Signed Binary Representations Revisited

Katsuyuki Okeya[1], Katja Schmidt-Samoa[2],
Christian Spahn[2], and Tsuyoshi Takagi[2]

[1] Hitachi, Ltd., Systems Development Laboratory,
292, Yoshida-cho, Totsuka-ku, Yokohama, 244-0817, Japan
ka-okeya@sdl.hitachi.co.jp
[2] Technische Universität Darmstadt, Fachbereich Informatik,
Hochschulstr. 10, D-64283 Darmstadt, Germany
{samoa,takagi}@informatik.tu-darmstadt.de

Abstract. The most common method for computing exponentiation of random elements in Abelian groups are sliding window schemes, which enhance the efficiency of the binary method at the expense of some precomputation. In groups where inversion is easy (e.g. elliptic curves), signed representations of the exponent are meaningful because they decrease the amount of required precomputation. The asymptotic best signed method is wNAF, because it minimizes the precomputation effort whilst the non-zero density is nearly optimal. Unfortunately, wNAF can be computed only from the least significant bit, i.e. right-to-left. However, in connection with memory constraint devices left-to-right recoding schemes are by far more valuable.
In this paper we define the MOF (*Mutual Opposite Form*), a new canonical representation of signed binary strings, which can be computed in any order. Therefore we obtain the first left-to-right signed exponent-recoding scheme for general width w by applying the width w sliding window conversion on MOF left-to-right. Moreover, the analogue right-to-left conversion on MOF yields wNAF, which indicates that the new class is the natural left-to-right analogue to the useful wNAF. Indeed, the new class inherits the outstanding properties of wNAF, namely the required precomputation and the achieved non-zero density are exactly the same.

Keywords: *addition-subtraction chains, exponentiation, scalar multiplication, signed binary, elliptic curve cryptosystem, efficient computation, non-adjacent form (NAF), mutual opposite form (MOF), left-to-right*

1 Introduction

In modern cryptosystems one of the most important basic operations is exponentiation g^d, where g is an element of an Abelian group G and d is an integer. A non-zero positive integer d is uniquely represented by a binary string:

$$d = d_{n-1}|d_{n-2}|...|d_1|d_0,$$

where $a|b$ denotes the concatenation of bits a, b, and $d_i \in \{0, 1\}$ for $i = 0, 1, ..., n-1$.

The most common method for performing an exponentiation is the square-and-multiply algorithm, which computes g^d according to the bits d_i (therefore it is often called binary method). The efficiency of this procedure may be enhanced if precomputation is allowed. In this case, we consider more general representations of the exponent, where each non-zero bit d_i is not restricted to be 1, but is an element of a suitable digit set $\mathcal{T}$ of integers. We call $d = \sum_i d_i 2^i$ a $\mathcal{T}$-representation, if $d_i \in \mathcal{T} \cup \{0\}$ holds for each i. In general, $\mathcal{T}$-representations loose the property of uniqueness. The left-to-right square-and-multiply algorithm is easily adjusted to work with a $\mathcal{T}$-representation of the exponent, namely multiplication by the base g is replaced with multiplication by precomputed elements g^{d_i}, where $d_i \in \mathcal{T}$ is the appropriate digit of d. Therefore, the important features of a $\mathcal{T}$-representation are the number of non-zero digits and the cardinality of $\mathcal{T}$, because they determine the required time and memory consumption for computing g^d, respectively. The research problem here is to find optimized representation classes in the sense of trade-off between high non-zero density and low memory consumption.

1.1 New Motivation for Exponentiation Algorithms

As the ubiquitous computing devices are penetrating our daily life, the importance of memory constraint devices (e.g. smart cards) in cryptography is increasing. Smart cards are equipped with several Kbytes RAM only and most of them are reserved for OS and stack. Thus, cryptographic algorithms should be optimized in terms of memory. For this reason we are reluctant to consume memory except the necessary precomputation related to $\mathcal{T}$ for computing exponentiation. Note that in connection with memory constraint devices, the most popular cryptosystems are based on elliptic curves [Kob87,Mil86], because elliptic curve cryptosystems (ECC) provide high security with moderate key-lengths. As elliptic curve groups are written additively, exponentiation has to be understood as scalar multiplication in this context.

Exponent recoding, i.e. the rewriting of the binary exponent to a $\mathcal{T}$-representation, may be performed from the least significant bit (we say "right-to-left") and from the most significant bit ("left-to-right"), respectively. For the purpose of ECC on memory constraint devices we prefer left-to-right to right-to-left recoding methods. The reason is as follows: In the case of elliptic curve scalar multiplication, the left-to-right evaluation stage is the natural choice (see Section 5 for details). If the exponent recoding is done right-to-left, it is necessary to finish the recoding and to store the recoded string before starting the left-to-right evaluation stage. In other words, we require additional n-bit (i.e. exponential size $\mathcal{O}(n)$) RAM for the right-to-left exponent recoding, where n is the bit size of the scalar.

On the contrary, if a left-to-right recoding technique is available, the recoding and evaluation stage may be merged to obtain an efficient exponentiation on the fly, without storing the recoded exponent at all. Therefore it is an important task to construct a left-to-right recoding scheme, even if the size of $\mathcal{T}$ and the non-zero density are not improved.

1.2 Known Solutions

The most established techniques for generating $\mathcal{T}$ representations are window methods (see, e.g., the textbooks [Knu81,MOV96] and the survey paper [Gor98]). Loosely speaking, in the window method with width w successively w consecutive bits of the binary exponent are scanned and, if necessary, replaced by a table-entry according to $\mathcal{T}$. We distinguish fixed window methods like the 2^w-ary method, where the window segmentation of the binary string is predetermined and the more advanced sliding window methods, where zero runs are skipped. As an example, let us consider the sliding window method with width $w = 3$. In this case, $\mathcal{T}$ equals $\{1, 3, 5, 7\}$. During the recoding stage, the binary exponent is rewritten by performing the following replacements: $1|1 \mapsto 0|3$, $1|0|1 \mapsto 0|0|5$, and $1|1|1 \mapsto 0|0|7$. Note that the sliding window conversion can be performed left-to-right and right-to-left as well. The results may differ syntactically, but the asymptotic non-zero density of both representations is the same, namely $1/(w+1)$. In the unsigned case (i.e. $\mathcal{T}$ consists only of positive integers), sliding window techniques are the method of choice.

However, a nice property of elliptic curves is that inversion is computed virtually for free. In this case, it is meaningful to consider digit sets containing negative integers, too. This reduces precomputation effort, because g^{-i} may be computed from g^i on the fly, such that only the elements $g^{|i|}$ for $i \in \mathcal{T}$ have to be precomputed. However, the question arises how to construct a signed $\mathcal{T}$ representation. In general, there are two strategies. The first one is to construct a $\{-1, +1\}$ representation of d (also called a signed binary representation) and to apply window methods afterwards. Here, the most common signed binary representation is NAF (non-adjacent-form) [Rei60,IEEE], which can be obtained from the binary representation by applying the conversion $*|1|1 \mapsto * + 1|0|\overline{1}$ repeatedly, where $\overline{1}$ denotes -1 and $*$ stands for any binary digit. However, the carry-over $+1$ occurring in the first digit forces the recoding to be performed from the least significant bit, i.e. right-to-left. The second strategy is to generalize the NAF recoding for $w > 2$ in order to obtain wNAF [Sol00,BSS99] (here, the non-adjacent property states that among any w adjacent bits, at most one is non-zero). According to [BSS99], this strategy is the optimal one for $w > 3$. But unfortunately, this strategy suffers from the same drawback as the first one, namely as carry-overs are required, the recoding is restricted to be done right-to-left. Consequently, all exponentiation strategies based on signed $\mathcal{T}$-representations require $\mathcal{O}(n)$ bits of RAM additional memory to store the recoded exponent. Solely in the case of $w = 2$, Joye and Yen proposed a left-to-right binary recoding algorithm [JY00]. But it has been an unsolved problem to generate a left-to-right recoding algorithm for a general width $w > 2$. Note that the asymptotic non-zero density of wNAF is the same as for the unsigned sliding window method on binary, namely $1/(w + 1)$. Therefore, wNAF can be seen as its natural signed analogue, and we guess that there could be a carry-free generation method for wNAF. In this paper, the term carry-free refers to an algorithm that transforms the input string in situ, i.e. in each step only the knowledge of a fixed number of consecutive input bits is necessary.

1.3 Our Contributions

The aim of this paper is to solve both problems as follows: (1) we define a new canonical representation class of signed binary. We call it MOF (Mutual Opposite Form) and prove that each integer can be uniquely represented as a MOF. But the outstanding property of MOF is that it can be efficiently developed from a binary string right-to-left or left-to-right, likewise. Consequently, analogue to the unsigned case, sliding window methods may be applied to receive left-to-right and right-to-left recoding schemes for general width w. Surprisingly, applying the right-to-left width w sliding window method on MOF yields wNAF. However, the observation that in the unsigned case right-to-left sliding window yields an unsigned string with non-adjacent property stresses the analogy between unsigned Binary and signed MOF. Therefore we achieve a *carry-free* wNAF generation, a benefit of its own.

(2) Our major aim is to develop a left-to-right recoding algorithm, and this is achieved straightforwardly by applying the width w sliding window method left-to-right on MOF. We call the so-defined class wMOF and prove that each integer can be uniquely represented as a wMOF and that the asymptotic non-zero density of wMOF equals $1/(w+1)$, which is the same as for wNAF. Therefore the classes wNAF and wMOF may be seen as dual to each other. In general our proposed algorithm asymptotically requires additional $\mathcal{O}(w)$ bits of RAM, which is independent from the bit size n and dramatically reduces the required space comparing with previous methods. Consequently, due to its left-to-right nature, the new scheme is by far more convenient with respect to memory consumption than previous schemes. Interestingly, a straight-forward proof shows that for $w = 2$ the proposed method produces the same output as the Joye-Yen recoding, but 2MOF is more efficient in terms of counting the number of basic operations.

