Algorithm and Implementation of Signed-Binary Encoding with Asymmetric Digit Sets for Elliptic Curve Cryptosystems

Xiaoyu Ruan, Rajendra Katti, and David Hinkemeyer
Department of Electrical and Computer Engineering
North Dakota State University, Fargo, ND 58105-5285
E-mail:{xiaoyu.ruan, rajendra.katti, david.hinkemeyer}@ndsu.edu

Abstract—Signed-binary representations of integer \( k \) with symmetric digit set \( \mathcal{D}_s = \{-(2^w - 1), -(2^w - 3), \cdots, -1, 0, 1, \cdots, 2^w - 3, 2^w - 1\} \) may have lower weight than the unsigned-binary expansion of \( k \). The “weight” is the number of nonzero digits in a binary expansion. Lower weight leads to fewer number of addition operations in the scalar multiplication, \( kP \), of elliptic curve cryptosystems. Here \( P \) is a point on an elliptic curve. On the other hand, computing the minimum-weight signed-binary representation from left (most significant bit) to right (least significant bit) significantly reduces memory requirements because intermediate results do not need to be stored. Since the size of \( \mathcal{D}_s \) is \( 2^w + 1 \), a \((w + 1)\)-bit data bus is necessary to represent the \( 2^w + 1 \) elements in \( \mathcal{D}_s \). This is inefficient because a \((w + 1)\)-bit bus is capable of denoting \( 2^{w+1} \) cases. We present a new signed-binary recoding algorithm with asymmetric digit set \( \mathcal{D}_a = \{-(2^w - 1), -(2^w - 3), \cdots, -1, 0, 1, \cdots, 2^w - 3\} \). For \( w = 2 \), our simulation results show that the average weight of signed-binary numbers with digit set \( \{-3, -1, 0, 1\} \) is 0.285 times the length of their unsigned-binary expansions. For the optimal representations with \( \{-1, 0, 1\} \) the average ratio is 0.333. The number of additions is decreased by 14.4%. The encoding circuit requires 7 flip-flops and 22 gates to realize.

I. INTRODUCTION

A. Elliptic Curve Cryptosystems

Elliptic curve cryptosystems (ECCs) were first proposed independently by Victor Miller [1] and Neal Koblitz [2] in the mid-1980s. A good survey is written by Lopez and Dahab [3]. The wide usability of the curve-based cryptography is discussed in [4]. At a high level, ECCs are analogs of existing public-key cryptosystems in which modular arithmetic is replaced by operations defined over elliptic curves. The security depends on the following hard problem:

Given two points \( G \) and \( P \) on an elliptic curve such that \( G = kP \) (\( G \) is \( P \) added to itself \( k \) times), find the integer \( k \).

This problem is commonly referred to as the elliptic curve discrete logarithm problem that needs fully exponential time for solution. The methods for computing general elliptic curve discrete logarithms are generally much less efficient than those for factoring or computing conventional discrete logarithms. As a result, shorter key sizes can be used to achieve the same security of conventional public-key cryptosystems, which might lead to better memory requirements, improved performance, and low hardware overhead.

At the encryption end, the main operation is the computation of \( kP \), also known as scalar multiplication, where \( k \) is a large integer and \( P \) a point on an elliptic curve. In the elliptic curve group, the negative value of a point \( P \) (i.e., \(-P\)) can be obtained from \( P \) at negligible cost. This property can be used to speed up scalar multiplication given that a proper binary representation of \( k \) is adopted.

B. Double-and-Add Algorithm

For computing \( G = kP \), integer \( k \) is denoted by its binary form and is scanned from left (the most significant bit) to right (the least significant bit). Table I shows a trivial example for \( k = 103 \). \( G \) is initialized to 0. The unsigned-binary expansion of \( k \) is written in the first row. If the digit being scanned is 1 then \( G \leftarrow G + P \) and the result is recorded in the row labelled \(+P\), otherwise \( G \) is left unchanged. \( G \) is doubled when the scan process moves one bit rightwards. The final result of \( G \) is indicated by the bottommost entry of the rightmost column. For this example, 5 additions are necessary because there are 5 ones in 1100111.

