Pipelined Computation of Scalar Multiplication in Elliptic Curve Cryptosystems

Pradeep Kumar Mishra

Cryptographic Research Group
Indian Statistical Institute
203 B T Road, Kolkata - 700108, INDIA
pradeep_t@isical.ac.in

Abstract. In the current work we propose a pipelining scheme for implementing Elliptic Curve Cryptosystems (ECC). The scalar multiplication is the dominant operation in ECC. It is computed by a series of point additions and doublings. The pipelining scheme is based on a key observation: to start the subsequent operation one need not wait until the current one exits. The next operation can begin while a part of the current operation is still being processed. To our knowledge, this is the first attempt to compute the scalar multiplication in such a pipelined method. Also, the proposed scheme can be made resistant to side-channel attacks (SCA). Our scheme compares favourably to all SCA resistant sequential and parallel methods.

Keywords: Elliptic curve cryptosystems, pipelining, scalar multiplication, Jacobian coordinates.

1 Introduction

Elliptic Curve Cryptosystems (ECC) were first proposed independently by Koblitz [16] and Miller [21] in 1985. The cryptosystem is based on the additive group of points on an elliptic curve over a finite field. It derives its security from the hardness of the elliptic curve discrete logarithm problem (ECDLP). For a carefully chosen curve over a suitable underlying field there is no subexponential time algorithm to solve ECDLP. This fact enables ECC to provide a high level of security with much smaller keys than RSA and primitives based on discrete logarithm problems on finite fields. However, this security does not come for free. The group operation in ECC is more complex than that of finite field based cryptosystems. This provides a strong motivation for the cryptographic community to work on ECC to make them more efficient.

The fundamental operation in ECC is scalar multiplication, namely, given an integer m and an elliptic curve point P, the computation of mP. It is computed by a series of doubling (DBL) and addition (ADD) operation of the point P, depending upon the bit sequence representing d. A plethora of methods have been proposed to perform the scalar multiplication in a secure and efficient way. For an excellent review see [10]. The performance of all these methods is dependent on the efficiency of the elliptic curve group operations: DBL and ADD. In the current work we will refer to them as EC-operations. EC-operations in affine coordinates involve inversion, which is a very costly operation particularly over prime fields. To avoid inversion various co-ordinate systems have been proposed in literature. In this work we will use Jacobian coordinates.

In the current work we describe a very general technique to compute the scalar multiplication. It can be applied to any scalar multiplication method that only uses doubling and addition (or subtraction), with or without precomputations.

The computation of scalar multiplication proceeds in a series of EC-operations. A key observation is that these operations can be computed in a pipeline, so that the subsequent operation need not wait till the current one exits. The ADD and DBL algorithms have their own set of inputs. These algorithms can be divided into parts some of which can be executed with only a part of the input. So one part of the algorithm can begin execution as soon as the corresponding part of the inputs is available to it. Thus two or more EC-operations can be executed in a pipeline.

In the current work we propose a two stage pipeline. At any point of time there will be at most two operations in the pipeline in a "Producer-Consumer Relation". The one which enters the pipeline earlier will be producing outputs which will be consumed by the second operation as inputs. As soon as the producer process exits the pipeline the subsequent EC-operation will enter the pipeline as the consumer process. The earlier consumer would be producing outputs now which will be consumed by the newer process.

Any processor capable of handling ECC must have capabilities (in hardware or software) for executing field arithmetic. It must have modules for field element addition, subtraction, multiplication and inversion. In computing the scalar multiplication using a co-ordinate system other than affine coordinates, only one field inversion is necessary. So the most important operations are addition and multiplication. In the pipelined architecture we need a multiplier and an adder for each of the pipe stages. The adder can be shared by the pipe stages as the addition operation is much cheaper in comparison to multiplication. Note that in this work, *we will consider a squaring as a multiplication.* However, this is not true in general.

Our method will also require slightly more memory. As two EC-operations will be computed simultaneously in two pipe stages more memory will be required for processing. This extra memory requirement is discussed in details in Section 5.

In [17], [18], Paul Kocher et al. proposed Side-channel attacks (SCA), which are considered to be the most potential threat against mobile devices and ECC. Many countermeasures against SCA have been proposed in literature. Almost all of them involve some computational overhead to resist SCA. One of the latest methods proposed in [3] involves the least amount of computational overhead to get rid of simple power analysis attacks (SPA). The authors divide each EC-operation into *atomic blocks* which are indistinguishable from the side-channel.

So, in this model the computation of scalar multiplication is a sequence of indistinguishable atomic blocks. We use a variant of this method to resist SPA. To resist DPA many standard methods can be incorporated to it.

In this work we have used the computation time of an atomic block as a unit of time. One atomic block has one multiplication, two additions and one negation. As computation time of an addition is quite small in comparison to that of a multiplication, computation time of an atomic block is approximately that of a finite field multiplication. A point addition (mixed coordinates) takes 11 atomic blocks and a doubling takes 10 atomic blocks. So they can be computed in 10 and 11 units of time respectively. In our pipelining scheme an EC-operation can be computed in 6 units of time. This leads to a significant improvement in performance (see Section 5). Furthermore, our scheme can be combined with windowing methods to obtain even faster scalar multiplication.

2 Background

In this section we briefly discuss the current state of affairs in ECC and side-channel attacks.