We finish this work with some explicit algorithms, proving that the proposed schemes are indeed useful for practical purposes. For example, we develop generating algorithms for wMOF based on efficient table-lookups, and we show how to exploit wMOF for implementing on-the-fly elliptic curve scalar multiplication.

2 Signed Representations

In this section we review some signed representations, which are important in connection with elliptic curve scalar multiplication. For the sake of simplicity, we only deal with non-negative integers d in the following. We call $d = \sum_i d_i 2^i$ a $\mathcal{T}$-representation, if $\mathcal{T}$ is a set of integers and $d_i \in \mathcal{T} \cup \{0\}$ holds for each i. If $\mathcal{T}$ contains negative integers, we speak of *signed* representations, and if $\mathcal{T}$ equals $\{\pm 1\}$, of *signed binary* representations. In general, signed binary representations are redundant. The most established one is NAF (non-adjacent form), introduced by Reitwiesner 1960 [Rei60]. A generalization of Reitwiesner's NAF recoding idea can be found in [Pro00,Avi61]. NAF can be easily defined by the property that at most one out of two consecutive digits is non-zero. Reitwiesner was able to show that ignoring leading zeros each integer has a unique NAF

representation. For this reason, some authors call NAF a canonical signed binary representation [EK94]. In addition, as shown among others by Jedwab and Mitchell [JM89], NAF representation provides the minimal Hamming weight. Consequently, the NAF representation of the exponent is the optimal choice if signed methods are meaningful and no precomputation is considered. It was first pointed out by Morain and Olivos that NAF can be used to speed up elliptic curve scalar multiplication [MO90].

However, the situation is less clear if extra memory is available and precomputation is admitted. In this case, signed representations using larger digit sets $\mathcal{T}$ should be taken into account. One strategy to construct a signed representation is to apply sliding window methods on signed binary representations. But as signed binary representation is redundant, the question arises which representation is the best for this purpose. Indeed, this is assumed to be an open problem by De Win et al. [WMPW98]. There are several methods to construct signed binary representations as a base for sliding window schemes [KT92,WMPW98], but none of these can be performed left-to-right. In this paper, we will develop a left-to-right recoding scheme, which is of high value in connection with memory constraint devices.

A different approach is wNAF. Instead of applying window techniques to signed binary representations, wNAF is computed directly from binary strings using a generalization of NAF recoding. First we review the definition of wNAF as stated in [Sol00].

Definition 1 (wNAF). *A sequence of signed digits is called wNAF iff the following three properties hold:*

1. *The most significant non-zero bit is positive.*
2. *Among any w consecutive digits, at most one is non-zero.*
3. *Each non-zero digit is odd and less than 2^{w-1} in absolute value.*

Note that 2NAF and NAF are the same. Algorithm 1 describes the generation of wNAF as proposed by Solinas [Sol00].

Algorithm 1 Generation of wNAF [Sol00]

Input: width w, an n-bit integer d
Output: wNAF $\delta_n|\delta_{n-1}|\ldots|\delta_0$ of d

```
i ← 0
while d ≥ 1 do
  if d is even then
    δ_i ← 0
  else
    δ_i ← d mods 2^w; d ← d − δ_i
  d ← d/2; i ← i + 1
return (δ_n, δ_{n−1}, ..., δ_0).
```

Here "mods" means the signed modulo, namely a mods b is defined as $a \bmod b$ and $-b/2 \leq a < b/2$. The algorithm generates wNAF from the least significant bit, that is right-to-left generation again. The average density of non-zero bits is asymptotically $1/(w+1)$ for $n \to \infty$, and the digit set equals $\mathcal{T} = \{\pm 1, \pm 3, \ldots, \pm(2^{w-1}-1)\}$ which seems to be minimal. Thus wNAF and its variants like modified window NAF [Möl02] are optimal in the sense of the trade-off between speed and memory for $w > 3$ [BSS99,BHLM01]. There are several other algorithms for generating wNAF, for example see [BSS99,MOC97] but each method needs carry-overs. Note that in the worst case all remaining bits are affected by the carry, therefore the previously known wNAF algorithms can not be considered as local methods. By inspecting Algorithm 1 closely, we observe that this generation can be seen as the natural signed analogue to the right-to-left sliding window method on (unsigned) Binary (here, mod instead of mods is computed). Indeed, the latter method produces a representation that fulfills the nonadjacent requirement (see Definition 1, property 3). Consequently, we conjecture that there might be a signed binary representation that produces wNAF when handled with sliding window conversions. The signed binary representation introduced in the next section will also serve for this purpose.

3 MOF: New Canonical Representation for Signed Binary Strings

In this section we present a new signed representation of integers. The proofs of the propositions in this section are Appendix A. In order to achieve a unique representation, we introduce the following special class of signed binary strings, called the mutual opposite form (MOF).

Definition 2 (MOF). *The n-bit mutual opposite form (MOF) is an n-bit signed binary string that satisfies the following properties:*

1. *The signs of adjacent non-zero bits (without considering zero bits) are opposite.*
2. *The most non-zero bit and the least non-zero bit are 1 and $\bar{1}$, respectively, unless all bits are zero.*

Some zero bits are inserted between non-zero bits that have a mutual opposite sign. An example of MOF is $0100\bar{1}01000\bar{1}001\bar{1}0$. An important observation is that each positive integer can be uniquely represented by MOF. Indeed, we have the following theorem.

Theorem 1. *Let n be a positive integer. $(n+1)$-bit MOF has 2^n pair-wise different representations. There is the bijective map between elements of $(n+1)$-bit MOF and n-bit binary strings.*

From this theorem, any n-bit binary string can be uniquely represented by $(n+1)$-bit MOF. We obviously have the following corollary about the non-zero density of MOF.

Corollary 1. *The average non-zero density of n-bit MOF is $1/2$ for $n \to \infty$.*

3.1 Converting Binary String to MOF

We show a simple and flexible conversion from n-bit binary string to $(n+1)$-bit MOF.

The crucial point is the following observation. The n-bit binary string d can be converted to a signed binary string by computing $\mu = 2d \ominus d$, where '$\ominus$' stands for a bitwise subtraction. Indeed, we convert d as follows:

$$\begin{array}{rl} & 2d = d_{n-1} \mid \quad d_{n-2} \quad \mid \ldots \mid \quad d_{i-1} \quad \mid \ldots \mid \quad d_1 \quad \mid \quad d_0 \quad \mid \\ \ominus & d = \quad \mid \quad d_{n-1} \quad \mid \ldots \mid \quad d_i \quad \mid \ldots \mid \quad d_2 \quad \mid \quad d_1 \quad \mid \quad d_0 \\ \hline & \mu = d_{n-1} \mid d_{n-2} - d_{n-1} \mid \ldots \mid d_{i-1} - d_i \mid \ldots \mid d_1 - d_2 \mid d_0 - d_1 \mid -d_0. \end{array}$$

Here the i-th signed bit of μ is denoted by μ_i, namely $\mu_i = d_{i-1} - d_i$ for $i = 1, ..., n-1$ and $\mu_n = d_{n-1}, \mu_0 = -d_0$. We can prove that the signed representation μ is MOF.

Proposition 1. *The operation $\mu = 2d \ominus d$ converts binary string d to its MOF μ.*

Algorithm 2 provides an explicit conversion from Binary to MOF.

Algorithm 2 Left-to-Right Generation from Binary to MOF

Input: a non-zero n-bit binary string $d = d_{n-1}|d_{n-2}|\ldots|d_1|d_0$
Output: MOF $\mu_n|\ldots|\mu_1|\mu_0$ of d

$\mu_n \leftarrow\!\shortmid\, d_{n-1}$
for $i = n-1$ down to 1 **do**
  $\mu_i \leftarrow\!\shortmid\, d_{i-1} - d_i$
$\mu_0 \leftarrow\!\shortmid\, -d_0$,
return $(\mu_n, \mu_{n-1}, ..., \mu_1, \mu_0)$.

In order to generate the i-th bit μ_i, Algorithm 2 stores just two consecutive bits d_{i-1} and d_i. This algorithm converts a binary string to MOF from the most significant bit in an efficient way. Note that it is also possible to convert a binary string to MOF right-to-left. Thus MOF representation is highly flexible.

Remark 1. Interestingly, the MOF representation of an integer d equals the recoding performed by the classical Booth algorithm for binary multiplication [Boo51]. The classical Booth algorithm successively scans two consecutive bits of the multiplier A (right-to-left). Depending on these bits, one of the following operations is performed:

No operation,	if $(a_i, a_{i-1}) \in \{(0,0),(1,1)\}$,
Subtract multiplicand B from the partial product,	if $(a_i, a_{i-1}) = (1,0)$,
Add multiplicand B to the partial product,	if $(a_i, a_{i-1}) = (0,1)$,

where a_{-1} is defined as 0. Of course, the design goal of this algorithm was to speed up multiplication when there are consecutive ones in the multiplier A, and to provide a multiplication method that works for signed and unsigned numbers as well. To our knowledge, this representation never served as a fundament of theoretical treatment of signed binary strings.

4 Window Methods on MOF

In this section we show how to decrease the non-zero density of MOF by applying window methods on it. First we consider the right-to-left width w sliding window method which surprisingly yields the familiar wNAF. In contrast to previously known generation methods, the new one is carry-free, i.e. in each step the knowledge of at most $w+1$ consecutive input bits is sufficient.

Then we define the dual new class wMOF as the result of the analogue left-to-right width w sliding window method on MOF. This conversion leads to the first left-to-right signed recoding scheme for general width w.