C. Signed-Binary Representation with Digit Set \( \{-1, 0, 1\} \)

The signed-binary representation uses \( \mathcal{D}_a = \{-1, 0, 1\} \) (\(-1\) is also referred to as \( T \) to recode integers, i.e., a signed-binary representation of integer \( k \) is denoted as

\[ k = (\cdots u_3 u_2 u_1 u_0). \]

Here \( k = \cdots + u_3 2^3 + u_2 2^2 + u_1 2^1 + u_0 \), and \( u_i \in \{T, 0, 1\} \) for all \( i \)'s. It is not difficult to see that signed-binary representation of a given non-negative integer is not unique. For
example, 1010100T and 10100111 are two of many signed-binary representations of 103. Of course, the normal binary expansion 0110111 can also be considered as a special kind of signed-binary representation. The concept of signed-binary is introduced to ECC since \(-P\) is available.

Table II shows the process of computing \(G = 103P\) by applying the signed-binary representation 1010100T in the double-and-add algorithm. If some bit in \(k\) is \(-1\) then \(G \leftarrow G + (-P)\) and the result is written in the row labelled \(-P\). By using the new signed expansion of \(k\), the number of additions is reduced to 4 because there exist 4 nonzero digits in 1010100T.

For a signed-binary representation, define its weight to be the number of nonzero digits. The number of additions needed for obtaining \(G = kP\) is equal to the weight of the representation of \(k\). Therefore it is important to find a minimum-weight signed-binary representation of \(k\) in order to minimize the number of additions. Moreover, computing the minimum-weight signed-binary representation of \(k\) from left to right is more advantageous than from right to left because, the double-and-add algorithm operates from left to right. In this manner, computing the signed-binary representation of \(k\) and computing \(G = kP\) can be performed in tandem. Consequently, the intermediate expansion of \(k\) does not need to be stored in memory. Note that in ECCs \(k\) is usually a long integer.

Two best-known approaches for obtaining minimum-weight signed-binary representations are non-adjacent form (NAF) [5] and the one introduced in [6]. The former is a right-to-left method while the latter is left-to-right. However, the property of no two consecutive digits being nonzero does not hold for the representation in [6]. Both methods result in an average weight of 0.333L where \(L\) is the number of bits, of all unsigned-binary expansions. Recall that the average weight for all unsigned-binary expansions is 0.5L. Later on the concept of weight was extended to joint weight\(^1\) for more than one integer. For computing a minimum-joint-weight signed-binary representation of two integers, Solinas [7] presents a right-to-left approach called joint sparse form (JSF), which was extended to \(n\) numbers by Proos in [8]. In [9] we proposed a left-to-right method for two integers and this was extended to the case of \(n\) integers in [10] and [11].

\(^1\)Joint weight is the number of nonzero columns when \(n\) signed-binary representations are written one below another.

### Algorithm 1 Computing Intermediate Signed-Binary Representation from Unsigned-Binary Expansion from Left to Right

**Input:** Unsigned-binary expansion \((\eta_0, \eta_1, \ldots, \eta_0)\) of \(k\).

**Output:** Intermediate signed-binary representation \((\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_0)\) of \(k\).

\[
\begin{align*}
\eta_j &\leftarrow 0 \\
\eta_{j-1} &\leftarrow 0 \\
\text{for } J &\geq j \geq 0 \text{ do} \\
\varepsilon_j &\leftarrow \eta_{j-1} - \eta_j
\end{align*}
\]

We introduce a simple algorithm for computing a minimum binary expansion of integer \(k\) with digit set \(D_s = \{-2^w - 1, -2^w - 3, \ldots, -1, 0, 1, \ldots, 2^w - 3, 2^w - 1\}\) from left to right. It is proved in [11] that the resulting expansion may be different from \(w\)-NAF, but it has the same weight as \(w\)-NAF. We therefore name it “equivalent \(w\)-NAF”. The conversion consists of two steps. The unsigned-binary expansion of \(k\) is first converted to a so-called “intermediate signed-binary representation” (see Algorithm 1), which is then input to Algorithm 2. The output of Algorithm 2 is a minimum-weight binary expansion of integer \(k\) with symmetric digit set \(D_s = \{-2^w - 1, -2^w - 3, \ldots, -1, 0, 1, \ldots, 2^w - 3, 2^w - 1\}\). We say \(D_s\) is “symmetric” because if \(\alpha \in D_s\) then \(-\alpha \in D_s\).