2.1 Elliptic Curve Preliminaries

There exists an extensive literature on elliptic curve cryptography. Here we only mention the results that we need, without proof. We refer the reader to [10] for details. In the current work we will concentrate on curves over large prime fields only. Over a finite field $F_q,\ q = p^r$ of odd characteristic $p > 3$, an elliptic curve has an equation of the form $y^2 = x^3 + ax + b$ where $a, b \in F_q$ and $4a^3 + 27b^2 \neq 0$. An elliptic curve point is represented using a pair of finite field elements. The group operations in affine coordinates involve finite field inversion, which is a very costly operations, particularly over prime fields [9]. To avoid these inversions, various co-ordinate systems like, projective, Jacobian, modified Jacobian, Chudnovsky-Jacobian have been proposed in literature [6]. We present our work in Jacobian coordinates, which are extensively used in implementations. In Jacobian coordinates, $(X : Y : Z)$ represents the point $(X/Z^2, Y/Z^3)$ on the curve.

The scalar multiplication is generally computed using a left-to-right binary algorithm.

Binary Algorithm (left-to-right) for scalar multiplication

```
Input: An integer $m = m_{n-1}2^{n-1} + \cdots + m_0, m_{n-1} \neq 0$ and a point $P$
Output: $mP$.
1. $P_0 = P$
2. for $i = 0$ to $n - 2$;
3.     $P_{i+1} = \mathrm{DBL}(P_i)$;
4.     if $m_{n-2-i} = 1$
5.         $P_{i+1} = \mathrm{ADD}(P_{i+1}, P)$;
6. Return $(P_{n-1})$
```

The ADD operation in Jacobian coordinates is much cheaper if the Z-coordinate of one point is 1. This operation is called mixed addition [6]. In implementations of scalar multiplication, the addition operation in Step 5 is generally a mixed addition.

The algorithm needs $n-1$ point doublings and on the average $n/2$ additions to compute the scalar multiplication. As computing the additive inverse of a given point is almost for free, addition and subtraction in the elliptic curve group have almost the same complexity. To reduce the complexity the *Non-adjacent form* (NAF) representation of the scalar multiplier has been proposed. In NAF, the coefficients of the representation belong to the set {0, 1, -1} and no two consecutive coefficients are non-zero. The number of non-zero terms in NAF representation is on the average $n/3$. Thus if the scalar multiplier is represented in NAF, the average number of point additions in the scalar multiplication algorithm reduces to $n/3$. To further reduce the number of additions the w-NAF representations have been proposed (see [22] [5]). With a precomputed table of size 2^{w-1} points, the number of additions comes down to $n/(w+1)$. The complexity of scalar multiplication is thus dependent on the efficiency of point addition and doubling. We discuss the ADD and DBL algorithm in Jacobian coordinates below.

If two points $P(X,Y,Z)$ and $P_i(X_i,Y_i,Z_i)$ are in Jacobian coordinates, then the double of P_i i.e. $2P_i = P_{i+1}(X_{i+1},Y_{i+1},Z_{i+1})$ is computed as:

$X_{i+1} = M^2 - 2S, Y_{i+1} = M(S - X_{i+1}) - 8Y_i^4$ and $Z_{i+1} = 2Y_iZ_i$, where $M = 3X_i^2 + aZ_i^4$ and $S = 4X_iY_i^2$.

The sum $P + P_i = P_{i+1}(X_{i+1},Y_{i+1},Z_{i+1})$ is computed as:

$X_{i+1} = W^3 - 2U_1W^2 + R^2, Y_{i+1} = -S_1W^2 + R(U_1W^2 - X_{i+1}), Z_{i+1} = ZZ_iW$, where $U_1 = XZ_i^2, U_2 = X_iZ^2, S_1 = YZ_i^3, S_2 = Y_iZ^3, W = U_1 - U_2, R = S_1 - S_2$.

Let $[a], [m]$ and $[s]$ denote the time required for one addition, multiplication and squaring in the underlying field respectively. Then, ADD has complexity $7[a] + 12[m] + 4[s]$ and DBL has complexity $11[a] + 4[m] + 6[s]$. In the current work, we do not distinguish between a multiplication and a squaring and neglect additions. So, roughly, we can say ADD involves 16 multiplications and DBL involves 10 multiplications. Mixed addition is quite cheaper, requiring only 11 multiplications.

2.2 Side-Channel Attacks and Side-Channel Atomicity

Side-channel attacks (SCA) are one of the most dangerous threat to ECC-implementations. Discovered by Paul Kocher et al. [17], [18] SCA reveals the secret information by sampling and analyzing the side-channel information like timing, power consumption and EM radiation traces. ECC is very suitable for mobile and hand held devices, which are used in hostile outdoor environments. Hence an implementation must be side-channel resistant. SCA's which use power consumption traces of the computation are called power attacks. Power attacks subsumes timing attacks [11]. They can be divided into simple power attacks

(SPA) and differential power attacks (DPA). Simple power attacks use information from one observation to break the secret. Differential power attacks use data from several observations and reveal the secret information by statistically analyzing them. Power analysis can be performed with very inexpensive equipments, hence the threat is real. Several countermeasures have been proposed in

Table 1. DBL Algorithm in Atomic Blocks

DBL Algorithm
Input: $P_i(X_i, Y_i, Z_i)$
Input: $P_i = (X_i, Y_i, Z_i)$
Output: $2P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$