4.1 Right-to-Left Case: wNAF

In order to describe the proposed scheme, we need the conversion table for width w. First, we define the conversions for MOF windows of length l, such that the first and the last bit is non-zero:

$$\underbrace{0|...|0|2^{l-2}+1}_{l} \leftarrow \begin{cases} 1|\bar{1}|0|\ldots|0|0|1 \\ \underbrace{1|\bar{1}|0|\ldots|0|1|\bar{1}}_{l} \end{cases} \qquad \underbrace{0|...|0|2^{l-2}+3}_{l} \leftarrow \begin{cases} 1|\bar{1}|0|\ldots|0|1|0|\bar{1} \\ \underbrace{1|\bar{1}|0|\ldots|0|1|\bar{1}|1}_{l} \end{cases} \ldots$$

$$\ldots \underbrace{0|...|0|2^{l-1}-3}_{l} \leftarrow \begin{cases} 1|0|\ldots|0|\bar{1}|1|\bar{1} \\ \underbrace{1|0|\ldots|0|\bar{1}|0|1}_{l} \end{cases} \qquad \underbrace{0|...|0|2^{l-1}-1}_{l} \leftarrow \begin{cases} 1|0|\ldots|0|0|\bar{1} \\ \underbrace{1|0|\ldots|0|\bar{1}|1}_{l} \end{cases}$$

In addition, we have analogue conversions with all signs changed. To generate the complete table for width w, we have to consider all conversions of length $l = 2, 3, \ldots, w$. If $l < w$ holds, the window is filled with leading zeros.

Example: In the case of $w = 3$, we use the following table for the right-to-left sliding window method:

$$\text{Table}_{3\overleftarrow{SW}} : 001 \leftarrow \begin{cases} 001 \\ 01\bar{1} \end{cases} \qquad 00\bar{1} \leftarrow \begin{cases} 00\bar{1} \\ 0\bar{1}1 \end{cases} \qquad 003 \leftarrow \begin{cases} 10\bar{1} \\ 1\bar{1}1 \end{cases} \qquad 00\bar{3} \leftarrow \begin{cases} \bar{1}01 \\ \bar{1}1\bar{1} \end{cases}$$

In an analogue way $\text{Table}_{w\overleftarrow{SW}}$ is defined for general w. Based on this table, Algorithm 3 provides a simple carry-free wNAF generation.

Algorithm 3 Right-to-left Generation from Binary to wNAF

Input: width w, a non-zero n-bit binary string $d = d_{n-1}|d_{n-2}|\ldots|d_1|d_0$
Output: wNAF $\nu_n|\ldots|\nu_1|\nu_0$ of d

```
d_{n+w-2} ← 0; d_{n+w-3} ← 0; ...; d_n ← 0; d_{-1} ← 0; i ← 0
while i ≤ n do
  if d_{i-1} = d_i then
    ν_i ← 0; i ← i + 1
  else {The MOF window begins with a non-zero digit righthand}
    (ν_{i+w-1}, ..., ν_i) ← Table_{wSW←}(d_{i+w-2} − d_{i+w-1}, d_{i+w-3} − d_{i+w-2}, ..., d_{i-1} − d_i)
    i ← i + w
return (ν_n, ..., ν_1, ν_0)
```

Obviously, the output of Algorithm 3 meets the notations of Definition 1, therefore it is wNAF. If we knew that Definition 1 provides a *unique* representation, we could deduce that Algorithm 3 outputs the same as Algorithm 1. This is true, although we could not find a proof in literature. For the sake of completeness, we prove the following theorem in Appendix A via exploiting the uniqueness of MOF representation.

Theorem 2. *Every non-negative integer d has a representation as wNAF, which is unique except for the number of leading zeros.*

4.2 Left-to-Right Case: wMOF

In this section we introduce our new proposed scheme. The crucial observation is that as the generation Binary $\mapsto$ MOF can be performed left-to-right, the combination of this generation and left-to-right sliding window method leads to a complete signed left-to-right recoding scheme dual to wNAF.

In order to describe the proposed scheme, we need the conversion table for width w. The conversions for MOF windows of length l, such that the first and the last bit is non-zero, are defined in exactly the same way as in the right-to-left case (see the table in section (4.1) and reflect the assignments). To generate the complete table for width w, we have to consider all conversions of length $l = 2, 3, \ldots, w$ as before. The only difference is that if $l < w$ holds, the window is filled with *closing* zeros instead of leading ones. As an example, we construct the conversion table $\text{Table}_{4\overrightarrow{SW}}$ for width 4:

$$1000\} \mapsto 1000 \quad 1\bar{1}00\} \mapsto 0100 \quad \left.\begin{matrix}1\bar{1}10\\10\bar{1}0\end{matrix}\right\} \mapsto 0030 \quad \left.\begin{matrix}1\bar{1}01\\1\bar{1}1\bar{1}\end{matrix}\right\} \mapsto 0005 \quad \left.\begin{matrix}100\bar{1}\\10\bar{1}1\end{matrix}\right\} \mapsto 0007$$

$$\bar{1}000\} \mapsto \bar{1}000 \quad \bar{1}100\} \mapsto 0\bar{1}00 \quad \left.\begin{matrix}\bar{1}1\bar{1}0\\\bar{1}010\end{matrix}\right\} \mapsto 00\bar{3}0 \quad \left.\begin{matrix}\bar{1}10\bar{1}\\\bar{1}1\bar{1}1\end{matrix}\right\} \mapsto 000\bar{5} \quad \left.\begin{matrix}\bar{1}001\\\bar{1}01\bar{1}\end{matrix}\right\} \mapsto 000\bar{7}$$

The table is complete due to the properties of MOF. Note that because of the equalities $*1\bar{1} = *01$, $*\bar{1}1 = *0\bar{1}$ usually two different MOF-strings are converted to the same pattern. In an analogue way, $\text{Table}_{w\overrightarrow{SW}}$ is defined for general width w. In this case the digit set equals $\mathcal{T} = \{\pm 1, \pm 3, \ldots, \pm 2^{w-1} - 1\}$, which is the same as for wNAF. Therefore, the scheme requires only 2^{w-2} precomputed elements. Algorithm 4 makes use of this table to generate wMOF left-to-right.

In order to deepen the duality between wNAF and wMOF, we give a formal definition of wMOF and prove that it leads to a unique representation of non-negative integers.

Definition 3. *A sequence of signed digits is called wMOF iff the following three properties hold:*

1. *The most significant non-zero bit is positive.*
2. *All but the least significant non-zero digit x are adjoint by w-1 zeros as follows:*
   - *in case of $2^{k-1} < |x| < 2^k$ for an integer $2 \leq k \leq w-1$ the pattern equals $\underbrace{0\ldots0}_{k} x \underbrace{0\ldots0}_{w-k-1}$,*

– *in case of* $|x| = 1$ *either the pattern equals* $x\underbrace{0\ldots0}_{w-1}$ *and the next lower non-zero digit has opposite sign from* x *or the pattern equals* $0x\underbrace{0\ldots0}_{w-2}$ *and the next lower non-zero digit has the same sign as* x.

If x *is the least significant non-zero digit, it is possible that the number of right-hand adjacent zeros is smaller than stated above. In addition it is not possible that the last non-zero digit is a 1 following any non-zero digit.*

3. *Each non-zero digit is odd and less than* 2^{w-1} *in absolute value.*

This definition is directly related to the generation of wMOF. Note that the exceptional case corresponding to the least significant bit takes in account that the last window may be shorter than w.

Algorithm 4 Left-to-Right Generation from Binary to wMOF

Input: width w, a non-zero n-bit binary string $d = d_{n-1}|d_{n-2}|...|d_1|d_0$
Output: wMOF $\delta = \delta_n|\delta_{n-1}|...|\delta_1|\delta_0$ of d

```
d_{-1} ↤ 0; d_n ↤ 0; i ↤ n
while i ≥ w − 1 do
  if d_i = d_{i−1} then
    δ_i ↤ 0; i ↤ i − 1
  else {The MOF window begins with a non-zero digit lefthand}
    (δ_i, δ_{i−1} ..., δ_{i−w+1}) ↤ Table_{wSW→}(d_{i−1} − d_i, d_{i−2} − d_{i−1}, ..., d_{i−w} − d_{i−w+1})
    i ↤ i − w
if i ≥ 0 then
  (δ_i, δ_{i−1} ..., δ_0) ↤ Table_{i+1SW→}(d_{i−1} − d_i, d_{i−2} − d_{i−1}, ..., d_0 − d_1, −d_0)
return (δ_n, δ_{n−1}, ..., δ_1, δ_0).
```

Regarding the uniqueness and the non-zero density of wMOF, we have the following two theorems, proven in Appendix A.

Theorem 3. *Every non-negative integer* d *has a representation as* w*MOF, which is unique except for the number of leading zeros.*

Theorem 4. *The average non-zero density of* w*MOF is asymptotically* $1/(w+1)$ *for* $n \mapsto \infty$.

We finish this section with a detailed example of the conversion from Binary to MOF and the effects of several sliding window methods.

