Optimal Left-to-right Binary Signed-Digit Recoding

[Published in *IEEE Transactions on Computers* **49**(7):740–748, 2000.]

Marc Joye[1] and Sung-Ming Yen[2]

[1] Gemplus Card International
Parc d'Activités de Gémenos, B.P. 100, 13881 Gémenos, France
marc.joye@gemplus.com
[2] Laboratory of Cryptography and Information Security (LCIS)
National Central University, Chung-Li, Taiwan 320, R.O.C.
yensm@csie.ncu.edu.tw

Abstract. This paper describes new methods for producing optimal binary signed-digit representations. This can be useful in the fast computation of exponentiations. Contrary to existing algorithms, the digits are scanned from left to right (i.e., from the most significant position to the least significant position). This may lead to better performances in both hardware and software.

Keywords. Computer arithmetic, converter, signed-digit representation, redundant number representation, SD2 left-to-right recoding, canonical/non-adjacent/minimum-weight form, exponentiation, elliptic curves, smart-cards, cryptography.

1 Introduction

Methodology using *signed-digit* (SD) representations, also called *redundant number* representations, for fast parallel arithmetic were considered in the late 1950's by Avizienis [1]. More recently, algorithms using signed-digit representations with the *digit set* $\{-1, 0, 1\}$ (in this paper, we call it the SD2 representation) accompanied with their applications to efficient methods for addition, multiplication, division, and their VLSI chip designs are presented in [2–6]. In general, after the computations in the SD2 domain, a *converter* is required to transform a number from its SD2 representation into the conventional binary representation. Such converters may be found in [7, Section 1.5], [8].

For many cryptosystems, (modular) *exponentiation* is one of the most time-consuming operations. Therefore, efficient algorithms to perform this operation are crucial in the performance of the resulting cryptographic protocols. Basically, when computing α^r, two types of exponentiations may be distinguished.

- The first type involves exponentiations with a fixed exponent r, such as in the RSA cryptosystem [9, 10]. The goal is then to quickly compute α^r for

randomly chosen α. This is usually achieved thanks to addition chains [11, Section 4.6.3]. The problem of finding the shortest addition chain was shown to be **NP**-hard by Downey, Leong and Sethi [12]; but good heuristics are known [13].
– In the second type of exponentiation, the base α is fixed and the exponent r varies. Examples include the ElGamal cryptosystem [14] and its numerous variations [15, Section 11.5]. In that case, good performances are obtained by the basic square-and-multiply technique (see Section 2). If larger amount of storage is available, this can be further improved via precomputations [16].

In this paper, we are mainly concerned with the second type of exponentiation. However, we note that our methods may lead to some advantages in the first type, too. Another application is when inverses can be virtually computed for free, as for elliptic curves [17]. The basic idea is to *recode* the exponent in a representation which has fewer nonzero digits, namely the SD2 representation. Already in 1951, this was successfully exploited by Booth to efficiently multiply two numbers [18]. An optimal version (in terms of the number of zero digits of the recoding) was later given by Reitwiesner [19]. In [20], Jedwab and Mitchell rediscovered Reitwiesner's algorithm and slightly generalized it by taking as input any SD2 representation of the exponent (instead of the binary representation). Our algorithms also produce optimal outputs but scan the digits of the exponent from left to right, i.e., from the most significant digit (MSD) to the least significant digit (LSD). This brings some advantages, especially in the hardware realization or for memory-constrained environments like smart-cards.

The rest of this paper is organized as follows. In Section 2, we review the square-and-multiply methods for fast exponentiation and extend them to exponents given in the SD2 representation. Section 3 presents Reitwiesner's algorithm. New exponent recoding algorithms are proposed in Section 4. Their hardware implementation is given in Section 5. Section 6 discusses further advantages of the proposed methods. Finally, we conclude in Section 7.

Notations

If $r = \sum_{i=0}^{m-1} r_i\, 2^i$ denotes the binary expansion of r, then we represent r as the vector $(r_{m-1}, \ldots, r_0)_2$. The bit-length of r is denoted by $|r|$. By abuse of notations, we do not make the distinction between the value of r and its representation and write $r = (r_{m-1}, \ldots, r_0)_2$. For signed-digit systems, we sometimes write $\bar{1}$ for -1. Moreover, if $r = \sum_{i=0}^{m} r'_i\, 2^i$ denotes the binary signed expansion of r (that is, $r'_i \in \{\bar{1}, 0, 1\}$), then we also abuse the notations and write $r = (r'_m, \ldots, r'_0)_{\text{SD2}}$.

Let S be a string. Then $\langle S \rangle^k$ means $S, S, \ldots, S$ (k times); for example, $(\langle 0, 1 \rangle^2, \bar{1})_{\text{SD2}}$ represents $(0, 1, 0, 1, \bar{1})_{\text{SD2}}$.

If t is a real number, then $\lfloor t \rfloor$ is the largest integer $\leq t$ and $\lceil t \rceil$ is the least integer $\geq t$.