Δ_1	$R_1 = T_8 \times T_8\ (Z_i^2)$ * * * *	Δ_6	$R_4 = T_7 \times T_7\ (Y_i^2)$ $R_2 = R_4 + R_4\ (2Y_i^2)$ $R_2 = R_4 + R_4\ (2Y_i^2)$ * *
Δ_2	$R_1 = R_1 \times R_1\ (Z_i^4)$ * * *	Δ_7	$R_4 = T_6 \times R_2\ (2X_iY_i^2)$ $R_4 = R_4 + R_4\ (S)$ $R_4 = -R_4\ (-S)$ $R_5 = R_4 + R_4\ (-2S)$
Δ_3	$R_1 = a \times R_1\ (aZ_i^4)$ * * *	Δ_8	$R_3 = R_1 \times R_1\ (M^2)$ $T_6 = R_3 + R_5\ (\underline{X_{i+1}})$ * $R_4 = T_6 + R_4\ (X_{i+1} - S)$
Δ_4	$R_2 = T_6 \times T_6\ (X_i^2)$ $R_3 = R_2 + R_2 (2X_i^2)$ * $R_2 = R_3 + R_2\ (3X_i^2)$	Δ_9	$R_2 = R_2 \times R_2\ (4Y_i^4)$ $R_2 = R_2 + R_2 (8Y_i^4)$ * *
Δ_5	$T_8 = T_7 \times T_8\ (Y_iZ_i)$ $T_8 = T_8 + T_8\ (\underline{Z_{i+1}})$ * $R_1 = R_1 + R_2\ (M)$	Δ_{10}	$T_7 = R_1 \times R_4\ (M(X_{i+1} - S))$ $T_7 = T_7 + R_2\ (-Y_{i+1})$ $T_7 = -T_7\ (\underline{Y_{i+1}})$ *

literature to guard ECC against SPA and DPA (see [2], [7], [11], [15], [4] for example). Almost all of them need some computational overhead for the immunization. *Side-Channel Atomicity* recently proposed in [3] involves nearly no overhead. There, the authors split the EC-operations into *atomic blocks*, which are indistinguishable from each other by means of side-channel analysis. Hence, if an implementation does not leak out any data regarding which operation being performed, the side-channel information becomes uniform. In order to immunize our computations against SPA, we choose this countermeasure with some modifications. Our division of the EC-operations into atomic blocks will be different than the one given in [3]. This is to facilitate our pipelining scheme.

To immunize ECC from DPA, many countermeasures have been proposed. Most of them involve randomization of the processed data, such as the representation of the point or of the curve or of the scalar. We do not discuss the these issues at length here. The interested reader can refer to Ciet's Thesis [4] for a comprehensive treatment. We just observe that almost all schemes can be adapted to our method to make it DPA resistant.

3 Dividing EC-Operations into Atomic Blocks

We divide each EC-operation into atomic blocks. Following [3], each block contains one multiplication and two additions. Subtraction is treated as a negation followed by an addition. To accommodate subtractions we include one negation in each atomic block. The atomic blocks are presented in Table 1 and Table 2. Our ADD and DBL algorithms are designed, keeping in mind scalar multiplication algorithm. In the binary algorithm, described in Section 2.1 whenever an addition is carried out, one input is fixed i.e. P. So we may assume that like DBL, algorithm ADD has also one input P_i. Also, we can keep the point P in affine coordinates and gain efficiency by using mixed addition algorithm. Note that in Table 1 and Table 2, we have assumed that the EC-operations always get their inputs (X_i, Y_i, Z_i) at three specific locations T_6, T_7, T_8 respectively. Also, the EC-operation write back their outputs as these are computed to these locations only. The coordinates of the point $P = (X, Y, 1)$, which is an argument to all addition operations are also stored in two specific locations $T_x = X, T_y = Y$. Also, the curve parameter a needs to be stored. These six locations are public in the sense that any EC-operations in any of the two pipe stages can use them. One more location is required for the dummy operations. Both operations in the pipeline will share this location. Besides while two EC-operations being computed in the pipeline, each of them will have some locations (five each) private to them to store their intermediate variables. Thus the method requires 17 locations for the computation.

In Table 2 we provide the mixed addition algorithm in atomic blocks. Mixed addition requires 11 multiplications and doublings involves 10. So, adding one dummy multiplication to the DBL and some additions and negations to ADD/DBL, we can use whole of them as atomic blocks. However in that case we have to use these EC-operations as atomic units of computation. So, one operation has to be completed before the other begins. We do not adopt this approach as our aim in this work is to break the EC-operations into parts such that a part of one can start execution while a part of another is still in the pipeline.

3.1 An Analysis of ADD and DBL

Let us analyze the ADD and DBL algorithms presented in the Table 1 and Table 2. To DBL, there are three inputs, namely, X_i, Y_i, Z_i. It computes the double of the input point. Let us look at the various atomic blocks more closely.

We make the following observations on DBL:

– The atomic blocks $\Delta_1, \Delta_2, \Delta_3$ can be computed with the input Z_i only.
– Input X_i is needed by DBL at block Δ_4 and thereafter.
– The block Δ_5 needs the input Y_i as well. But Δ_5 produces the output Z_{i+1}. So, the next EC-operation can begin after DBL completes Δ_5.
– The atomic block Δ_8 produces the output X_{i+1}.
– Δ_{10} produces the output Y_{i+1} and the process terminates.