Bin	1	1	1	0	1	0	0	1	1	0	0	1	0	0	0	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	1	1
MOF	1	0	0	$\overline{1}$	1	$\overline{1}$	0	1	0	$\overline{1}$	0	1	$\overline{1}$	0	0	1	$\overline{1}$	1	$\overline{1}$	1	0	0	$\overline{1}$	1	$\overline{1}$	1	$\overline{1}$	1	$\overline{1}$	1	0	0	$\overline{1}$
2MOF	1	0	0	0	$\overline{1}$	$\overline{1}$	0	1	0	$\overline{1}$	0	0	1	0	0	0	1	0	1	1	0	0	0	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	0	0	$\overline{1}$
3MOF	1	0	0	0	0	$\overline{3}$	0	0	0	3	0	0	1	0	0	0	0	3	0	$\overline{1}$	0	0	0	0	$\overline{3}$	0	0	3	0	$\overline{1}$	0	0	$\overline{1}$
4MOF	0	0	0	7	0	0	0	5	0	0	0	0	$\overline{7}$	0	0	0	0	0	5	0	0	0	7	0	0	0	5	0	0	3	0	0	$\overline{1}$
NAF	1	0	0	$\overline{1}$	0	1	0	1	0	$\overline{1}$	0	0	1	0	0	1	0	$\overline{1}$	0	$\overline{1}$	0	0	0	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	0	0	$\overline{1}$
3NAF	1	0	0	0	0	$\overline{3}$	0	0	0	3	0	0	1	0	0	0	1	0	0	3	0	0	0	$\overline{1}$	0	0	$\overline{3}$	0	0	3	0	0	$\overline{1}$
4NAF	0	0	0	7	0	0	0	5	0	0	0	$\overline{3}$	0	0	0	$\overline{7}$	0	0	0	$\overline{5}$	0	0	0	0	$\overline{3}$	0	0	0	5	0	0	0	7

4.3 Left-to-Right Generation of (w)NAF

Although in the preceding section we have presented left-to-right generated signed representations that are at least as useful as (w)NAFs, from a theoretical point of view it is still an interesting question how to generate the (w)NAF from the most significant bit. The reason for the difficulty is a carry caused by the statement $d \leftarrow\!\!\shortmid d - \delta_i$ of Algorithm 1. To illustrate the problem, note that the binary strings 101010 and 101011 that only differ in the last digit are converted to the NAFs 101010 and $10\bar{1}0\bar{1}0\bar{1}$, respectively, which differ completely. Intuitively, it is not possible to generate NAF left-to-right without scanning any higher bits. In this section we exploit the MOF representation to discuss how many bits have to be scanned and how many additional storage is required.

Note that we obtain NAF if we apply the conversions $1\bar{1} \mapsto 01$ and $\bar{1}1 \mapsto 0\bar{1}$ right-to-left on MOF. However, performing the same conversions left-to-right may yield a different result. The critical sequence is of the shape

$$0\underbrace{1\bar{1}\ldots1}_{\text{odd}}0, \quad \text{or} \quad 0\underbrace{\bar{1}1\ldots\bar{1}}_{\text{odd}}0.$$

Note that this sequence corresponds to the binary string $1010\ldots011$. If the length of the sequence of alternating bits is even, then both of left-to-right and right-to-left conversions uniquely generate the same string, namely $\bar{b}b...\bar{b}b \mapsto 0\bar{b}...0\bar{b}$ for $b \in \{\pm 1\}$. But if the length is odd, left-to-right we obtain $\bar{b}b\ldots b\bar{b} \mapsto 0\bar{b}\ldots 0\bar{b}\bar{b}$, whereas right-to-left generates $\bar{b}b\ldots b\bar{b} \mapsto \bar{b}0b0b\ldots 0b$. Consequently, if this sequence appears, we have to scan it completely in order to compute the corresponding NAF. However, the first bit and the length of the critical sequence can uniquely determine the corresponding NAF, hence it is not necessary to store the sequence. Thus, the additional required storage in RAM is at most a few bits, namely the bit length of the critical sequence. Therefore, we obtain Algorithm 5.

Algorithm 5 Left-to-Right Generation Binary to NAF

Input: a non-zero n bit binary string $d = d_{n-1}|d_{n-2}|\ldots|d_1|d_0$
Output: NAF $\nu_n|\nu_{n-1}|\ldots|\nu_1|\nu_0$ of d

```
i ←\!\!\shortmid n; d_n ←\!\!\shortmid 0; d_{-1} ←\!\!\shortmid 0; d_{-2} ←\!\!\shortmid 0
while i > −1 do
  b ←\!\!\shortmid d_{i−1} − d_i
  if b = 0 then
    ν_i ←\!\!\shortmid 0; i ←\!\!\shortmid i − 1
  else {b ≠ 0}
    find the largest j s.t. d_{i−j−1} = d_{i−j}
    if j is odd then
      ν_i ←\!\!\shortmid b; ν_{i−1} ←\!\!\shortmid 0; ν_{i−2} ←\!\!\shortmid −b; ...; ν_{i−j+2} ←\!\!\shortmid 0; ν_{i−j+1} ←\!\!\shortmid −b; ν_{i−j} ←\!\!\shortmid 0
    else {j is even}
      ν_i ←\!\!\shortmid 0; ν_{i−1} ←\!\!\shortmid b; ...; ν_{i−j+2} ←\!\!\shortmid 0; ν_{i−j+1} ←\!\!\shortmid b; ν_{i−j} ←\!\!\shortmid 0
  i ←\!\!\shortmid i − j − 1
return (ν_n, ν_{n−1}, ...., ν_1, ν_0)
```

It is also possible to construct a left-to-right generation algorithm of wNAF, $w > 2$. In this case, the critical sequence is of the following shape

$$\underbrace{0\ldots0}_{w-1} a_i a_{i-1} \ldots a_1 a_0 \underbrace{0\ldots0}_{w-1}, \tag{1}$$

where the most and least $(w-1)$ bits are zero and no zero run of length $w-1$ appears in $a_i a_{i-1} \ldots a_1 a_0$. If it is possible to convert the critical sequence (1) left-to-right to wNAF, then we can generate wNAF from any MOF. In order to find the corresponding wNAF of (1), we scan the whole sequence right-to-left and obtain the segmentations that are produced by the right-to-left sliding window conversion MOF $\mapsto$ wNAF. Note that there is no need to store the width w windows, but we must detect and store the length of the zero runs between any two windows. In addition, the content of the left-most window, which may be smaller than w, has to be transfered. Afterwards, the sequence (1) can be rewritten as follows:

$$\underbrace{0\ldots0}_{w-1} |r|b_i|t_i| \ldots |b_2|t_2|b_1|t_1| \underbrace{0\ldots0}_{w-1}, \tag{2}$$

where r consists of at most $w-1$ consecutive bits of MOF (and may be the empty word ε), $b_j \in \{\epsilon, 0, 00, \ldots, \underbrace{0\ldots0}_{w-2}\}$, and each t_j is a length w pattern of MOF, corresponding to an entry of Table$_{w\overleftarrow{SW}}$. Here we have to store r and the b_j. Based on these informations, the corresponding wNAF is completely determined left-to-right. Thus we need to store at most $(w - 1 + \log_2(w-2))\frac{n}{w})$ bits.

4.4 Comparison with Previous Methods

In this section we clarify the difference to previous schemes for generating signed representations.

In 1992, Koyama and Tsuruoka developed a new recoding technique to convert a binary string to a signed binary string [KT92]. Following this step, a left-to-right sliding window method is applied. The new signed binary representation has the benefit that it reduces the asymptotic non-zero density, but it requires the sub-optimal digit set $\mathcal{T} = \{\pm1, \pm3, \ldots, \pm(2^w - 3)\}$. If the sliding window method is directly applied to NAF, due to the NAF property fewer possible window contents have to be taken into account, resulting in a smaller digit set $\mathcal{T}$. An easy calculation shows that the largest odd NAF consisting of at most w digits equals $\frac{1}{3}(2^{w+1} - 1)$ for odd w (cf. $1010\ldots01$) and $\frac{1}{3}(2^{w+1} + 1) - 2$ for even w (cf. $1010\ldots1001$). For this reason, De Win et al. prefer the latter method for elliptic curve scalar multiplication [WMPW98]. Although there are slightly more point operations needed to evaluate the scalar multiplication if the exponent is represented as wNAF compared to the [WMPW98] representation, the required precomputation is less in the wNAF case because of the smaller digit set. Indeed, Blake et al. proved that wNAF is asymptotically better than sliding window on NAF schemes if $w > 3$ [BSS99]. In the context of memory constraint

devices, a small digit set $\mathcal{T}$ is even more valuable, because fewer precomputed elements have to be stored. But as none of the preceding methods is a left-to-right scheme, each one requires *additional* memory $\mathcal{O}(n)$ to store the recoded string before starting the left-to-right evaluation of the scalar product. Note that in the context of sliding window on signed binary schemes like [KT92,WMPW98] the sliding window conversion may be performed left-to-right, but to obtain the signed binary representation we have to proceed right-to-left in either case.

In contrast, wMOF turns out as a complete left-to-right scheme. Consequently, there is no additional memory required for performing the scalar multiplication. In addition, due to the properties of MOF, the digit set of wMOF is the same as for wNAF and therefore minimal.

In order to compare the proposed algorithms with previous ones, we summarize the memory requirements of the new left-to-right schemes in the following theorem.

Theorem 5. *Algorithm 4 requires only $\mathcal{O}(w)$ bits memory for generating wMOF.*
Algorithm 5 requires at most $(\log_2 n)$ bits memory for generating NAF left-to-right. For general width w, there is a left-to-right algorithm that generates wNAF with at most $(w-1+\log_2(w-2))\frac{n}{w})$ bit memory.

Next, we compare the characterizing properties for the proposed schemes and some previous ones. In the second column, the value $\#\mathcal{T}/2$ equals the number of elements, that have to be precomputed and stored. In the last column, we describe the amount of memory (in bits) that is required additionally to this storage, e.g. to construct the signed representation or to store the converted string in right-to-left schemes. As usual, n equals the bit-length of the scalar, and SW is an abbreviation for sliding window.

Scheme	$\#\mathcal{T}/2$	1/N.-z. Density	Additional Memory
wNAF [Sol00,BSS99,MOC97]	2^{w-2}	$w+1$	$\mathcal{O}(n)$
[KT92]	$2^{w-1}-1$	$w+\frac{3}{2}$	$\mathcal{O}(n)$
NAF+SW as [WMPW98]	$\frac{1}{3}(2^w+(-1)^{w+1})$	$w+\frac{4}{3}-\frac{(-1)^w}{3\cdot 2^{w-2}}$	$\mathcal{O}(n)$
wMOF, Sec. 4.2	2^{w-2}	$w+1$	$\mathcal{O}(w)$
l-t-r wNAF, Sec. 4.3	2^{w-2}	$w+1$	$\mathcal{O}(\log n),\ w=2$ $\mathcal{O}(\frac{\log w}{w}n),\ w>2$

Table 1. Comparison of Memory Requirement and Non-zero Density

5 Applications to Elliptic Curve Scalar Multiplication

Let $K = GF(p)$ be a finite field, where $p > 3$ is a prime. Let E be an elliptic curve over K. The elliptic curve E has an Abelian group structure with identity element $\mathcal{O}$ called the point of infinity. A point $P \in E$ is represented as $P = (x, y)$. The inverse of point $P = (x, y)$ is equal to $-P = (x, -y)$, hence it can be computed virtually for free. The elliptic curve additions $P_1 + P_2$ and $2P$ are denoted by ECADD and ECDBL, respectively, where $P_1, P_2, P \in E$.