Throughout this paper, the multiplicative notation is used. However, the described techniques also apply to additively written sets (e.g., additive group

of integers, points on an elliptic curve over a field). Exponentiation has then to be understood as multiplication.

2 Binary Algorithms for Fast Exponentiation

The most commonly used algorithms for computing α^r are the *binary methods* [11, Section 4.6.3]. The binary methods (also called *square-and-multiply methods*) scan the bits of exponent r either from right to left or from left to right (Fig. 1). At each step, a squaring is performed and depending on whether the scanned bit-value is equal to 1, a multiplication is also performed. Let $r = \sum_{i=0}^{m-1} r_i\, 2^i$ (with $r_{m-1} = 1$) be the binary expansion of r. The right-to-left (RL) algorithm is based on the observation that $\alpha^r = \left(\alpha^{2^0}\right)^{r_0} \left(\alpha^{2^1}\right)^{r_1} \cdots \left(\alpha^{2^{m-1}}\right)^{r_{m-1}}$, while the left-to-right (LR) algorithm follows from

$$\alpha^r = \left(\cdots \left(\left((\alpha^{r_{m-1}})^2\, \alpha^{r_{m-2}} \right)^2 \alpha^{r_{m-3}} \right)^2 \cdots \alpha^{r_1} \right)^2 \alpha^{r_0} \ .$$

```
INPUT: α, r = (r_{m-1}, ..., r_0)_2
OUTPUT: M = α^r
  M ← 1; S ← α
  for i from 0 to m − 1 do
    if (r_i = 1) then M ← M · S
    S ← S^2
  od
```

(a) Right-to-left (RL).

```
INPUT: α, r = (r_{m-1}, ..., r_0)_2
OUTPUT: M = α^r
  M ← 1
  for i from m − 1 down to 0 do
    M ← M^2
    if (r_i = 1) then M ← M · α
  od
```

(b) Left-to-right (LR).

Fig. 1. Square-and-multiply algorithms.

We remark that the LR algorithm (Fig. 1 (b)) requires 2 registers (for α and for M) and that the RL algorithm (Fig. 1 (a)) requires one more register (for S). However, we note that S can be used in place of α if the value of α is not needed thereafter. The RL algorithm presents the advantage to be parallelizable: one multiplier performs the multiplications $M \leftarrow M \cdot S$ and another one performs the squarings $S \leftarrow S^2$. However, if only one multiplier is available, the LR algorithm may be preferred because the multiplications are always done by the fixed value α, $M \leftarrow M \cdot \alpha$. So, if α has a special structure, these multiplications may be easier than multiplying two arbitrary numbers (see [15, Note 14.81] or [21, pp. 9–10] for examples of application).

Let $\omega(r)$ denote the *Hamming weight* of r (that is, the number of 1's in the binary representation of r). Both algorithms require $\omega(r) - 1$ multiplications and $m-1$ squarings (we do not count multiplications by 1, nor $1 \cdot 1$, nor the last squaring in Fig. 1 (a)) to compute α^r. It is well-known that $m-1$ is a lower bound for the number of squarings. However, the number of subsequent multiplications can

be further reduced by using a recoding algorithm [20, 22]. For example, to compute α^{15}, the LR algorithm will successively evaluate $\alpha, \alpha^2, \alpha^3, \alpha^6, \alpha^7, \alpha^{14}, \alpha^{15}$, that is, it performs 3 squarings and 3 multiplications. If the value of α^{-1} is supplied along with α (or if α^{-1} can cheaply be computed, as for elliptic curves [17]), then α^{15} is more quickly evaluated as $\alpha^{16} \cdot \alpha^{-1} = (((\alpha^2)^2)^2)^2 \cdot \alpha^{-1}$, which requires 4 squarings and 1 multiplication.

If we allow the digits of the exponent to be in $\{\bar{1}, 0, 1\}$, then the binary methods to compute α^r are easily modified as depicted in Fig. 2. Note that the SD2 representation of r may require an extra digit, r'_m, we refer to Section 3 for an explanation.

```
INPUT: α, r = (r'_m, ... , r'_0)_SD2
OUTPUT: M = α^r
  M ← 1; S ← α
  for i from 0 to m do
    if (r'_i = 1) then M ← M · S
    if (r'_i = 1̄) then M ← M · S^{-1}
    S ← S^2
  od
```

(a) Modified right-to-left (MRL).

```
INPUT: α, r = (r'_m, ... , r'_0)_SD2
OUTPUT: M = α^r
  M ← 1
  for i from m down to 0 do
    M ← M^2
    if (r'_i = 1) then M ← M · α
    if (r'_i = 1̄) then M ← M · α^{-1}
  od
```

(b) Modified left-to-right (MLR).

Fig. 2. Square-and-multiply-or-divide algorithms.