Consider \(k = 103\) and \(w = 2\) as an example. In this case \(D_s = \{-3, -1, 0, 1, 3\}\). Before computing \(G = kP\), the values of \(-3P, -P, P, +P, +3P\) must be precomputed and stored in memory. After applying Algorithm 1 and Algorithm 2 sequentially we get \(k = 30100T\). The operation of the double-and-add algorithm is explained in Table III. The number of additions is 3.

### II. Equivalent \(w\)-NAF

If more digits are allowed in recoding, obviously the weight of a binary expansion can be further reduced. The cost for this is the precomputations of \(\alpha P\) where \(\alpha \in D_s\).

Let \(w \geq 1\) be an integer. The \(w\)-NAF of an integer \(k\) is the unique binary expansion with digits in \(D_s = \{-2^w - 1, -2^w - 3, \ldots, -1, 0, 1, \ldots, 2^w - 3, 2^w - 1\}\) such that any two nonzero digits are separated by at least \(w\) zeros. In the case \(w = 1\), the 1-NAF is usually called NAF [5]. The \(w\)-NAF can be computed from right to left by selecting the rightmost digit according to \(n \mod 2^{w+1}\). Avanzi [12] showed that the \(w\)-NAF has minimum weight amongst all signed-binary representations with digits of absolute value less than \(2^w\).

### III. Asymmetric \(w\)-NAF

We were trying to implement Algorithm 2 in hardware when we realized that a \((w + 1)\)-bit output data bus is necessary to represent all digits in \(D_s = \{-2^w - 1, -2^w - 3, \ldots, -1, 0, 1, \ldots, 2^w - 3, 2^w - 1\}\) since \(|D_s| = 2^{w+1}\). This is inefficient due to the fact that a \((w + 1)\)-bit bus is capable of denoting up to \(2^{w+1}\) cases. To cover this drawback, by modifying Algorithm 2 we came up with a new algorithm.
Algorithm 2 Computing a Minimum Binary Expansion with \( D_s = \{-(2^w-1), -(2^w-3), \ldots, -1, 0, 1, \ldots, 2^w-3, 2^w-1\} \) from Left to Right

**Input:** Intermediate signed-binary representation \((\eta_j, \ldots, \eta_0)\) of \(k; \ w \geq 1\)

**Output:** Equivalent \(w\)-NAF \(\varepsilon\) of \(k\)

\[
\begin{align*}
\varepsilon & \leftarrow 0 \\
j & \leftarrow J \\
\text{while } j \geq 0 \text{ do} \\
\quad \text{if } \eta_j = 0 \text{ then} \\
\quad \quad j & \leftarrow j - 1 \\
\quad \text{else} \\
\quad \quad t & \leftarrow \text{max}(j - w, 0) \\
\quad \quad \text{while } \eta_t = 0 \text{ do} \\
\quad \quad \quad t & \leftarrow t + 1 \\
\quad \quad \text{end while} \\
\quad \quad \varepsilon_t & \leftarrow \sum_{\ell=t} ^j \eta_\ell 2^{\ell-t} \\
\quad \quad j & \leftarrow j - w - 1 \\
\quad \text{end if} \\
\text{end while}
\end{align*}
\]

<table>
<thead>
<tr>
<th>(k = )</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>(\text{T})</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>0</td>
<td>6(P)</td>
<td>12(P)</td>
<td>26(P)</td>
<td>52(P)</td>
<td>104(P)</td>
</tr>
<tr>
<td>(+P)</td>
<td>(-P)</td>
<td>13(P)</td>
<td>103(P)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+\mbox{3P})</td>
<td>3(P)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-\mbox{3P})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III**