As elliptic curves are written additively, exponentiation has to be understood as scalar multiplication. The familiar binary algorithms are adopted by computing ECADD instead of multiplying and ECDBL instead of squaring.

In general, we distinguish two main concepts of performing scalar multiplication: left-to-right and right-to-left. Here, d is represented as $d = \sum_{i=0}^{n} d_i 2^i,\ d_i \in \{0,1\},\ d_{n-1} = 1$.

Algorithm Binary Method, l-t-r
Input: $P;\ d = d_{n-1}|\dots|d_1|d_0$
Output: scalar multiplication dP
$Q \leftarrow\!\shortmid P$
for $i = n-2$ down to 0
  $Q \leftarrow\!\shortmid \mathrm{ECDBL}(Q)$
  if $d_i = 1$
    $Q \leftarrow\!\shortmid \mathrm{ECADD}(Q, P)$
return Q.

Algorithm Binary Method, r-t-l
Input: $P;\ d = d_{n-1}|\dots|d_1|d_0$
Output: scalar multiplication dP
$Q_1 \leftarrow\!\shortmid P;\ Q_2 \leftarrow\!\shortmid \mathcal{O}$
for $i = 0$ to $n-1$
  if $d_i = 1$
    $Q_2 \leftarrow\!\shortmid \mathrm{ECADD}(Q_2, Q_1)$
  $Q_1 \leftarrow\!\shortmid \mathrm{ECDBL}(Q_1)$
return Q_2.

Though in general both methods provide the same efficiency, the left-to-right method is preferable due to the following reasons:

1. The left-to-right method can be adjusted for general $\mathcal{T}$-representations of d like wNAF or wMOF in a more efficient way than the right-to-left method.
2. The ECADD step in the left-to-right method has the fixed input $tP,\ t \in \mathcal{T}$. Therefore it is possible to speed up these steps if tP is expressed in affine coordinates for each $t \in \mathcal{T}$, since some operations are negligible in this case. The improvement for a 160-bit scalar multiplication is about 15% with NAF over right-to-left scheme in the Jacobian coordinates [CMO98].
3. The right-to-left method needs an auxiliary register for storing $2^i P$.

5.1 Explicit Implementation for $w = 2$

In the following we show how the ideas of Section 4.2 lead to an efficient left-to-right scalar multiplication algorithm. For the sake of simplicity, we begin with the special case $w = 2$. The treatment for general width w can be found in Appendix B.

Let d be a binary string. The MOF and 2MOF representation of d are denoted by μ and δ, respectively. The proposed scheme scans the two bits of μ from the most significant bit, and if the sequences $1\bar{1}$ or $\bar{1}1$ appear, we perform the following conversions: $1\bar{1} \mapsto 01$ and $\bar{1}1 \mapsto 0\bar{1}$. Two consecutive bits of d determine the corresponding bit of MOF μ. Thus, three consecutive bits of d can generate the corresponding bit of the 2MOF δ. In order to find an efficient implementation, we discuss the relationship of bit representation among μ, δ, and d. The i-th bits of μ, δ, d are denoted by μ_i, δ_i, d_i, respectively. Because of the relation $\mu_i = d_{i-1} - d_i$, we know $\mu_i = 0$ if and only if $d_{i-1} = d_i$. The other 3-bit binary strings (d_i, d_{i-1}, d_{i-2}) where $d_{i-1} \neq d_i$ are only $(d_i, d_{i-1}, d_{i-2}) = (0,1,1), (1,0,0), (0,1,0), (1,0,1)$, corresponding to $(\delta_i, \delta_{i-1}) = (1,0), (-1,0), (0,1), (0,-1)$. Thus, there is a one-to-one map between (δ_i, δ_{i-1}) and (d_i, d_{i-1}, d_{i-2}) leading to the explicit Algorithm 6.

Algorithm 6 Explicit Left-to-Right Generation of 2MOF

Input: a non-zero n-bit binary string $d = d_{n-1}|d_{n-2}|...|d_1|d_0$
Output: 2MOF $\delta = \delta_n|\delta_{n-1}|...|\delta_1|\delta_0$ of d
  $d_{-1} \leftarrow\!\shortmid 0$
  $i \leftarrow\!\shortmid c + 1$ for the largest c with $d_c \neq 0$
  $\delta_n \leftarrow\!\shortmid 0;\ \delta_{n-1} \leftarrow\!\shortmid 0; \ldots;\ \delta_{i+1} \leftarrow\!\shortmid 0$
  while $i \geq 1$ **do**
    if $d_{i-1} = d_i$ **then**
      $\delta_i \leftarrow\!\shortmid 0;\ i \leftarrow\!\shortmid i - 1$
    else $\{d_{i-1} \neq d_i\}$
      $\delta_i \leftarrow\!\shortmid -d_i + d_{i-2};\ \delta_{i-1} \leftarrow\!\shortmid -d_{i-2} + d_{i-1};\ i \leftarrow\!\shortmid i - 2$
    if $i = 0$ **then**
      $\delta_0 \leftarrow\!\shortmid -d_0$
  return $\delta_n, \delta_{n-1}, ..., \delta_1, \delta_0$.

Finally, Algorithm 7 merges the recoding stage and evaluation stage of scalar multiplication.

Algorithm 7 Left-to-Right Scalar Multiplication Algorithm (On the Fly), $w = 2$

Input: a point P, a non-zero n-bit binary string $d = d_{n-1}|d_{n-2}|...|d_1|d_0$
Output: product dP
  $d_{-1} \leftarrow\!\shortmid 0;\ d_n \leftarrow\!\shortmid 0$
  $i \leftarrow\!\shortmid c + 1$ for the largest c with $d_c \neq 0$
  if $d_{i-2} = 0$ **then**
    $Q \leftarrow\!\shortmid P;\ i \leftarrow\!\shortmid i - 2$
  else $\{d_{i-2} = 1\}$
    $Q \leftarrow\!\shortmid$ECDBL$(P);\ i \leftarrow\!\shortmid i - 2$
  while $i \geq 1$ **do**
    if $d_{i-1} = d_i$ **then**
      $Q \leftarrow\!\shortmid$ECDBL$(Q);\ i \leftarrow\!\shortmid i - 1$
    else $\{d_{i-1} \neq d_i\}$
      $Q \leftarrow\!\shortmid$ECDBL(Q)
      if $(d_i, d_{i-2}) = (1, 1)$ **then**
        $Q \leftarrow\!\shortmid$ECDBL$(Q);\ Q \leftarrow\!\shortmid$ECADD$(Q, -P)$
      else if $(d_i, d_{i-2}) = (1, 0)$ **then**
        $Q \leftarrow\!\shortmid$ECADD$(Q, -P);\ Q \leftarrow\!\shortmid$ECDBL$(Q)$
      else if $(d_i, d_{i-2}) = (0, 1)$ **then**
        $Q \leftarrow\!\shortmid$ECADD$(Q, P);\ Q \leftarrow\!\shortmid$ECDBL$(Q)$
      else if $(d_i, d_{i-2}) = (0, 0)$ **then**
        $Q \leftarrow\!\shortmid$ECDBL$(Q);\ Q \leftarrow\!\shortmid$ECADD$(Q, P)$
      $i \leftarrow\!\shortmid i - 2$
  if $i = 0$ **then**
    $Q \leftarrow\!\shortmid$ECDBL$(Q);\ Q \leftarrow\!\shortmid$ECADD$(Q, -d_0P)$
  return Q.

The advantage of the previous algorithm is that it reduces the memory requirement since it does not store the converted representation of d.

6 Conclusion

It was an unsolved problem to generate a signed representation *left-to-right* for a general width w. In this paper we presented a solution of this problem. The proposed scheme inherits the outstanding properties of wNAF, namely the set of pre-computed elements and the non-zero density are same as those of wNAF. In order to achieve a left-to-right exponent recoding, we defined a new canonical representation of signed binary strings, called the mutual opposite form (MOF). An n-bit integer can be uniquely represented by $(n+1)$-bit MOF, and this representation can be constructed efficiently left-to-right. Then the proposed exponent recoding is obtained by applying the width w (left-to-right) sliding window conversion to MOF. The proposed scheme is conceptually easy to understand and it is quite simple to implement. Moreover, if we apply the width w (right-to-left) sliding window conversion to MOF, we surprisingly obtain the classical wNAF. This is the first *carry-free* algorithm for generating wNAF. Therefore the proposed scheme has a lot of advantages and it promises to be a good alternative to wNAF. We believe that there will be many new applications of this algorithms for cryptography.

References

[Avi61] Aviziensis, A., *Signed digit number representations for fast parallel arithmetic*, IRE Trans. Electron. Comput., 10:389-400, (1961).

[BSS99] Blake, I., Seroussi, G., and Smart, N., *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[BHLM01] Brown, M., Hankerson, D., Lopez, J., and Menezes, A., *Software Implementation of the NIST Elliptic Curves Over Prime Fields*, Topics in Cryptology - CT-RSA 2001, LNCS 2020, (2001), 250-265.

[Boo51] Booth, A., *A signed binary multiplication technique*, Journ. Mech. and Applied Math., 4(2), (1951), 236-240.

[CMO98] Cohen, H., Miyaji, A., and Ono, T., *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates*, Advances in Cryptology - ASIACRYPT '98, LNCS1514, (1998), 51-65.