We see that the modified right-to-left (MRL) algorithm (Fig. 2 (a)) now works more differently and less efficiently. Indeed, when $r'_i = \bar{1}$, the inverse of S has to be computed (note that the value of S varies at each step). Most commonly used cryptosystems work in the multiplicative group of a finite field or ring. Inversion is then usually achieved via the extended Euclidean algorithm [15, Algorithm 2.142] or Fermat's Theorem [15, Fact 2.127] (see also [23] for a specialized implementation). This is a rather costly operation; therefore the benefits resulting from the reduced number of multiplications may be annihilated. The modified left-to-right (MLR) algorithm (Fig. 2 (b)) only needs the *fixed* value of α^{-1}, which can be precomputed. Consequently, in the sequel, we will only consider the MLR algorithm. We note, however, that the MRL algorithm may be useful when the inverse is available at no cost, as for elliptic curves or for the additive group of integers.

With the SD2 representation, the (minimal) Hamming weight $\omega(r)$ of r (i.e., the number of nonzero digits in the SD2 expansion of r) is equal to $(m+1)/3$, on average [24]. The computation of α^r can thus be performed with $\frac{4}{3}\,m + \mathrm{O}(1)$ multiplications plus squarings with the MLR algorithm, while the (standard) binary algorithms need $\frac{3}{2}\,m + \mathrm{O}(1)$ multiplications plus squarings (see Fig. 1), on average. Assuming that a squaring is approximatively as costly as a multiplication, then we can roughly expect a gain of $(\frac{3}{2} - \frac{4}{3})/\frac{3}{2} \approx 11.11\,\%$ over the (standard) binary methods.

The next section presents an algorithm to convert the exponent r from its binary representation into its SD2 representation. This algorithm, due to Reitwiesner, is optimal in the sense that it gives an SD2 output with *minimal* Hamming weight. Unfortunately, Reitwiesner's algorithm scans the bits of the exponent from *right to left*, while we have seen that only the modified *left-to-right* algorithm may bring some advantages. These *heterogeneous* modes of operation require to first recode the exponent r into its SD2 representation $(r'_m, \ldots, r'_0)_{\text{SD2}}$ and to temporarily store it for its latter usage in the MLR exponentiation algorithm. Noting that each digit in SD2 representation is encoded with 2 bits, *twice* the memory space taken by r is needed to store its SD2 representation. These shortcomings are alleviated in Section 4, where optimal left-to-right recoding algorithms are presented.

3 Reitwiesner's Method

In binary signed-digit notation[1] (i.e., using the digits $\{\bar{1}, 0, 1\}$), a number is not uniquely represented. Two representations $(a_\ell, a_{\ell-1}, \ldots, a_0)_{\text{SD2}}$ and $(b_\ell, b_{\ell-1}, \ldots, b_0)_{\text{SD2}}$ of a same number are *equivalent* if they have both the same length and the same Hamming weight; this equivalence will be denoted $(a_\ell, a_{\ell-1}, \ldots, a_0)_{\text{SD2}} \equiv (b_\ell, b_{\ell-1}, \ldots, b_0)_{\text{SD2}}$. A binary signed-digit representation is said to be *canonical* (or *sparse*) if no two adjacent digits are nonzero. For that reason, some authors sometimes call it the *nonadjacent form* (NAF) of a number [25].

The canonical recoding was studied by Reitwiesner [19]. He proved that this representation is unique (if the binary representation is viewed as padded with an initial 0). Following Hwang [7, pp. 150–151], Reitwiesner's method to convert a number $r = (r_{m-1}, \ldots, r_0)_2$ with $r_i \in \{0, 1\}$ into its canonical form $r = (r'_m, r'_{m-1}, \ldots, r'_0)_{\text{SD2}}$ with $r'_i \in \{\bar{1}, 0, 1\}$ is given by the algorithm depicted in Fig. 3. This is also known as Booth canonical recoding algorithm; however, Booth's method does not present the NAF property (see Footnote 2).

```
INPUT: (r_{m-1}, ..., r_0)_2
OUTPUT: (r'_m, r'_{m-1}, ..., r'_0)_SD2
  c_0 ← 0; r_{m+1} ← 0; r_m ← 0
  for i from 0 to m do
    c_{i+1} ← ⌊(c_i + r_i + r_{i+1})/2⌋
    r'_i ← c_i + r_i − 2c_{i+1}
  od
```

Fig. 3. Reitwiesner's canonical recoding algorithm.

Reitwiesner's algorithm is very efficient. It can be done by using the following look-up table ('X' stands for 0 or 1, that is, the output is independent of this value).

[1] Also called ternary balanced notation [11, p. 190].