Table 2. ADD Algorithm in Atomic Blocks

ADD Algorithm
Input: $P = (T_x, T_y), P_i = (X_i, Y_i, Z_i)$
Output: $P + P_i = (X_{i+1}, Y_{i+1}, Z_{i+1})$.

Γ_1	$R_1 = T_8 \times T_8$ (Z_i^2) * * *	Γ_7	$R_2 = R_2 \times R_4$ $(-U_1W^2)$ $R_5 = R_2 + R_2$ $(-2U_1W^2)$ * *
Γ_2	$R_2 = T_x \times R_1$ (U_1) * $R_2 = -R_2$ $(-U_1)$ *	Γ_8	$R_1 = R_4 \times R_1$ (W^3) $R_1 = R_1 + R_5$ $(W^3 - 2U_1W^2)$ $R_3 = -R_3$ $(-S_1)$ $R_5 = R_3 + T_7$ $(S_2 - S_1 = -R)$
Γ_3	$R_3 = T_y \times T_8$ (YZ_i) * * *	Γ_9	$T_6 = R_5 \times R_5$ (R^2) $T_6 = T_6 + R_1$ $(\underline{X_{i+1}})$ $R_2 = T_6 + R_2$ $(X_{i+1} - U_1W^2)$
Γ_4	$R_3 = R_3 \times R_1$ (S_1) $R_1 = R_2 + T_6$ $(-W)$ $R_1 = -R_1$ (W) *	Γ_{10}	$R_2 = R_5 \times R_2$ $(-R(X_{i+1} - U_1W^2))$ * *
Γ_5	$T_8 = R_1 \times T_8$ $(\underline{Z_{i+1}})$ * * *	Γ_{11}	$T_7 = R_3 \times R_4$ $(-S_1W^2)$ $T_7 = T_7 + R_2$ $(\underline{Y_{i+1}})$ * *
Γ_6	$R_4 = R_1 \times R_1$ (W^2) * * *		

If instead a subtraction should be performed (add the negative $(-T_x, T_y)$), incorporate $R_2 = -T_x$ in Γ_1 and replace the first step of Γ_2 by $R_2 = R_2 \times R_1$.

We have similar observations on ADD:

– The atomic blocks $\Gamma_1, \Gamma_2, \Gamma_3$ can be computed with the input Z_i only.
– Input X_i is needed by ADD at block Γ_4 and thereafter.

– Γ_5 produces the output Z_{i+1}. So, the next EC-operation can begin after ADD completes Γ_5.
– The input Y_i is not required till the atomic block Γ_8.
– Γ_9 produces the output X_{i+1} and Γ_{11} produces Y_{i+1} and the process terminates.

The most interesting part of this division into atomic blocks is that both EC-operations perfectly match in a producer-consumer relation. In most situations as we will see in the next section, as soon as an output is produced by the producer process the consumer process consumes it. In some situations when the consumer process requires the input before it is produced by the producer, the consumer process has to wait an atomic block. However, such situations will not arise much frequently, hence it does not affect the efficiency much.

4 Pipelining the Scalar Multiplication Algorithm in ECC

In this section we describe our pipelining scheme – a two stage one, each stage executing an EC-operation in parallel. In the following discussion we assume that the EC-operation executing in pipe stage 1 gets its inputs when it needs. Later we will see, it is not always true. However such cases will not occur very frequently.

In the computation of the scalar multiplication, an DBL is always followed by an ADD or DBL, but an ADD is always followed by an DBL. So in the proposed pipeline we always see a pattern like DBL(producer)-DBL(consumer) or DBL(producer)-ADD(consumer) or ADD(producer)-DBL(consumer). This is true even if the scalar is represented in NAF or w-NAF and makes the sequence DBL-DBL more frequent.

Let us see how these EC-operations play their parts in this producer-consumer relation. We show this in the Table 3. The atomic blocks Γ_i's belong to an ADD and Δ_j's belong to DBL. Besides we have given a superscript to each of them to denote which EC-operation has entered the pipeline earlier. In the following description we will refer to pipe stage 1 and pipe stage 2 as PS1 and PS2 respectively. Also, in this discussion *our unit of time is time taken to execute one atomic block*. In the next three subsections we will discuss how EC-operations coupled with each other behave in the pipeline.

4.1 DBL-DBL Scenario

Let us first consider the DBL-DBL scenario. It is presented in Columns 2 and 3 of Table 3.

– Let us assume that the first DBL (say, $\text{DBL}^{(i)}$) and enters PS1 at time $k+1$.
– At time $k+5$, $\text{DBL}^{(i)}$ produces its first output (Z_{i+1}) and enters PS2. The second doubling $\text{DBL}^{(i+1)}$ enters the stage PS1.
– At time $k+8$, $\text{DBL}^{(i)}$ produces its second output. $\text{DBL}^{(i+1)}$ completes its 3rd atomic block $\Delta_3^{(i+1)}$ at the same time.

