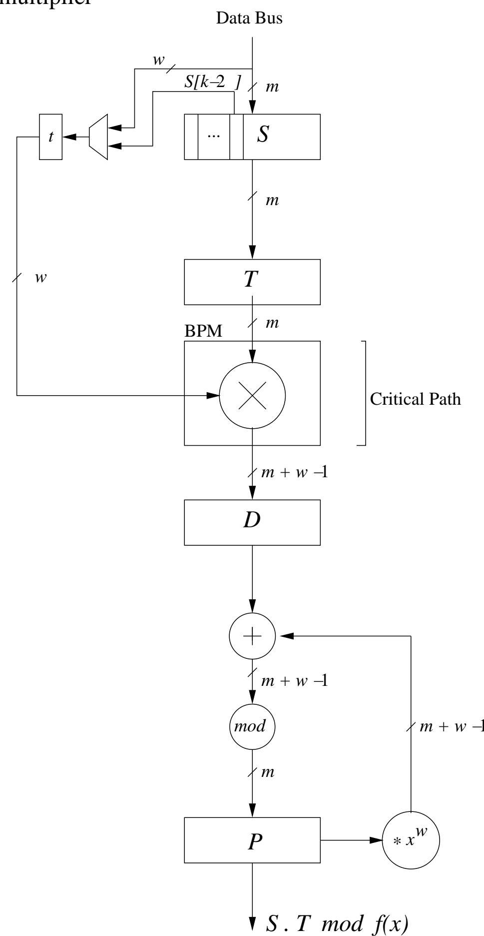


from/to the register file can be performed in parallel. A squaring, for example, is performed in 3 clock cycles – one for each of the following operands: loading the accumulator with data from the register file, squaring and finally saving the result in the register file. The multiplier and the squaring/addition unit can be loaded with the same data at the same clock cycle, to prevent redundant data transfer in the data bus.

B. Pseudo-Pipelined WS Finite Field Multiplier

In this section a pseudo-pipelined finite field multiplier is introduced which can be used in the proposed scalar multiplication scheme. A polynomial basis representation of the field is assumed. In Fig. 6 four registers S, T, D and P construct the interstage buffers of the pipeline. The multiplier operands are fed sequentially to the multiplier. This arrangement provides the multiplier with one more level of delay/pipelining which allows us to transfer one of the operands into the multiplier while the multiplier is still busy performing the previous multiplication. i.e. the inputs to the $m \times w$-bit bit-parallel multiplier (BPM) finite field multiplier, shown as $\otimes$ in Fig. 6, remain unchanged after transferring the first operand to the multiplier. Another benefit is that at some point in the scalar multiplication algorithm, the addition/squaring unit and the multiplier use the same operand (see Fig. 3). This arrangement provides a good mechanism to load both units at the same time and save one clock cycle. The multiplier is controlled by its

Fig. 6. Pseudo-pipelined finite field multiplier



own state machine.

Register S is arranged like a stack of $\lceil m/w \rceil$ words with w bits in each word. Therefore no multiplexer is needed for word selection, as it is used in conventional WS multipliers. This reduces the gate delay in the critical path to $T_{AND} + (\log_2 w)T_{XOR}$ resulting from the multiplication of the contents of registers T and t. This is apparently the shortest critical path in WS multipliers reported in the literature, especially when the field defining irreducible polynomial is known in advance and hence reduces the complexity of the "mod" operation shown in the multiplier structure.

The operation of the multiplier for the special case of $\lceil m/w \rceil = 4$ is presented in Table III. 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

TABLE III

STATE DIAGRAM OF THE PSEUDO-PIPELINED FINITE FIELD MULTIPLIER

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

similarly $U = U_3x^{3w} + U_2x^{2w} + U_1x^w + U_0$. One can see from the table that each multiplication takes 7 cycles. The multiplier has a pipeline rate of 4, i.e. after every 4 clock cycles a new set of input operands can start entering the multiplier.