Table 1. Right-to-left SD exponent recoding.

c_i	r_i	r_{i+1}	c_{i+1}	r'_i
0	0	X	0	0
0	1	0	0	1
0	1	1	1	$\bar{1}$
1	0	0	0	1
1	0	1	1	$\bar{1}$
1	1	X	1	0

At first glance, it is not so obvious that this algorithm effectively yields the canonical representation of a number. However, if we closely observe how it works, we see that this algorithm comes down to subtract r from $3r$ (with the additional rule $0 - 1 = \bar{1}$) and then to discard the last (i.e., least significant) 0 [26].

$$\begin{array}{rl}
2r = & (r_{m-1},\ r_{m-2}\,,\ r_{m-3},\ \ldots,\ r_1,\ r_0,\ 0\,)_2 \\
+\,r = & (r_{m-1},\ r_{m-2},\ \ldots,\ r_2,\ r_1,\ r_0)_2 \\
\hline
3r = & (s_m,\ s_{m-1}\,,\ s_{m-2}\,,\ s_{m-3},\ \ldots,\ s_1,\ s_0,\ r_0)_2 \\
-\,r = & (r_{m-1},\ r_{m-2},\ \ldots,\ r_2,\ r_1,\ r_0)_2 \\
\hline
2r = & (r'_m,\ r'_{m-1}\,,\ r'_{m-2}\,,\ r'_{m-3},\ \ldots,\ r'_1,\ r'_0,\ 0\,)_{\text{SD2}}
\end{array}$$

Fig. 4. A simple explanation of Reitwiesner's method.

Indeed, if $r_0 + \sum_{i=0}^{m} s_i\, 2^{i+1}$ denotes the binary expansion of $3r$, then the conventional pencil-and-paper method to add nonnegative integers [11, p. 251] gives $s_i = (c_i + r_i + r_{i+1}) \bmod 2 = c_i + r_i + r_{i+1} - 2\lfloor (c_i + r_i + r_{i+1})/2 \rfloor$ where c_i is the carry-in. Moreover, since the carry-out c_{i+1} is equal to 1 if and only if there are two or three 1's among c_i, r_i and r_{i+1}, we can write $c_{i+1} = \lfloor (c_i + r_i + r_{i+1})/2 \rfloor$. Hence, $s_i = c_i + r_i + r_{i+1} - 2c_{i+1}$ and thus $r'_i = s_i - r_{i+1} = c_i + r_i - 2c_{i+1}$.

To see that the output is sparse, it suffices to remark that the algorithm scans the bits from right to left and replaces a consecutive block of several 1's by a block of 0's and $\bar{1}$ according to $(\langle 1 \rangle^a)_2 \mapsto (1, \langle 0 \rangle^{a-1}, \bar{1})_{\text{SD2}}$.[2] Furthermore, if two blocks of 1's are separated by an isolated 0, the algorithm implicitly uses the fact that $(\bar{1}, 1)_{\text{SD2}} \equiv (0, \bar{1})_{\text{SD2}}$. For example, $(\langle 1 \rangle^a, 0, \langle 1 \rangle^b)_2$ is replaced by $(1, \langle 0 \rangle^a, \bar{1}, \langle 0 \rangle^{b-1}, \bar{1})_{\text{SD2}}$—and not by $(1, \langle 0 \rangle^{a-1}, \bar{1}, 1, \langle 0 \rangle^{b-1}, \bar{1})_{\text{SD2}}$. A more formal (but less intuitive) proof of the sparseness property is given in the next lemma.

Lemma 1. *$r'_i \cdot r'_{i+1} = 0$ for all $0 \le i \le m - 1$.*

[2] This is the transformation initially proposed by Booth [18].

Proof. Suppose $r'_i \neq 0$. Since $r'_i = c_i + r_i - 2c_{i+1}$, we must have $c_i + r_i = 1$, whence $c_{i+1} = \lfloor (1 + r_{i+1})/2 \rfloor = r_{i+1}$ and thus $r'_{i+1} = 2(r_{i+1} - c_{i+2}) = 0$. □

The main advantage of Reitwiesner's algorithm is that, in some sense, it is optimal. Indeed, Reitwiesner [19] proved that:

Proposition 1. *Among the* SD2 *representations, the canonical representation has minimal Hamming weight.* □

Although, this does not rule out the existence of other minimal representations. For example, $(1, 0, 1, 1)_{\text{SD2}}$ and $(1, 0, \bar{1}, 0, \bar{1})_{\text{SD2}}$ are both minimal representations for 11.

The general case was later addressed by Clark and Liang [26]. They present a minimal representation for any signed-radix b. In that case, Arno and Wheeler [24] proved that the average proportion of nonzero digits is equal to $(b-1)/(b+1)$. This has to be compared with the average proportion $(b-1)/b$ of nonzero digits in the standard radix b representation. So, we see that exponent recoding is mostly interesting for binary signed-digit representation ($b = 2$) because the savings rapidly go down.

4 Proposed Methods

In Section 2, we pointed out that a left-to-right recoding algorithm might be desirable for fast exponentiation. Designing such an algorithm is not as straightforward as it appears and is even considered as a hard problem by some authors [27].

Our first algorithm is an adaptation of Reitwiesner's algorithm. As in the right-to-left algorithm, it also presents the NAF property. Unfortunately, look-up tables cannot be used. Our second algorithm does not have the NAF property but is *equally* efficient as Reitwiesner's algorithm. Moreover, it enables the use of a look-up table.