Table 3. EC-operations in the pipeline

	DBL-DBL		DBL-ADD		ADD-DBL	
Time	PS1	PS2	PS1	PS2	PS1	PS2
k	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k+1$	$\Delta_1^{(i)}$	-	$\Delta_1^{(i)}$	-	$\Gamma_1^{(i)}$	-
$k+2$	$\Delta_2^{(i)}$	-	$\Delta_2^{(i)}$	-	$\Gamma_2^{(i)}$	-
$k+3$	$\Delta_3^{(i)}$	-	$\Delta_3^{(i)}$	-	$\Gamma_3^{(i)}$	-
$k+4$	$\Delta_4^{(i)}$	-	$\Delta_4^{(i)}$	-	$\Gamma_4^{(i)}$	-
$k+5$	$\Delta_5^{(i)}$	-	$\Delta_5^{(i)}$	-	$\Gamma_5^{(i)}$	-
$k+6$	$\Delta_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Gamma_1^{(i+1)}$	$\Delta_6^{(i)}$	$\Delta_1^{(i+1)}$	$\Gamma_6^{(i)}$
$k+7$	$\Delta_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Gamma_2^{(i+1)}$	$\Delta_7^{(i)}$	$\Delta_2^{(i+1)}$	$\Gamma_7^{(i)}$
$k+8$	$\Delta_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Gamma_3^{(i+1)}$	$\Delta_8^{(i)}$	$\Delta_3^{(i+1)}$	$\Gamma_8^{(i)}$
$k+9$	$\Delta_4^{(i+1)}$	$\Delta_9^{(i)}$	$\Gamma_4^{(i+1)}$	$\Delta_9^{(i)}$	*	$\Gamma_9^{(i)}$
$k+10$	*	$\Delta_{10}^{(i)}$	$\Gamma_5^{(i+1)}$	$\Delta_{10}^{(i)}$	$\Delta_4^{(i+1)}$	$\Gamma_{10}^{(i)}$
$k+11$	$\Delta_5^{(i+1)}$	*	$\vdots$	$\Gamma_6^{(i+1)}$	*	$\Gamma_{11}^{(i)}$
$k+12$	$\vdots$	$\Delta_6^{(i+1)}$	$\vdots$	$\Gamma_7^{(i+1)}$	$\Delta_5^{(i+1)}$	*

- At time $k+9$ DBL$^{(i)}$ computes $\Delta_9^{(i)}$ and DBL$^{(i+1)}$ computes $\Delta_4^{(i+1)}$. Note that DBL$^{(i+1)}$ requires its second input i.e. X_i in this block, which is available to it. It was computed by DBL$^{(i)}$ in the previous atomic block.
- During time $k+10$, DBL$^{(i)}$ computes $\Delta_{10}^{(i+1)}$ and computes its third output (Y_{i+1}). DBL$^{(i+1)}$ should compute $\Delta_5^{(i)}$. But it needs its third input which is being computed at this time only. Hence it waits. DBL$^{(i)}$ terminates at the end of time $k+10$.
- DBL$^{(i+1)}$ computes $\Delta_5^{(i)}$, produces its first output and moves to PS2 in the next time unit. Although the other pipe stage is vacant now it can not be utilized as DBL$^{(i+1)}$ has not yet produced its first output.

Note that in this scenario, when two DBL's enter the pipeline one by one, two pipeline stages remain idle (one at time $k+10$ and another at time $k+11$) during the computations. We have marked them by '*' in the table. If the attacker using the side-channel information can detect this he may be able to conclude that two doubling were being computed now. To keep the adversary at bay we can compute two dummy blocks at these times. That will also implement the wait for the other process.

One can easily convince oneself that these choices are optimal. The computation of Z_{i+1} requires Y_i which is only provided in the final stage of the previous doubling. Hence, a wait stage cannot be avoided.

4.2 DBL-ADD Scenario

Let us consider the situation when an DBL is followed by an ADD. This scenario is described in Columns 4 and 5 of Table 3. Unlike the previous discussion we will refer to the operations as ADD and DBL only, without any superscript. Note that the DBL has entered the pipeline first and ADD later. Suppose the DBL starts at time $k+1$. We can see that:

- At time $k+5$, DBL computes block $\Delta_5^{(i)}$, its first output Z_{i+1} and then it enters PS2 at the next time unit.
- At time $k+6$, the ADD enters at PS1, uses the output Z_{i+1} of DBL.
- At time $k+8$, the DBL completes its block $\Delta_8^{(i)}$ and produces the output X_{i+1}.
- At time $k+9$, ADD computes $\Gamma_4^{(i+1)}$. It needs its second input (X_{i+1}), which is produced by the DBL in the previous time interval.
- At time $k+10$, the DBL computes its last atomic block and provides its third output. The ADD computes $\Gamma_5^{(i+1)}$. The last output computed by the DBL is required by the ADD two time units later.

In this scenario the coupling of operations is perfect. No pipeline stages are wasted. Note however, that this sequence is always followed by a doubling. That sequence is discussed in the next paragraph.

4.3 ADD-DBL Scenario

The scenario, when an ADD is followed by an DBL has been presented in Columns 6 and 7 of Table 3. For sake of brevity we are not going for an analysis of it. One can see that here the combination of the EC-operations involves three wait stages at times $k+9$, $k+11$ and $k+12$. Still this is the optimal way of performing this sequence and it fits perfectly after the DBL-ADD sequence discussed above. In DBL-ADD-DBL the addition finishes 12 steps after entering PS1 and at the same time the following doubling can enter PS2. These observations also guarantee that 6 atomic blocks are necessary for computation of each EC-operation (see Section 5), except for the first and the last ones. This requirement of 6 atomic blocks is exact and not just asymptotic.