As stated before, in theory, WS finite field multiplication algorithm takes $\lceil m/w \rceil$ iterations or clock cycles. However, the operation of loading the inputs and unloading the output occupies the data bus and takes a few extra clock cycles([3], [7], [9]). The key to the fast execution of scalar multiplication is to perform loading and unloading in parallel with the finite field computations, namely addition, multiplication and squaring. As an example, consider the execution of $Z_1 = Z_1 \times X_1$ 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: $S \leftarrow Z_1, ACC \leftarrow ACC^2$ {*the content of S will be pushed automatically to T by the multiplier state machine. The squaring is performed in one cycle.*}

3: $ACC \leftarrow Z_1, X_1 \leftarrow ACC$ {*Save the result in ACC. Load ACC with the next operand. Multiplier is 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: {*The result of multiplication is ready. i.e.* $P \leftarrow Z_1 * X_1$ }

C. Implementation Results

Using the proposed multiplier, the scalar multiplication scheme can be implemented in an architecture with the delay due to gates in the critical path equal to $T_{AND} + \log_2 w T_{XOR}$. Synthesized for $GF(2^{163})$ using with Synopsys Design Analyzer, the layout was analyzed with Cadence Encounter. The critical path equals to $6ns$ for $0.18\mu m$ CMOS technology. The delay in the critical path is caused by the wiring delay in the layout as well as gate delay. Using Xilinx ISE, the critical path equals to $10ns$ for Xilinx XC2V2000 FPGA. The system computes the scalar multiplication for curves over $GF(2^{163})$ in 21 and 41 μS on ASIC and FPGA respectively. The hardware takes about 36000 gates in CMOS $0.18\mu m$; on Xilinx XC2V2000 FPGA it takes 8300 lookup tables (LUT) and 1100 flip flops (FF) and 7 block RAM.

D. Comparison

In Table IV a number of high speed elliptic curve processors (ECP) are compared with the proposed one on the basis of number of clock cycles for scalar multiplication. The scalar multiplier of [1] is not included in the table above. This is because the scalar multiplier of [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 of [1] for a 160 bit scalar (i.e. $m = 160$) and r-NAF with $r = 4$, an elliptic curve scalar multiplication over $GF(2^m)$ takes $1152(M + A + A + 0) \approx 91(m - 1)$ clock cycles (the cost of additive inversion in a binary extension field is zero clock cycles).

In the scalar multiplication algorithm a conversion from projective to affine coordinates, which includes a finite field inversion, is required at the end of the loop (i.e. `Mxy(.)`). Using the Itoh-Tsujii algorithm it takes $(m - 1) + M(\lfloor \log_2(m - 1) \rfloor + h(m - 1) - 1) \approx (m - 1)$ clock cycles. When the extended Euclidean algorithm is used, inversion takes $2m$ clock cycles [3]. The middle column of Table IV gives the number of clock cycles that include the conversion and the $m - 1$

TABLE IV

PERFORMANCE OF THE SCALAR MULTIPLIERS

Design	$\lceil m/w \rceil$	Number of Clk for kP	Point Representation
[9]	-	$44(m-1)$ (est.)	Projective with NAF [a]
[8]	4	$47(m-1)$ (est.)	Montgomery Projective, $w = 42$ [b]
[3]	4	$57(m-1)$ (est.)	Montgomery Projective
[7]	4	$93(m-1)$ (est.)	Projective with NAF
Proposed Scheme		$25(m-1)$	Montgomery Projective

[a]For [9], we have used NAF representation for scalar k.

[b]In [8], the maximum value of w is 16. For our comparison we have scaled it up to $w = 42$.

passes of the main loop of the scalar multiplication operation. The proposed scheme performs better than a parallel system because it is using a pseudo-pipeline multiplier effectively.

E. Comments

- Security Against Simple Power Analysis Attack (SPA): The Montgomery algorithm is considered to be inherently 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 have 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. 4 can be used to derive a straight-forward architecture. The basic building block is composed of an adder, a squarer, a multiplier, and a set of registers which hold the output of these units and a data path control unit (Fig. 8). The output of

Fig. 7. Swapping mechanism

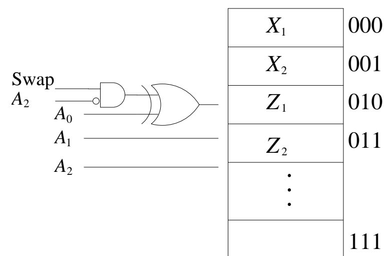
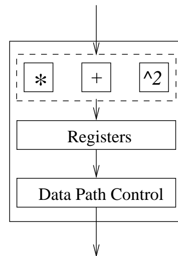


Fig. 8. Basic Building Block



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 which outputs the result of one scalar multiplication every M_p clock cycles with a latency of mM_p clock cycles.