4.1 A Simple Left-to-right Recoding Algorithm

The interpretation of the NAF given in the previous section suggests a simple way to construct a left-to-right recoding algorithm. Let $r = (r_{m-1}, \ldots, r_0)_2$ and $3r = (s_m, \ldots, s_0, r_0)_2$, then the NAF for r is $(r'_m, r'_{m-1}, \ldots, r'_0)_{\text{SD2}}$ where $r'_i = s_i - r_{i+1}$. So, if we have at our disposal an algorithm to add $r = (r_{m-1}, \ldots, r_0)_2$ and $2r = (r_{m-1}, \ldots, r_0, 0)_2$ from *left to right*, then we can compute $3r$, subtract r (in SD2 representation), discard the last 0 and obtain the canonical representation of r. Fortunately, such algorithms exist. A left-to-right addition algorithm is presented in Fig. 5 (see [11, Exercise 4.3.1.6]).

Taking as input $u = (0, r_{m-1}, \ldots, r_1)_2$ and $v = (r_{m-1}, r_{m-2}, \ldots, r_0)_2$, we get $w = (s_m, \ldots, s_0)_2$; we have now to subtract $(r_{m-1}, \ldots, r_1)_2$ (with the rule $0 - 1 = \bar{1}$). However, some technical difficulties occur: when we enter in the `while loop` (Lines 5–7 and 12–14 in Fig. 5), the algorithm may output *several*

```
INPUT: u = (u_{m-1}, ..., u_0)_2 and v = (v_{m-1}, ..., v_0)_2
OUTPUT: u + v = (w_m, w_{m-1}, ..., w_0)_2
  j ← m
  for i from m − 1 down to 0 do
    if (u_i = v_i) then
      w_j ← v_i
      while (j > i + 1) do
        j ← j − 1; w_j ← 1 − v_i
      od
      j ← j − 1
    fi
  od
  w_j ← 0
  while (j > 0) do
    j ← j − 1; w_j ← 1
  od
```

Fig. 5. Left-to-right addition algorithm.

s_j's (with $j > i$) wherefrom, for each s_j, we have to subtract r_{j+1} in order to obtain the corresponding r'_j. This means that the r_{j+1}'s with $j > i$ (i.e., $r_{i+2}, r_{i+3}, \dots$) must be available. On the other hand, we remark that at step $i = I$, variable j is only decremented when $r_{I+1} = r_I$. So, if J denotes the value of j before entering in the `while loop`, we see that this loop is only executed after a block $(r_{J+1}, r_J, \dots, r_{I+1}, r_I)$ either of the form

$$\text{(B1)} \qquad (0, \langle 0,1\rangle^{(J-i)/2}, 1) \quad \text{or} \quad (0, \langle 0,1\rangle^{(J-i-1)/2}, 0, 0) \ ,$$

or of the form

$$\text{(B2)} \qquad (1, \langle 1,0\rangle^{(J-i)/2}, 0) \quad \text{or} \quad (1, \langle 1,0\rangle^{(J-i-1)/2}, 1, 1) \ .$$

We therefore introduce an additional variable b to distinguish between the two cases. Because of the alternation of 0 and 1 inside a block, we do not have to know the value of the r_{j+1}'s inside the `while loop`; only the value of the two first consecutive equal bits (i.e., r_{J+1} and r_J) is necessary: we use the variable b to keep track of this value, that is, before entering the loop, b contains the value of r_J ($b = 0$ in Case (B1) and $b = 1$ in Case (B2)). Next, inside the `while loop`, we alternately subtract 0 or 1, starting with 0 or 1 depending on the value of b. Putting all together, we finally our first algorithm (Fig. 6).

While this algorithm yields the canonical representation, it is not fully satisfying. It looks quite cumbersome compared to the original Reitwiesner's algorithm (see Fig. 3). The next paragraph considers another minimal representation which leads to a very elegant left-to-right recoding algorithm.

```
INPUT: (r_{m-1}, ..., r_0)_2
OUTPUT: (r'_m, r'_{m-1}, ..., r'_0)_SD2
  j ← m; b ← 0; r_m ← 0
  for i from m − 1 down to 0 do
    if (r_{i+1} = r_i) then
      r'_j ← r_i − b
      while (j > i + 1) do
        j ← j − 1; r'_j ← 1 − r_i − b
        b ← 1 − b
      od
      b ← r_i; j ← j − 1
    fi
  od
  r'_j ← −b
  while (j > 0) do
    j ← j − 1; r'_j ← 1 − b
    b ← 1 − b
  od
```

Fig. 6. Canonical left-to-right recoding algorithm.

4.2 Minimum-weight Left-to-right Recoding Algorithm

The main difficulty in the previous algorithm comes from the fact that it can only "decide" what the output will be after two consecutive equal bits. In other words, when entering in an alternate block of $\langle 0,1\rangle^k$ (or $\langle 1,0\rangle^k$), the algorithm must know *a priori* if this block will end with two consecutive bits equal to 0 or 1. The next lemma enables to give a *minimal* (albeit not sparse) output whatever the ending bits of an alternate block. The `while loop`'s in the canonical algorithm (Fig. 6) can therefore be removed.