Using Equivalent 2-NAF and Double-and-Add Algorithm to Compute 103\(P\)

Algorithm 3 Computing a Minimum Binary Expansion with \( D_s = \{-(2^w-1), -(2^w-3), \ldots, -1, 0, 1, \ldots, 2^w-3\} \) from Left to Right

**Input:** Intermediate signed-binary representation \((\eta_j, \ldots, \eta_0)\) of \(k; \ w \geq 1\)

**Output:** Asymmetric \(w\)-NAF \(\varepsilon\) of \(k\)

\[
\begin{align*}
\varepsilon & \leftarrow 0 \\
j & \leftarrow J \\
e & \leftarrow 0 \\
\text{while } j \geq 0 \text{ do} \\
\quad \text{if } \eta_j = 0 \text{ then} \\
\quad \quad j & \leftarrow j - 1 \\
\quad \text{else} \\
\quad \quad t & \leftarrow \text{max}(j - w, 0) \\
\quad \quad \text{while } \eta_t = 0 \text{ do} \\
\quad \quad \quad t & \leftarrow t + 1 \\
\quad \quad \text{end while} \\
\quad \quad \varepsilon_t & \leftarrow \sum_{\ell=t} ^j \eta_\ell 2^{\ell-t} \\
\quad \quad j & \leftarrow j - w - 1 \\
\quad \text{end if} \\
\text{end while}
\end{align*}
\]

<table>
<thead>
<tr>
<th>(k = )</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>(\text{T})</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>0</td>
<td>2(P)</td>
<td>4(P)</td>
<td>8(P)</td>
<td>16(P)</td>
<td>26(P)</td>
<td>52(P)</td>
<td>104(P)</td>
</tr>
<tr>
<td>(+P)</td>
<td>(-P)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\mbox{3P})</td>
<td></td>
<td>13(P)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| \(-\mbox{3P}\) |

**TABLE IV**

Using Asymmetric 2-NAF and Double-and-Add Algorithm to Compute 103\(P\)

that employs asymmetric digit set \(D_a = \{-(2^w-1), -(2^w-3), \ldots, -1, 0, 1, \ldots, 2^w-3\}\) for recoding. The pseudo code is given in Algorithm 3. We name the new representation “asymmetric \(w\)-NAF”. Since \(|D_a| = 2^w\), a \(w\)-bit data bus is used to represent all elements in \(D_a\).

For example, let \(k = 103\) and \(w = 2\), then \(D_a = \{-3, -1, 0, 1\}\). Precomputations of \(-3\(P\), \(-P\), and \(+P\) are required. Note that \(+\mbox{3P}\) does not need to be precomputed. By sequentially applying Algorithm 1 and Algorithm 3 to \(k\) we have the asymmetric 2-NAF 10000\(\overline{001}\)\(T\). Then the double-and-add method is applied to obtain 103\(P\) as shown in Table IV. The number of additions is 3, the same as in Table III. However using asymmetric 2-NAF saves one precomputation and its storage space. Furthermore, the equivalent 2-NAF requires a 3-bit output data bus, whereas the asymmetric 2-NAF only needs a 2-bit one.

**IV. IMPLEMENTATION FOR ASYMMETRIC 2-NAF**

For \(w = 2\), Algorithm 3 reduces to replacements \(R1\) through \(R6\). The intermediate signed-binary representation of \(k\) is scanned from left to right 3 bits at a time. If the consecutive 3 bits being scanned have the same form as one on the left-hand-side of \(R1\) through \(R4\), then they are replaced by the corresponding form on the right-hand-side. If a replacement is performed then the 3 bits that are involved in the replacement are skipped. If the rightmost two bits are not involved in previous replacements and they are either \(1\overline{1}\) or \(10\overline{5}\) then either \(R5\) or \(R6\) is applied.