V. Conclusion

A high performance scalar multiplication scheme based on the Montgomery scalar multiplication algorithm has been proposed. Using a pseudo pipelined word serial multiplier the scheme performs a scalar multiplication in $25(m-1)$ clock cycles, which is approximately 2.75 times faster than a straight-forward implementation and 1.6 times faster than best implementations reported in this category in the open literature. The underlying finite field multiplier performs loading and unloading of data while it is in operation and hence performs a field multiplication in $\lceil m/w \rceil$ clock cycles. The gate delay in the critical path of the multiplier is $T_{AND} + \log_2 wT_{XOR}$. Implemented on FPGAs, the scalar multiplication system operates at 100MHz and performs about 24000 scalar multiplications per second for curves over $GF(2^{163})$.

APPENDIX

Assume that E is a non-supersingular 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 [14]. In these functions, x and y are the coordinates of the original point P which are fixed during the calculation of kP and, x_k and y_k are the coordinates of $Q = kP$.

function `Mdouble` (**input** X_1, **input** Z_1)

{

$X \leftarrow X_1^4 + b \cdot Z_1^4$

$Z \leftarrow Z_1^2 \cdot X_1^2$

return (X, Z)

}

function `Madd` (**input** X_1, **input** Z_1, **input** Z_2, **input** Z_2)

{

$X \leftarrow (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2$

$Z \leftarrow x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1)$

return (X, Y)

}

function `Mxy` (**input** X_1, **input** Z_1)

{

$x_k = X_1/Z_1$

$y_k = (x + x_k)[(y + x^2) + (X_2/Z_2 + x)(X_1/Z_1 + x)] \times (1/x) + y$

return (x_k, y_k)

}

REFERENCES

[1] P. K. Mishra, "Pipelined computation of scalar multiplication in elliptic curve cryptosystems." in *CHES*, 2004, pp. 328–342.

[2] A. K. Daneshbeh and M. A. Hasan, "Area efficient high speed elliptic curve cryptoprocessor for random curves." in *ITCC (2)*, 2004, pp. 588–.

[3] H. Eberle, N. Gura, and S. C. Shantz, “A cryptograhpic processor for arbitrary elliptic curves over $GF(2^m)$.” in *ASAP*, 2003, pp. 444–454.

[4] T. Izu and T. Takagi, “Fast elliptic curve multiplications with simd operations.” in *ICICS*, 2002, pp. 217–230.

[5] 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, 2004.

[6] H. Wu, “Bit-parallel finite field multiplier and squarer using polynomial basis.” *IEEE Trans. Computers*, vol. 51, no. 7, pp. 750–758, 2002.

[7] J. Lutz and M. A. Hasan, “High performance fpga based elliptic curve cryptographic co-processor.” in *ITCC (2)*, 2004, pp. 486–492.

[8] G. Orlando and C. Paar, “A high performance reconfigurable elliptic curve processor for $GF(2^m)$.” in *CHES*. London, UK: Springer-Verlag, 2000, pp. 41–56.

[9] C. Grabbe, M. Bednara, J. von zur Gathen, J. Shokrollahi, and J. Teich, “A high performance vliw processor for finite field arithmetic.” in *IPDPS*, 2003, p. 189.

[10] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel, “A reconfigurable system on chip implementation for elliptic curve cryptography over $GF(2^m)$.” in *CHES*, 2002, pp. 381–399.

[11] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curves Cryptography.* Springer, 2003.

[12] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography.* Cambridge University Press, 2002.

[13] P. Montgomery, “Speeding the pollard and elliptic curve methods of factorization.” *Mathematics of Computation*, vol. 48, pp. 243–264, 1987.

[14] J. López and R. Dahab, “Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation.” in *CHES*, 1999, pp. 316–327.

[15] A. Satoh and K. Takano, “A scalable dual-field elliptic curve cryptographic processor.” *IEEE Trans. Computers*, vol. 52, no. 4, pp. 449–460, 2003.