[EK94] Egecioglu, Ö, and Koc, C, *Exponentiation using Canonical Recoding*, Theoretical Computer Science, 129(2), (1994), 407-417.

[Gor98] Gordon, D., *A survey of fast exponentiation methods*, Journal of Algorithms, vol.27, (1998), 129-146.

[Häg02] Häggström, O., *Finite Markov Chains and Algorithmic Applications*, London Mathematical Society Student Texts 52, Cambridge University Press, (2002).

[IEEE] IEEE P1363, Standard Specifications for Public-Key Cryptography. `http://groupe.ieee.org/groups/1363/`

[JM89] Jedwab, J., and Mitchell, C.J., *Minimum Weight Modified Signed-digit Representations and Fast Exponentiation*, Electronics Letters 25, (1989), 1171-1172.

[JY00] Joye, M., and Yen, S.-M., *Optimal Left-to-Right Binary Signed-digit Exponent Recoding*, IEEE Transactions on Computers 49(7), (2000), 740-748.

[Knu81] Knuth, D. E., *The art of computer programmming, vol. 2, Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, Mass. (1981).
[Kob87] Koblitz, N., *Elliptic Curve Cryptosystems*, Math. Comp. 48, (1987), 203-209.
[KT92] Koyama, K. and Tsuruoka, Y., *Speeding Up Elliptic Curve Cryptosystems using a Signed Binary Windows Method*, Advances in Cryptology - CRYPTO '92, LNCS740, (1992), 345-357.
[Mil86] Miller, V.S., *Use of Elliptic Curves in Cryptography*, Advances in Cryptology - CRYPTO '85, LNCS218, (1986), 417-426.
[MO90] Morain, F., Olivos, J., *Speeding Up the Computations on an Elliptic Curve using Addition-Subtraction Chains*, RAIRO Theoretical Informatics and Applications , 24, (1990), pp.531-543.
[MOC97] Miyaji, A., Ono, T., and Cohen, H., *Efficient Elliptic Curve Exponentiation*, Information and Communication Security, ICICS 1997, LNCS 1334, (1997), 282-291.
[MOV96] Menezes, A., van Oorschot, P. and Vanstone, S., *Handbook of Applied Cryptography*, CRC Press, 1996.
[Möl02] Möller, B., *Improved Techniques for Fast Exponentiation*, The 5th International Conference on Information Security and Cryptology (ICISC 2002), LNCS 2587, (2003), 298-312.
[OCo99] O'Connor, L., *An Analysis of Exponentiation Based on Formal Languages*, Advances in Cryptology - EUROCRYPT '99, LNCS1592, (1999), 375-388.
[Pro00] Prodinger, H., *On Binary Representations of Integers with Digits {-1,0-1}*, Integers: Electronic Journal of Combinatorial Number Theory 0, (2000)
[Rei60] Reitwiesner, G. W., *Binary arithmetic*, Advances in Computers, vol.1, (1960), 231-308.
[Sol00] Solinas, J.A., *Efficient Arithmetic on Koblitz Curves*, Designs, Codes and Cryptography, 19, (2000), 195-249.
[WMPW98] Win, E., Mister, S., Preneel, B., and Wiener, M., *On the Performance of Signature Schemes Based on Elliptic Curves*, Algorithmic Number Theory, ANTS-III, LNCS 1423, (1998), 252-266.

A Several Proofs

In this section we prove several propositions and theorems described in this paper.

Theorem 1. *Let n be a positive integer. $(n+1)$-bit MOF has 2^n pair-wise different representations. There is a bijective map between elements of $(n+1)$-bit MOF and n-bit binary string.*

Proof. We prove the theorem by induction of n. At first we prove the case of $n = 1$. The 2-bit MOF is either 00 or $1\bar{1}$. The 1-bit binary strings 0 and 1 are converted to MOFs 00 and $1\bar{1}$, respectively. Next, we assume that the theorem is correct for $n = 1, 2, ..., k$ and prove the case of $n = k + 1$. We classify it into two classes, namely $(k+1)$-th bit of binary string is 0 and 1. If the $(k+1)$-th bit of binary string is 0, we can assign the $(k+2)$-th bit of MOF as 0 and apply the one-to-one conversion of lower k bits. This case contains 2^k pair-wisely different

elements. If the $(k+1)$-th bit of binary string is 1, we have two additional cases, namely k-th bit of binary string is 0 or 1. We convert the $(k+1)$-bit binary string $10*$ and $11*$ to $(k+2)$-bit MOF $1\bar{1}*$ and $10*$, respectively. Then these elements are pair-wisely different (2^k elements) that are different from any elements of the previous cases, and thus there are 2^{k+1} pair-wise different representations. There is the one-to-one conversion for the lower $(k-1)$ bits. Thus we are able to construct a bijective map for $n = k+1$. Consequently, we proved the assertion of the theorem. □

Proposition 1. *The operation $\mu = 2d \ominus d$ converts binary string d to MOF μ.*

Proof. Wlog we assume $d \neq 0$ because otherwise the assertion is trivial. We prove the statement from the most significant bit. At first we show that the left-most non-zero bit of μ is 1. We know $\mu_n = 1$ if $d_{n-1} = 1$. The relationship $\mu_i = d_{i-1} - d_i$ yields $\mu_i = 0$ or 1 for $d_i = 0$. Thus the left-most non-zero bit of μ will be 1. Next, we prove that $\mu_{i-1} = 0$ or -1 holds for $\mu_i = 1$. From $\mu_i = d_{i-1} - d_i = 1$ we know $d_{i-1} = 1$, and thus $\mu_{i-1} = d_{i-2} - d_{i-1} = 0$ or -1 based on $d_{i-2} = 1$ or 0, respectively. This relationship yields $\mu_i|\mu_{i-1}|...|\mu_{i-k+1}|\mu_{i-k} = 1\underbrace{0,..,0}_{k-1}\bar{1}$ for some k. Similarly, if $\mu_i = -1$, then there is some integer k' such that $\mu_i|\mu_{i-1}|...|\mu_{i-k'+1}|\mu_{i-k'} = \bar{1}|\underbrace{0|\ldots|0}_{k'-1}|1$. It remains to show that the least non-zero bit of μ is negative. Let b be the last non-zero bit of the binary string d, namely $d_b = 1$ and $d_{b-1} = d_{b-2} = ... = d_1 = d_0 = 0$. In this case, we know $\mu_b = -1$ and $\mu_{b-1} = \mu_{b-2} = ... = \mu_1 = \mu_0 = 0$. Consequently the converted signed string μ satisfies the conditions characterizing MOF. □

Theorem 2. *Every non-negative integer d has a representation as wNAF, which is unique except for the number of leading zeros.*

Proof. We show (ignoring leading zeros in the following) that Definition 1 leads to a unique representation of positive integers as follows:

1. We define a conversion MOF $\mapsto$ wNAF.
2. We define a conversion wNAF $\mapsto$ MOF.
3. We show, that these two conversion are inverse to each other.

Consequently, there is a bijection between wNAF and MOF. As there is also a bijection between MOF and Binary, this proves the uniqueness of wNAF.

ad (1): MOF $\mapsto$ wNAF is defined by performing the sliding window method with width w from the least significant bit (i.e. right-to-left) on MOF as described in Section 4.1.

ad (2): wNAF $\mapsto$ MOF scans the bits right-to-left using a window with width w, until the LSB in the window is non-zero. All scanned zeros are taken to the converted string as usual. If the LSB in the window is non-zero, due to the properties of wNAF the window content is of the shape $\underbrace{0\ldots0}_{w-1}d, d \in \{\pm1, \pm3, \ldots, \pm2^{w-1}\}$. To find the correct replacement, we distinguish two cases:

Case 1 The most significant non-zero bit of the already converted string equals -1:
We build the length-w-MOF corresponding to $|d|$ (padded with leading zeros, if necessary). In the case of $d < 0$, we change all signs of this length-w-MOF and take this string. Otherwise, we force the last bit of the length-w-MOF to be 1 by replacing its last 2 bits as follows: $1\bar{1} \mapsto 01,\ 0\bar{1} \mapsto \bar{1}1$.

Case 2 The most significant non-zero bit of the already converted string equals 1 or no non-zero bit has been converted at all:
We build the length-w-MOF corresponding to $|d|$ (padded with leading zeros, if necessary). In the case of $d > 0$, we take this string. Otherwise, we change all signs of this length-w-MOF and we force the last bit of this string to be $\bar{1}$ by replacing its last 2 bits as follows: $01 \mapsto 1\bar{1},\ \bar{1}1 \mapsto 0\bar{1}$.

This case-differentiation ensures that the converted string possesses the MOF properties (particularly the alternating signs of the non-zero bits).

Example: In the case of $w = 3$, we use the following table for the right-to-left conversion

$$001 \mapsto \begin{cases} 001 & \text{Case1} \\ 01\bar{1} & \text{Case2} \end{cases} \quad 00\bar{1} \mapsto \begin{cases} 0\bar{1}1 & \text{Case1} \\ 00\bar{1} & \text{Case2} \end{cases}$$

$$003 \mapsto \begin{cases} 1\bar{1}1 & \text{Case1} \\ 10\bar{1} & \text{Case2} \end{cases} \quad 00\bar{3} \mapsto \begin{cases} \bar{1}01 & \text{Case1} \\ \bar{1}1\bar{1} & \text{Case2} \end{cases}$$

ad (3): The two conversions are inverse to each other, because if we perform e.g. MOF $\mapsto$ wNAF and wNAF $\mapsto$ MOF afterwards, the fragmentations of the strings are exactly the same and the tables are inverse to each other. □

Theorem 3. *Every non-negative integer d has a representation as wMOF, which is unique except for the number of leading zeros.*

Proof. We proceed as follows:

1. We define a conversion MOF $\mapsto$ wMOF.
2. We define a conversion wMOF $\mapsto$ MOF.
3. We show, that these two conversion are inverse to each other.