5 Implementation and Results

In this section we will discuss the issues related to the implementation of the scheme. Also, we will demonstrate the speed-up that can be achieved in an implementation.
Hardware Requirement: As the proposed scheme processes two EC-operation simultaneously, we will require more hardware support than is generally required for ECC. To implement the pipe stages we will require a multiplier and an adder for each of the pipe stages. As addition is a much cheaper operation

than multiplication one adder can be shared between the pipe stages. We do not need separate (multiplicative) inverter for each pipe stage. In fact, we need only an inversion after completing all the EC-operations. So one inverter would suffice. Thus, in comparison to a sequential computation we need only one more multiplier to implement the proposed scheme.

Memory Requirement: In general ADD requires 7 locations and DBL requires 6 locations in memory in sequential execution, where one EC-operation is executed at a time. So in a sequential implementation the whole scalar multiplication can be computed using 7 locations for the EC-operations. The proposed scheme requires 17 memory locations i.e. 10 extra memory locations. From the space this corresponds to 5 precomputed points, however our scheme needs active registers and not only storage. For a fair comparison we will later compare our algorithm to a sequential one with 8 precomputed points.

Synchronization: As we have said there are seven locations where some values will be stored during whole process of computation. So, if two processes working at two stages of the pipeline wish to access these values simultaneously, conflict may arise. Particularly, if one process is trying to read and the other is trying to write the same location at the same time, then it will lead to a very serious problem. The input values X, Y and a are static and no attempt is made to write on these locations. Checking the above tables one can observe that the atomic blocks are arranged in a manner that no conflicts occur.

Resistance Against SCA: As the technique uses side-channel atomicity, it is secure against simple power analysis under the assumption (cf. [3,4]) that dummy operations cannot be detected and that squarings and multiplications are indistinguishable from the side channel. Note that the Hamming weight of the scalar is leaked; we come back on this later. To resist DPA Joye-Tymen's curve randomization [15] can be easily adopted into the scheme. It will require two more storage locations. As the scheme uses affine representation of the point, it is does not adapt directly to Coron's point randomization [7]. However, note that after the first doubling, the output point is no more affine. Hence it can be randomized. Also this later randomization does not compromise the security because, the first EC-operation is always a doubling.

A second option is to do the preprocessing step $T_6 = T_x \times Z^2, T_7 = T_y \times Z^3, T_8 = Z$, for some randomly chosen Z. This requires 4 multiplications and the input to the first doubling is no longer affine; hence, the costs are higher than in the first proposal. Both ways there is absolutely no problem in the scheme to adapt to scalar randomizations.

Performance: We discuss the performance of the scheme in depth. There are two multipliers, one for each of the pipe stages. The multiplications in the atomic blocks being executed in the pipe stages are computed in parallel. As said earlier, the scheme can be made resistant against DPA, using various randomization techniques. That will require some routine computations. In the discussion below we neglect these routine computations. Also, we will neglect the routine computation required at the end to convert the result from the Jacobian to affine coordinates.

To compute mP, if m has hamming weight h and length n one has to compute $n-1$ DBL and h ADD. An DBL operation requires 10 atomic blocks and an ADD requires 11 atomic blocks to complete. In a sequential execution that will consume $10(n-1)+11h$ units of time.

In the binary algorithm with the scalar multiplier m expressed in binary, $h = n/2$ on average. So, the computation of the scalar multiplication requires $10(n-1)+11(n/2) = 15.5n - 10$ atomic blocks. That is, one has to compute about 15.5 atomic blocks per bit on the average. If m is represented in 160 bits, i.e. $n = 160$, the scalar multiplication can be carried out by executing 2470 atomic blocks or in 2470 units of time.

In NAF representation of the multiplier, $h = n/3$ on average. So the computation time is $10(n-1)+11 \times n/3$ time units. That is one has to compute about $13.6n$ atomic blocks or 13.6 atomic blocks per bit of the multiplier. If m is expressed in NAF and $n = 160$, the computation requires to execute 2177 atomic blocks. That is the computation takes 2177 units of time.

The binary methods with or without NAF representation use less memory than our methods. For sake of fairness let us compare the performance of our method with with the method using w-NAF (see [22]). The method requires storing of 2^{w-1} points and $n-1$ doublings and $1/(w+1)$ additions on the average. For a scalar of 160 bits with $w = 5$, the method in a sequential execution requires to store 16 points and computes the scalar multiplication in 1893 units of time.

Example:

In Table 4, we have exhibited the computation process for a small scalar multiplier $38 = 100110$. To compute $38P$, one has to carry out EC-operations as DBL, DBL, DBL, ADD, DBL, ADD, DBL. Note that this multiplier encompasses all possibilities, i.e. DBL-DBL, DBL-ADD and ADD-DBL. The computation takes 46 units of time. In the table we have shown how the computation progresses. Each atomic block has been assigned a superscript to denote the serial number of the EC-operation to which it belongs. Also some atomic blocks are prefixed or suffixed by (X) or (Y) or (Z). A suffix indicates that at that atomic block the EC-operation outputs the corresponding value. A prefix indicates that at the specified atomic block the EC-operation consumes that input. Also, a '#' sign in the time column indicates that an EC-operation exits the pipeline at that time. A '*' in a pipe stage indicates a dummy atomic block has to be computed there. A '-' indicates no computation.