Lemma 2. *In* SD2 *representation, we have the following equivalences:*

(a) $(\langle 0,1\rangle^k, 1)_{\mathrm{SD2}} \equiv (1, \langle 0,\bar{1}\rangle^k)_{\mathrm{SD2}}$*;*
(b) $(\langle 0,\bar{1}\rangle^k, \bar{1})_{\mathrm{SD2}} \equiv (\bar{1}, \langle 0,1\rangle^k)_{\mathrm{SD2}}$*.*

Proof. Let $S_k = \sum_{i=1}^{k} 2^{2(i-1)} = (2^{2k}-1)/3$. In signed-digit representation, $(\langle 0,1\rangle^k, 1)_{\mathrm{SD2}}$ represents the number $N = 1 + \sum_{i=1}^{k} 2^{2(i-1)+1}$. However since $N = 1 + 2S_k = (3S_k + 1) - S_k = 2^{2k} - \sum_{i=1}^{k} 2^{2(i-1)}$, it may also be represented as $(1, \langle 0,\bar{1}\rangle^k)_{\mathrm{SD2}}$. Noting that $(\langle 0,\bar{1}\rangle^k, \bar{1})_{\mathrm{SD2}}$ and $(\bar{1}, \langle 0,1\rangle^k)_{\mathrm{SD2}}$ are both representations of $-N$, the second equivalence follows from the first one. □

Elementary Blocks [This paragraph explains how the recoding algorithm was found. The reader uniquely interested by the algorithm itself may skip it.]

The exponent r to be recoded may be viewed as a succession of *elementary blocks* $(r_{J+1}, r_J, \ldots, r_{I+1}, r_I)$ with $J > I$, $r_{J+1} = r_J$ and $r_{I+1} = r_I$, and whose form is given by one of the four following possibilities.

(E1) $(0, \langle 0,1\rangle^{(J-I)/2}, 1)$,
(E2) $(0, \langle 0,1\rangle^{(J-I-1)/2}, 0, 0)$,
(E3) $(1, \langle 1,0\rangle^{(J-I)/2}, 0)$,
(E4) $(1, \langle 1,0\rangle^{(J-I-1)/2}, 1, 1)$.

For each of these forms, our left-to-right canonical recoding algorithm (Fig. 6) will respectively output the corresponding block $(r'_J, \ldots, r'_{I+1})$ given by

(E1') $(1, \langle 0,\bar{1}\rangle^{(J-I-2)/2}, 0)$,
(E2') $(0, \langle 1,0\rangle^{(J-I-1)/2})$,
(E3') $(\bar{1}, \langle 0,1\rangle^{(J-I-2)/2}, 0)$,
(E4') $(0, \langle \bar{1},0\rangle^{(J-I-1)/2})$.

Now, using Lemma 2, we may respectively replace these outputs by

(E1*) $(\langle 0,1\rangle^{(J-I-2)/2}, 1, 0)$,
(E2*) $(\langle 0,1\rangle^{(J-I-1)/2}, 0)$, (= E2')
(E3*) $(\langle 0,\bar{1}\rangle^{(J-I-2)/2}, \bar{1}, 0)$,
(E4*) $(\langle 0,\bar{1}\rangle^{(J-I-1)/2}, 0)$. (= E4')

Example 1. Suppose that $r = (1,1,1,0,1,0,0,1)_2$. By adding artificial beginning and ending 0's, we decompose r into 4 elementary blocks. We then apply our transformation to each elementary block to finally obtain $r = (1,0,0,0,\bar{1},\bar{1},0,0,1)_{\text{SD2}}$.

$$
\begin{array}{rl}
 & r = (\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ .\ 0\ 0\ )_2 \\
\text{(E1)} & \quad 0\ 0\ 1\ 1 \\
\text{(E4)} & \quad\ \ 1\ 1\ 1 \\
\text{(E3)} & \quad\ \ \ 1\ 1\ 0\ 1\ 0\ 0 \\
\text{(E2)} & \quad\ \ \ \ \ \ \ \ 0\ 0\ 1\ .\ 0\ 0 \\
\text{(E1*)} & \quad\ 1\ 0 \\
\text{(E4*)} & \quad\ \ \ 0 \\
\text{(E3*)} & \quad\ \ \ \ 0\ \bar{1}\ \bar{1}\ 0 \\
\text{(E2*)} & \quad\ \ \ \ \ \ \ \ \ \ 0\ 1\ .\ 0 \\
 & \rightarrow r = (\quad 1\ 0\ 0\ 0\ \bar{1}\ \bar{1}\ 0\ 0\ 1\ .\ 0\quad)_{\text{SD2}}
\end{array}
$$