- \(R1\): \(\overline{T}10 \rightarrow 0\overline{T}0\)
- \(R2\): \(1\overline{T}0 \rightarrow 010\)
- \(R3\): \(\overline{T}1\overline{T} \rightarrow 003\)
- \(R4\): \(0\overline{T}1 \rightarrow 003\)
- \(R5\): \(\overline{T}1 \rightarrow 0\overline{T}\)
- \(R6\): \(1\overline{T} \rightarrow 01\)

The code assignment for \{−3, −1, 0, 1\} used in our implementation is as follows: 01 for −3, 11 for −1, 00 for 0, and 01
for 1. The system includes two steps: Algorithm 1 (unsigned-binary to intermediate signed-binary) and the aforementioned 6 replacements (intermediate signed-binary to asymmetric 2-NAF). The complete circuit contains 7 flip-flops and 22 logic gates (multiple-input AND, NAND, OR, and NOR gates). The gate-level circuit diagram is not drawn in this paper due to space limitations.

The first conversion is realized by the finite state machine in Fig. 1 working from left to right, i.e., reading the most significant digits first. The label of the states corresponds to the last input digit read. Symbol $\perp$ denotes the end of the input sequence. The circuit for this conversion requires 1 flip-flop and 2 gates.

The second conversion can be realized by a transducer with 19 states. This is described by the truth table instead of a complex transducer for the sake of clarity. The state table after all possible eliminations and simplifications is illustrated in Table V. The column labelled ‘current state’ indicates the consecutive 3 signed-binary digits being scanned. The corresponding output is shown in the second column. $x$ denotes the next input bit, i.e., the digit to the right of the bits being scanned. In the implementation $-3, -1, 0, 1$ are respectively replaced by their corresponding 2-bit binary codes $01, 11, 00,$ and $01,$ respectively. This part of the circuit consists of 6 flip-flops and 20 gates.

As mentioned before, NAF results in an average weight of $0.333L$ where $L$ is the number of bits in unsigned-binary expansions. Our simulation resulting from 1 million randomly generated 571-bit (the size of field elements for the elliptic curves defined by NIST [13]) binary patterns shows that the average weight for asymmetric 2-NAF is $0.285L$. Thereafter the number of additions for computing $kP$ in the double-and-add algorithm is decreased by 14.4% with the cost of only one more precomputation (i.e., $-3P$). Also, the volume of output data bus remains 2 bits as in NAF or the representation in [6]. We remark that the symmetric 2-NAF with digit set $\{-3, -1, 0, 1, 3\}$ has an average weight of $0.250L$ [14], but requires a 3-bit output data bus.

V. CONCLUSION

We have proposed a new signed-binary recoding for the scalar multiplication $kP$ in ECCs, called asymmetric $w$-NAF. The asymmetric $w$-NAF uses digits in $\{-2w - 1, -2w - 3, \ldots, -1, 0, 1, \ldots, 2w - 3\}$ to denote integer $k$. The circuit for $w = 2$ consists of 7 flip-flops and 22 logic gates. Compared to NAF or the representation in [6], the number of additions is reduced by 14.4%.

![Fig. 1. Transducer realizing Algorithm 1 from left to right.](image)

### TABLE V

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$000$</td>
<td>$000$ $001$ $001$</td>
</tr>
<tr>
<td>$001$</td>
<td>$010$ $101$ $001$</td>
</tr>
<tr>
<td>$001^\perp$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$010$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$011$</td>
<td>$000$ $001$ $001$</td>
</tr>
<tr>
<td>$101$</td>
<td>$010$ $-1$ $001$</td>
</tr>
<tr>
<td>$101^\perp$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$100$</td>
<td>$000$ $001$ $001$</td>
</tr>
<tr>
<td>$101^\perp$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$110$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$111$</td>
<td>$070$ $000$ $001$</td>
</tr>
<tr>
<td>$111^\perp$</td>
<td>$070$ $000$ $001$</td>
</tr>
</tbody>
</table>

#### REFERENCES