Consequently, there is a bijection between wMOF and MOF. As there is also a bijection between MOF and Binary, this proves the uniqueness of wMOF. This proof is similar to the preceding one (MOF $\leftrightarrow$ wNAF).

ad (1): MOF $\mapsto$ wMOF is defined by performing the sliding window method with width w from the most significant bit (i.e. left-to-right) on MOF.

ad (2): wMOF $\mapsto$ MOF scans the bits right-to-left using a window with width w, until the window content equals one of the patterns described in Definition 3, property 2. All scanned zeros are taken to the converted string as usual. The replacements are performed as in the preceding proof with the following

exceptions: If the window content equals $1\underbrace{0\ldots0}_{w-1}$ and case 1 applies, then we adopt the content as it stands. But if case 2 applies, it follows from Definition 3, property 2, that the left-hand neighbor of the window must be a zero. In this case we shift the window one step leftwards and replace $01\underbrace{0\ldots0}_{w-2}$ by $1\bar{1}\underbrace{0\ldots0}_{w-2}$. The dual case ($\bar{1}$ instead of 1) is treated in the analogue way.
As before the case differentiation ensures the MOF properties.

ad (3): We can argue exactly in the same way as in the preceding proof. □

Theorem 4. *The average non-zero density of wMOF is asymptotically* $1/(w+1)$ *for* $n \mapsto \infty$.

Proof. Fig. A shows Markov chain version of the proposed scheme with width w. First, we define a conversion $\delta_w(\cdot)$, and then we explain the Markov chain. After that, we prove the theorem.

The conversion $\delta_w(\cdot)$ converts a non-zero integer u in the range $[-2^{w-1}+1, 2^{w-1}-1]$ to a width w converted MOF $\delta = \delta_{w-2}|\delta_{w-3}|...|\delta_0$ as follows:

$$\delta_w(u) = \underbrace{0|0|...|0}_{w-2-\alpha}|u/2^\alpha|\underbrace{0|0|...|0}_{\alpha},$$

where α is the largest integer j such that u is divisible by 2^j.

Next, we describe the Markov chain. The states of the Markov chain are (d_{i-1}, u) ($d_{i-1} \in \{0,1\}, u \in [-2^{w-1}+1, 2^{w-1}-1]$), where u is a buffer of scanned bits. Thus, there are $2(2^w-1)$ states. The trigger of transition is d_{i-2}, and the flow outputs width w converted MOF δ_i according to a current state. Generally speaking, if the current state is (d_{i-1}, u), the next state is $(d_{i-2}, 2u + (d_{i-2} - d_{i-1}))$. If the transition exceeds the horizontal dotted line in Fig. A, the flow outputs $\delta_w(u)$ as the width w converted MOF δ_i, and the buffer u is cleared. Note that the states $(0,0), (1,0), (1,1), (0,-1)$ under and over the horizontal dotted line are the same states, respectively.

Next, we describe the transitions in detail. We have three cases; $(0,0)$ and $(1,0)$, $(1, 2^{v-1})$ and $(0, -2^{v-1})$, and a general (d_{i-1}, u). First, if the current state is $(0,0)$ or $(1,0)$, the flow outputs $\delta_i = 0$ and goes to the state $(d_{i-2}, d_{i-2} - d_{i-1})$. In the case of the general state (d_{i-1}, u), the next state (d', u') is $(d_{i-2}, 2u + (d_{i-2} - d_{i-1}))$ if the updated u' is in the range $[-2^{w-1}+1, 2^{w-1}-1]$. If not, the flow outputs $\delta_w(u)$ as the width w converted MOF $\delta_{i-w+1}|...|\delta_i$, and goes to the state $(d_{i-2}, d_{i-2} - d_{i-1})$. In the case of $(1, 2^{v-1})$ (resp. $(0, -2^{v-1})$), the transition is the same as the general state, exclusive of the following: If $d_{i-2} = 0$ (resp. $d_{i-2} = 1$), the flow outputs $\delta_{i-v+1} = 0$. The initial state is (d_{n-1}, d_{n-1}), and i is initialized as $i = n$. The terminal condition is $i = 0$, and $d_{-1} = 0$ is appended such as the case of $w = 2$. If $i = 0$, the flow outputs $\delta_v(u)$ and terminates.

We prove the theorem using this Markov chain. Since the Markov chain is aperiodic and irreducible, there exists the stationary distribution. The non-zero density is the probability of outputting $\delta_w(u)$, because non-zero δ_i corresponds to $\delta_w(u)$. Easy calculation induces the non-zero density is $1/(w+1)$. □

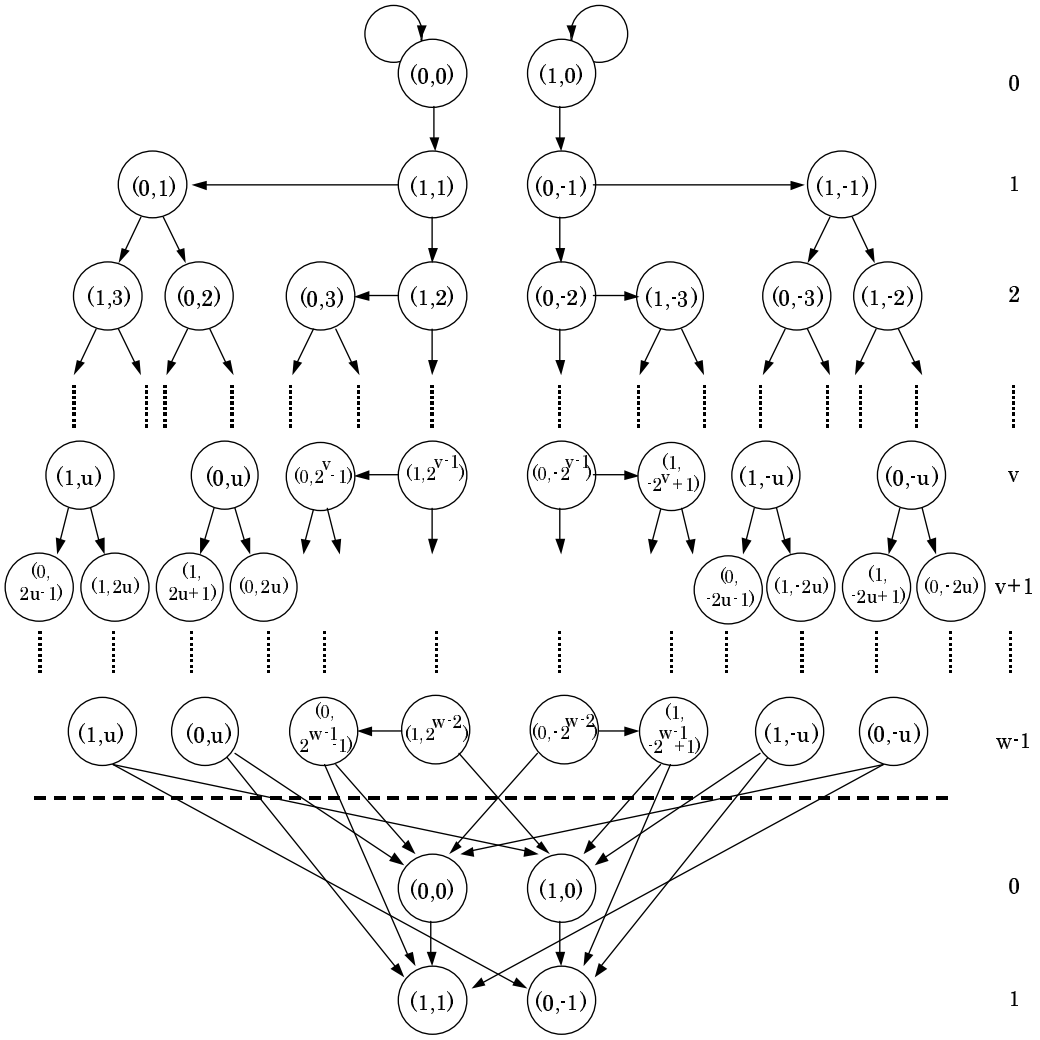

Fig. 1. Markov chain for width w

In order to confirm the non-zero density of Theorem 4 for 160-bit binary strings, we show the result of converting 160-bit binary strings where each bit is randomly chosen. The non-zero density is computed by dividing the number of non-zero digits through 160. The following table shows the inverse density (1/density) for easier comparison with expected values.

width w	1/observed density	1/expected density
2	2.988	3
3	3.970	4
4	4.946	5
5	5.914	6
6	6.878	7

Table 2. Average inverse non-zero density.

The observed non-zero density approximates the expected density. It is conspicuous that the observed density is higher than the expected one. The reason for this is that the length of the last converted window may be smaller than w, and thus not optimal. The above values are connected with bit-length of 160 bits. For longer bit-strings the observed density comes closer to the expected density. This experimental result confirms Theorem 4.

B Implementation Details for the Proposed Scheme, $w > 2$

In this section we discuss an explicit implementation for the proposed left-to-right window chain with general width w. This implementation is divided into two parts. The first one describes an algorithm for table-computation. This table is used by the main algorithm described in the second section and by the on-the-fly multiplication.

Table Computation

The conversion method performs many table look-ups, so this must be done in an efficient way. The memory usage is also important because devices like smart cards have rare resources. The following algorithm tries to consider these facts.