Although not sparse, the resulting representation has the same Hamming weight as the canonical representation $(1,0,0,\bar{1},0,1,0,0,1)_{\text{SD2}}$. Both representations being equivalent, $(1,0,0,0,\bar{1},\bar{1},0,0,1)_{\text{SD2}}$ is thus a *minimal* representation for r. $\Diamond$

Consider an elementary block $(r_{J+1}, r_J, \ldots, r_{I+1}, r_I)$ and the corresponding signed-digit block $(r'_J, \ldots, r'_{I+1})$. Let b_i denote the value of variable b when bits r_i and r_{i-1} are scanned, namely $b_i = 0$ or 1 depending on whether r_i and r_{i-1} belong to an elementary block beginning with two zeros or two 1's. Suppose that $r_{J+1} = r_J = 0$. Then we have $b_i = 0$ for all $J+1 \geq i \geq I+2$, and $b_{I+1} = 0$ if $r_{I+1} = r_I = 0$ or $b_{I+1} = 1$ if $r_{I+1} = r_I = 1$. Similarly, if $r_{J+1} = r_J = 1$, then

$b_i = 1$ for all $J+1 \geq i \geq I+2$, and $b_{I+1} = 0$ if $r_{I+1} = r_I = 0$ or $b_{I+1} = 1$ if $r_{I+1} = r_I = 1$. Because of the alternation of 0 and 1 inside an elementary block, we have $r_i + r_{i-1} = 1$ for all $J \geq i \geq I+2$. Hence, if we set

$$b_i = \lfloor (b_{i+1} + r_i + r_{i-1})/2 \rfloor \ , \tag{1}$$

we have $b_i = \lfloor (b_{i+1} + 1)/2 \rfloor = b_{i+1}$ for all $J \geq i \geq I+2$ and thus by induction $b_i = b_{J+1}$ for all $J \geq i \geq I+2$ as required. Moreover, since $r_{I+1} = r_I$, we also have $b_{I+1} = \lfloor (b_{I+2} + 2r_{I+1} + r_I)/2 \rfloor = r_{I+1}$ as required. Finally, we remark that if $b_i = b_{i-1} = 0$ then $r'_i = r_i$, and that if $b_i = b_{i-1} = 1$ then $r'_i = r_i - 1$. Consequently, when $b_i = b_{i-1}$, we can write $r'_i = r_i - b_i$. If $b_i \neq b_{i-1}$, then (1) if $b_i = 1$ and $b_{i-1} = 0$ then $r'_i = \bar{1}$; (2) if $b_i = 0$ and $b_{i-1} = 1$ then $r'_i = 1$. So, when $b_i \neq b_{i-1}$, we can write $r'_i = b_{i-1} - b_i$. Noting that if $b_i \neq b_{i-1}$ then $b_i = r_i$, we can see that

$$r'_i = (r_i - b_i) + (b_{i-1} - b_i) = r_i - 2b_i + b_{i-1} \tag{2}$$

is a valid expression for r'_i.

Our Second Algorithm From the simple formulations for b_i and r'_i (see Eqs. (1) and (2)), we obtain an elegant and efficient left-to-right exponent recoding algorithm.

```
INPUT: (r_{m-1}, ..., r_0)_2
OUTPUT: (r'_m, r'_{m-1}, ..., r'_0)_SD2
  b_m ← 0; r_m ← 0; r_{-1} ← 0; r_{-2} ← 0
  for i from m down to 0 do
    b_{i-1} ← ⌊(b_i + r_{i-1} + r_{i-2})/2⌋
    r'_i ← −2b_i + r_i + b_{i-1}
  od
```

Fig. 7. Minimum-weight left-to-right recoding algorithm.

Similarly as Reitwiesner's algorithm, our second algorithm can be performed still more efficiently thanks to table look-up. The corresponding look-up table is given hereafter (as aforementioned 'X' stands for 0 or 1).

Note that entries $(b_i, r_i, r_{i-1}, r_{i-2}) = (0, 1, 1, \mathrm{X})$ and $(1, 0, 0, \mathrm{X})$ are missing; see the discussion before Lemma 3 (next paragraph) for an explanation.

Main Theorem From its construction, the proposed algorithm (Fig. 7) produces an SD2 representation with minimal Hamming weight. However, for completeness, we now give a formal proof of this assertion.

We first consider three auxiliary lemmas. Lemma 3 implies that the cases $(b_i, r_i, r_{i-1}, r_{i-2}) = (0, 1, 1, \mathrm{X})$ and $(b_i, r_i, r_{i-1}, r_{i-2}) = (1, 0, 0, \mathrm{X})$ never occur (see Table 2). Lemma 4 shows that if $b_i = r_i$ then the output is sparse.

Table 2. Left-to-right SD exponent recoding.

b_i	r_i	r_{i-1}	r_{i-2}	b_{i-1}	r'_i
0	0	0	X	0	0
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	X	0	1
1	0	1	X	1	$\bar{1}$
1	1	0	0	0	$\bar{1}$
1	1	0	1	1	0
1	1	1	X	1	0

Finally, Lemma 5 proves that our algorithm effectively yields an SD2 representation.