As we can check from the table, in the pipelining scheme, the first EC-operation which is usually an DBL, completes in 10 time units. In fact, as we take the base point in affine coordinates, first three blocks are not necesssary and it needs only 7 blocks. If we use Coron's randomization here 5 more blocks are required for that. After that an EC-operation (be it an DBL or an ADD) completes in every 6 units of time. Let m be represented by n bits with hamming weight h. Then the scalar multiplication will involve $h+n-1$ EC-operations ($n-1$ doublings and h additions). The first doubling will take 7 units of time and the other $n+h-2$ will be computed in 6 units of time in the pipelining scheme. So it will take $7+6(n+h-2)+5 = 6(n+h)$ units of time. For a scalar

Table 4. An Example of the Pipelining

Time	PS1	PS2	Time	PS1	PS2
1	$(Z)\Delta_1^{(1)}$	-	24	$\Delta_2^{(5)}$	$\Gamma_7^{(4)}$
2	$\Delta_2^{(1)}$	-	25	$\Delta_3^{(5)}$	$(Y)\Gamma_8^{(4)}$
3	$\Delta_3^{(1)}$	-	26	*	$\Gamma_9^{(4)}(X)$
4	$(X)\Delta_4^{(1)}$	-	27	$(X)\Delta_4^{(5)}$	$\Gamma_{10}^{(4)}$
5	$(Y)\Delta_5^{(1)}(Z)$	-	28#	*	$\Gamma_{11}^{(4)}(Y)$
6	$(Z)\Delta_1^{(2)}$	$\Delta_6^{(1)}$	29	$(Y)\Delta_5^{(5)}(Z)$	*
7	$\Delta_2^{(2)}$	$\Delta_7^{(1)}$	30	$(Z)\Gamma_1^{(6)}$	$\Delta_6^{(5)}$
8	$\Delta_3^{(2)}$	$\Delta_8^{(1)}(X)$	31	$\Gamma_2^{(6)}$	$\Delta_7^{(5)}$
9	$(X)\Delta_4^{(2)}$	$\Delta_9^{(1)}$	32	$\Gamma_3^{(6)}$	$\Delta_8^{(5)}(X)$
10#	*	$\Delta_{10}^{(1)}(Y)$	33	$(X)\Gamma_4^{(6)}$	$\Delta_9^{(5)}$
11	$(Y)\Delta_5^{(2)}(Z)$	*	34#	$\Gamma_5^{(6)}(Z)$	$\Delta_{10}^{(5)}(Y)$
12	$(Z)\Delta_1^{(3)}$	$\Delta_6^{(2)}$	35	$(Z)\Delta_1^{(7)}$	$\Gamma_6^{(6)}$
13	$\Delta_2^{(3)}$	$\Delta_7^{(2)}$	36	$\Delta_2^{(7)}$	$\Gamma_7^{(6)}$
14	$\Delta_3^{(3)}$	$\Delta_8^{(2)}(X)$	37	$\Delta_3^{(7)}$	$(Y)\Gamma_8^{(6)}$
15	$(X)\Delta_4^{(3)}$	$\Delta_9^{(2)}$	38	*	$\Gamma_9^{(6)}(X)$
16#	*	$\Delta_{10}^{(2)}(Y)$	39	$(X)\Delta_4^{(7)}$	$\Gamma_{10}^{(6)}$
17	$(Y)\Delta_5^{(3)}(Z)$	*	40#	*	$\Gamma_{11}^{(6)}(Y)$
18	$(Z)\Gamma_1^{(4)}$	$\Delta_6^{(3)}$	41	$(Y)\Delta_5^{(7)}(Z)$	*
19	$\Gamma_2^{(4)}$	$\Delta_7^{(3)}$	42	-	$\Delta_6^{(7)}$
20	$\Gamma_3^{(4)}$	$\Delta_8^{(3)}(X)$	43	-	$\Delta_7^{(7)}$
21	$(X)\Gamma_4^{(4)}$	$\Delta_9^{(3)}$	44	-	$\Delta_8^{(7)}(X)$
22#	$\Gamma_5^{(4)}(Z)$	$\Delta_{10}^{(3)}(Y)$	45	-	$\Delta_9^{(7)}$
23	$(Z)\Delta_1^{(5)}$	$\Gamma_6^{(4)}$	46#	-	$\Delta_{10}^{(7)}(Y)$

Table 5. Comparison of Performance for $n = 160$

Algorithm	Binary	NAF	w-NAF $(w = 4)$
Sequential	2477	2177	1893
Pipelined	1438	1278	1152

multiplier of length n bits represented in binary form, $h = n/2$ on average. Thus the pipelining scheme will require $6(n + n/2) = 9n$ units of time on the average. For $n = 160$ the proposed scheme will take 1440 units of time to compute the scalar multiplication.

If the scalar multiplier is expressed in NAF, then $h = n/3$ on the average. Hence time requirement will be $8n$ time-units. This implies, for $n = 160$ the time required is 1280. In either case it is a speed-up of around 41 percent.

Note that in both cases described above our method is better than even sequential w-NAF method. If w-NAF is used in pipelining scheme with those extra storage, then for $w = 4$, the scalar multiplication can be computed in 1152 units of time. We have summarized this discussion in the Table 5.