We use a table representation in Section 5.1. All columns from the original table appear in this one: the original bits d, the MOF μ and the converted one δ. The table is sorted by column d. Column d_{dec} is the decimal representation of the binary string d. This would be a good table index but unfortunately it does not start from 0. We have to subtract 2^{w-1} from d_{dec} to get the table index. All rows, which start with 0,0 or 1,1, are ignored because they are not converted. Column μ_{dec} is the decimal representation of μ, below named as c. The table computation algorithm has to compute γ and ξ which fit the equation $c = \gamma * 2^{\xi}$. Column ν can be recreated from γ and ξ: $\delta = (\underbrace{0, ..., 0}_{w-\xi-1}, \gamma, \underbrace{0, ..., 0}_{\xi})$ but it is not

necessary for the algorithm. For the implementation only columns γ and ξ are essential.

d	d_{dec}	$d_{dec} - 2^{w-1}$	μ	$\mu_{dec} = c$	γ	ξ	δ
(1,1*) or (0,0,*)							
(0,1,0,0)	4	0	(1,-1,0)	$2 = 1 * 2^1$	1	1	(0,1,0)
(0,1,0,1)	5	1	(1,-1,1)	$3 = 3 * 2^0$	3	0	(0,0,3)
(0,1,1,0)	6	2	(1,0,-1)	$3 = 3 * 2^0$	3	0	(0,0,3)
(0,1,1,1)	7	3	(1,0,0)	$4 = 1 * 2^2$	1	2	(1,0,0)
(1,0,0,0)	8	4	(-1,0,0)	$-4 = -1 * 2^2$	-1	2	(-1,0,0)
(1,0,0,1)	9	5	(-1,0,1)	$-3 = -3 * 2^0$	-3	0	(0,0,-3)
(1,0,1,0)	10	6	(-1,1,-1)	$-3 = -3 * 2^0$	-3	0	(0,0,-3)
(1,0,1,1)	11	7	(-1,1,0)	$-2 = -1 * 2^1$	-1	1	(0,-1,0)

Table 3. Conversion table for $w = 3$

Next we present the table creation algorithm for any w. The algorithms use some bit-string operators. & is "bitwise and", XOR is "exclusive or" operation. The operator >> performs right-shift that is equivalent to a division by a power of two.

Table Computation with Width w
INPUT: width w.
OUTPUT: arrays $\gamma_{0...tw}$ and $\xi_{0...tw}$ where $tw = 2^w - 1$.

1. For $d \leftarrow\!\!\!\shortmid 2^{w-1}$ to $3 * 2^{w-1} - 1$ do the following
   1.1. $c \leftarrow\!\!\!\shortmid (d \;\&\; (2^w - 1)) - (d >> 1)$
   1.2. $\xi_{d-2^{w-1}} \leftarrow\!\!\!\shortmid 0$
   1.3. While $(c \;\&\; 1) = 0$ do the following
      1.3.1. $\xi_{d-2^{w-1}} \leftarrow\!\!\!\shortmid \xi_{d-2^{w-1}} + 1$
      1.3.2. $c \leftarrow\!\!\!\shortmid c >> 1$
   1.4. $\gamma_{d-2^{w-1}} \leftarrow\!\!\!\shortmid c$
2. return $\gamma_{0...tw}$ and $\xi_{0...tw}$

The loop in step 1 passes all values for d_{dec} like in the table. To use it as table index 2^{w-1} has to be subtracted. Step 1.1 computes c. The first expression of the difference is d without the first bit. The second expression is d right-shifted by a distance of 1. The illustration shows a conversion for $d = (1, 0, 1, 0) = 10_{dec}$.

$$\begin{array}{rr} 1 \mid 0\ 1\ 0 \mid & 2 = 10 \;\&\; 7 \\ - \quad \mid 1\ 0\ 1 \mid 0 & - \;\; 5 = 10 >> 1 \\ \hline -1\ 1\ -1 & -3 \end{array}$$

Step 1.2 initializes ξ with zero. The loop 1.3 divides c until it is an odd number. If c is even ξ is increased by one in step 1.3.1 and c is divided by two in step 1.3.2. This is correct because of the following equation: $\gamma/2 * 2^{\xi+1} = \gamma * 2^{\xi}$. When the loop terminates c is assigned to γ. Step 2 returns arrays γ and ξ.

We will estimate the table size. In general table width w the resulting table has 2^w rows. ξ is in $\{0, 1, ..., w-1\}$ which are w different values. $\lceil \log_2 w \rceil$ bits are required to store ξ. Each element of γ is in $\{\pm 1, \pm 3, ..., \pm(2^{w-1} - 1)\}$ which

has the cardinality of 2^{w-2} and requires $w-2$ bits. The whole table has a size of $2^w(\lceil \log_2 w \rceil + w - 2)$ bits. For example, the bit sizes for small $w = 3, 4, 5, 6$ are 24, 64, 192, 448, respectively.

In applications w is often fixed or has only a few different values. Thus the table can be pre-computed and stored in a read only memory because it is independent from runtime parameters.

Main Algorithm using table-lookup

The explicit algorithm below differs in the table-lookup part from the original.

Explicit Algorithm with Width w
INPUT: width w, a non-zero n-bit integer $d = d_{n-1}|d_{n-2}|...|d_1|d_0$,
the pre-computed table $\gamma_{0...tw}$ and $\xi_{0...tw}$.
OUTPUT: width w converted MOF $\delta = \delta_n|\delta_{n-1}|...|\delta_1|\delta_0$ of d.

1. $i \leftarrow\!\!\!| \; n$, $\delta_{0...n} \leftarrow\!\!\!| \; 0$
2. While $i \geq 1$ do the following
   2.1. if $(d_i$ XOR $d_{i-1}) = 0$, then set $i \leftarrow\!\!\!| \; i - 1$, else do the following
      2.1.1. $index \leftarrow\!\!\!| \; ((d >> (i - w)) \;\&\; (2^{w+1} - 1)) - 2^{w-1}$
      2.1.2. $\delta_{i-w+1+\xi_{index}} \leftarrow\!\!\!| \; \gamma_{index}$
      2.1.3. $i \leftarrow\!\!\!| \; i - w$
3. if $i = 0$ and $d_0 = 0$, then set $\delta_0 \leftarrow\!\!\!| \; -1$
4. return $\delta_n, \delta_{n-1}, ..., \delta_1, \delta_0$.

The first step initializes the loop-variable i to the highest bit of d. Step 2 assigns zero to all result variables sd_i, because later we proceed only result variables disparate zero.

At Step 1 we initialize the parameters. While i is greater than or equal to one, the loop is executed. The bit equality condition of the proposed algorithm is implemented by a xor-operation in Step 2.1. The main difference is in Step 2.1.1 where the table index is created. Instead of searching bits in the table, the algorithm masks out the relevant bits to use it as table index. d is right-shifted to get the bits beginning at position i. The and-mask $2^{w+1} - 1$ will return the rightmost $w + 1$ bits. Finally 2^{w-1} has to be subtracted because of the shifted table-index. After that we can access to the corresponding table row. The result variable δ_i is initialized with zero, thus only one assignment has to be done in Step 2.1.2. All masked bits from Step 2.1.1 are equal to γ shifted by a distance of ξ. The index $i - w + 1$ of δ is the rightmost position of the part we want to replace. The shift distance ξ added to assign γ at the correct position. Then i is decreased by w. Step 3 is necessary because the loop condition is $i \geq 1$ instead of $i \geq 0$. This avoids negative indices. In Step 4 the result is returned.

B.1 On the Fly Multiplication for $w > 2$

The following algorithm merges the exponent recoding and a scalar multiplication for any w. It is based on the table created by the table computation algorithm in this section.

Scalar Multiplication Algorithm with Width w

INPUT a non-zero n-bit binary string d, a point P and the multiple of the point P, $\gamma_{0...tw}$ and $\xi_{0...tw}$, the precomputed table.

OUTPUT scalar multiplication dP.

1. $i \leftarrow\!\!\!\shortmid\ n$
2. $Q \leftarrow\!\!\!\shortmid\ \mathcal{O}$
3. While $i \geq 1$ do the following
   - 3.1. if $(d_i$ XOR $d_{i-1}) = 0$, then do the following
     - 3.1.1. $Q \leftarrow\!\!\!\shortmid$ECDBL$(Q)$
     - 3.1.2. $i \leftarrow\!\!\!\shortmid\ i - 1$
   - 3.2. else do the following
     - 3.2.1. $index \leftarrow\!\!\!\shortmid\ ((d >> (i - w))\ \&\ (2^{w+1} - 1)) - 2^{w-1}$
     - 3.2.2. For $j = 1$ to $w - \xi_{index}$ do the following
       1. $Q \leftarrow\!\!\!\shortmid$ ECDBL(Q)
       2. $i \leftarrow\!\!\!\shortmid\ i - 1$
     - 3.2.3. $Q \leftarrow\!\!\!\shortmid$ECADD$(Q,\gamma_{index}P)$
     - 3.2.4. For $j = 1$ to ξ_{index} do the following
       1. If $i \geq 0$ then $Q \leftarrow\!\!\!\shortmid$ ECDBL(Q)
       2. $i \leftarrow\!\!\!\shortmid\ i - 1$
4. If $i = 0$ do the following
   - 4.1. $Q \leftarrow\!\!\!\shortmid$ ECDBL(Q)
   - 4.2. If $d_0 = 0$ then $Q \leftarrow\!\!\!\shortmid$ECADD$(Q,-P)$
5. return Q

Step 1 initializes the loop variable i to the bit-length n. The second step initializes the result variable Q to $\mathcal{O}$. This is done to avoid a special case for the first digit. While i is greater or equal to 1 the loop is executed. The condition in Step 3.1 checks whether the two adjacent bits equals or not. If they have the same value, Q is doubled and i is decreased. This correspondent to a zero in the recoding algorithm, where the result is not changed. If the else-part in Step 3.2 is executed, the following step creates a table index. This is the same index like in the algorithm above. After that we can receive γ and ξ from the table. In this whole else-part w ECDBL and one ECADD operations have to be done. ξ constitutes how many doubling-operations are done before the addition. Step 3.2.2 doubles Q $w - \xi$ times. The next step adds a multiple of P to Q. These multiples of P are precomputed values and an input of this algorithm. The loop in Step 3.2.4 does the remaining doubling-operations. The condition in Step 3.2.4.1 avoids supernumerary doublings. Altogether i is decreased w times. Step 4 checks if i is zero to avoid negative indices, like in the algorithm above. It doubles the result-value Q and subtracts P from Q if the last bit is zero. The last step returns the result Q.