Comparison with Parallel Implementations

Parallelised computation of scalar multiplication on ECC was described for the first time by Koyama and Tsuruoka in [19]. A special hardware was used to carry out the computation in their proposal. We compare our scheme with some of the recent proposals which are claimed to be SCA resistant. The scheme proposed in [8], uses a parallelized encapsulated-add-and-double algorithm using Montgomery arithmetic. This algorithm uses two multipliers and takes $10[m]$ computations per bit of the scalar. Our algorithm as shown previously with NAF representation of the scalar takes only $8[m]$ computation per bit. The storage requirements are similar. Furthermore, we can obtain additional speed-up by allowing precomputations. In [1], the authors have proposed efficient algorithms for computing the scalar multiplication with SIMD (*Single Instruction Multiple data*). Similar and more efficient algorithms are also proposed in [12]. In [12] the authors have given two proposals. The first proposal, like our scheme, does not use precomputations and takes $1629[m]$ to compute the scalar multiplication. They have taken $[s] = 0.8[m]$ and the cost includes all routine calculation including the cost of Joye-Tymen's countermeasure for DPA. In contrast, pipelining requires only $1319[m]$ (all inclusive). Their second proposal uses precomputed points, applies signed window expansions of the scalar and is quite efficient. However, in a later work with Möller, the same authors (see [11]) remark that using a precomputed table in affine coordinates is not secure against *fixed table attacks*, a differential power attack. Even in Jacobian coordinates while using a fixed precomputed table, the values in the table should always be randomized before use.

Acknowledgment. The author is thankful to Dr. T. Lange, Prof. D. Hankerson and Dr. P. Sarkar for reading the manuscript of the paper and giving some constructive comments. Particularly, he is greatly indebted to Dr. Lange, who made several suggestions for improvement of the paper.

References

1. K. Aoki, F. Hoshino, T. Kobayashi and H. Oguro. *Elliptic Curve Arithmetic Using SIMD*, In ISC, 2001, LNCS 2200, pp. 235-247, Springer-Verlag, 2001
2. E. Briér and M. Joye. *Weierstrass Elliptic Curves and Side-Channel Attacks.* In PKC 2002, LNCS 2274, pages 335-345, Springer-Verlag,2002.
3. B. Chevallier-Mames, M. Ciet and M. Joye. *Low-cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity*, IEEE Trans. on Computers, 53(6):760-768, 2004.
4. M. Ciet. *Aspects of Fast and Secure Arithmetics for Elliptic Curve Cryptography*, Ph. D. Thesis, Louvain-la-Neuve, Belgique.
5. C. Cohen. Analysis of the flexible window powering algorithm, To appear *J. Cryptology*, 2004.

6. H. Cohen, A. Miyaji, and T. Ono. *Efficient Elliptic Curve Exponentiation Using Mixed coordinates*, In ASIACRYPT'98, LNCS 1514, pp. 51-65, Springer-Verlag, 1998.
7. J. -S. Coron. *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems*, In *CHES 1999*, pages 292-302.
8. W. Fischer, C. Giraud, E. W. Knudsen, J. -P. Seifert. *Parallel Scalar Multiplication on General Elliptic Curves over* $\mathbf{F}_p$ *hedged against Non-Differential Side-Channel Attacks*, Available at IACR eprint Archive, Technical Report No 2002/007, http://www.iacr.org.
9. K. Fong and D. Hankerson and J. López and A. Menezes. *Field inversion and point halving revisited*, Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, 2003.
10. D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004.
11. T. Izu, B. Möller and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant Against Side Channel Attacks, Proceedings of Indocrypt 2002, LNCS 2551, pp 296-313, Springer-Verlag.
12. T. Izu and T. Takagi. Fast Elliptic Curve Multiplications with SIMD operation, ICICS 2002, LNCS, pp 217-230, Springer-Verlag.
13. T. Izu and T. Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks, ICICS 2002, LNCS, pp 217-230, Springer-Verlag.
14. T. Izu and T. Takagi. Improved Elliptic Curve Multiplication Methods Resistant against Side Channel Attacks, INDOCRYPT 2002, LNCS, pp , Springer-Verlag.
15. M. Joye and C. Tymen. *Protection against differential attacks for elliptic curve cryptography*, CHES 2001, LNCS 2162, pp 402-410, Springer-Verlag.
16. N. Koblitz. *Elliptic Curve Cryptosystems*, Mathematics of Computations, 48:203-209, 1987.
17. P. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems*, CRYPTO'96, LNCS 1109, pp. 104-113, Springer-Verlag, 1996.
18. P. Kocher, J. Jaffe and B, Jun. *Differential Power Analysis,* CRYPTO'99, LNCS 1666, pp. 388-397, Springer-Verlag, 1999.
19. K. Koyama, Y. Tsuruoka. *Speeding up elliptic Curve Cryptosystems Using a Signed Binary Windows Method*, In CRYPTO'92, LNCS 740, pp 345-357, Springer-Verlag, 1992.
20. A. J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
21. V. S. Miller. *Use of Elliptic Curves in Cryptography,* CRYPTO'85, LNCS 218, pp. 417-426, Springer-Verlag, 1985.
22. J. Solinas. *Efficient arithmetic on Koblitz curves*, in Designs, Codes and Cryptography, 19:195-249, 2000.