Lemma 3. *If $r_i = r_{i-1}$ then $b_i = r_i$.*

Proof. Straightforward because $r_i = r_{i-1}$ yields $b_i = \lfloor (b_{i+1} + r_i + r_{i-1})/2 \rfloor = \lfloor b_{i+1}/2 + r_i \rfloor = r_i$. □

Lemma 4. *If $b_i = r_i$ then $r'_i \cdot r'_{i-1} = 0$.*

Proof. Suppose first that $r_{i-1} + r_{i-2} = 2(1 - b_i)$. Then $r_{i-1} = r_{i-2} = 1 - b_i$. So, by Lemma 3, $b_{i-1} = 1 - b_i$ and thus $b_{i-2} = \lfloor (b_{i-1} + r_{i-2} + r_{i-3})/2 \rfloor = \lfloor (1 - b_i) + r_{i-3}/2 \rfloor = 1 - b_i$. Hence, $r'_{i-1} = -2b_{i-1} + r_{i-1} + b_{i-2} = 0$ and $r'_i \cdot r'_{i-1} = 0$. If $r_{i-1} + r_{i-2} \neq 2(1 - b_i)$, then $r_{i-1} + r_{i-2} \leq 1$ when $b_i = 0$ and $r_{i-1} + r_{i-2} \geq 1$ when $b_i = 1$. So, $b_{i-1} = \lfloor (b_i + r_{i-1} + r_{i-2})/2 \rfloor = 0$ when $b_i = 0$ and $b_{i-1} = 1$ when $b_i = 1$; or equivalently, $b_{i-1} = b_i$. Therefore, since $b_i = r_i$, we have $r'_i = -2b_i + r_i + b_{i-1} = 0$ and $r'_i \cdot r'_{i-1} = 0$. □

Lemma 5. *Let $r = (r_{m-1}, \ldots, r_0)_2$ be the binary representation of r. Then the output $(r'_m, \ldots, r'_0)_{\text{SD2}}$ given by the minimum-weight left-to-right recoding algorithm is an SD2 representation of r.*

Proof. From the look-up table (which enumerates all the possible cases), we have $r'_i \in \{\bar{1}, 0, 1\}$. Hence, it remains to prove that $\sum_{i=0}^{m} r'_i \, 2^i = r$. The algorithm gives

$$\begin{aligned}\sum_{i=0}^{m} r'_i \, 2^i &= \sum_{i=0}^{m} (-2b_i + r_i + b_{i-1}) \, 2^i \\ &= \sum_{i=0}^{m} r_i \, 2^i + \sum_{i=0}^{m} (b_{i-1} \, 2^i - b_i \, 2^{i+1}) \\ &= r_m \, 2^m + r + (b_{-1} - b_m \, 2^{m+1}) = r\end{aligned}$$

since $r_m = b_m = 0$ (by definition) and $b_{-1} = 0$ (because $r_{-1} = r_{-2} = 0$). □

Theorem 1. *As Reitwiesner's algorithm, our left-to-right recoding algorithm produces an* SD2 *representation with minimal Hamming weight.*

Proof. Proposition 1 says that a sparse method gives a minimal output. So, from Lemmas 3 and 4, it remains to prove that the following inputs lead to optimal outputs.

1. $\boxed{(b_i, r_i, r_{i-1}, r_{i-2}) = (0,1,0,0)}$

   Hence, $(b_{i-1}, r_{i-1}, r_{i-2}, r_{i-3}) = (0,0,0,r_{i-3})$. So, again from Table 2, we have $r'_{i-1} = 0$ and thus $r'_i \cdot r'_{i-1} = 0$.

2. $\boxed{(b_i, r_i, r_{i-1}, r_{i-2}) = (0,1,0,1)}$

   Then we have $(b_{i-1}, r_{i-1}, r_{i-2}, r_{i-3}) = (0,0,1,r_{i-3})$ and $r'_i = 1$.
   (a) If $r_{i-3} = 0$ then, from Table 2, $r'_{i-1} = 0$ and thus $r'_i \cdot r'_{i-1} = 0$.
   (b) Otherwise if $r_{i-3} = 1$ then $b_{i-2} = 1$ and $r'_{i-1} = 1$. Furthermore, since $b_{i-1} = r_{i-1}$, it follows from Lemma 4 that $r'_{i-2} = 0$. We now examine the outputs on the left. Let $k \geq 1$ be the least integer such that $r_{i+2k} = 0$. Hence $r_{i+2j} = 1$ for all $0 \leq j \leq k-1$. Define $\vec{Y}_j = (b_{i+2j}, r_{i+2j}, r_{i+2j-1})$. We can prove by induction that

   $$\vec{Y}_j = (b_{i+2j}, r_{i+2j}, r_{i+2j-1}) = (0,1,0) \qquad (*)$$

   for all $0 \leq j \leq k-1$. If $j = 0$ then $\vec{Y}_0 = (0,1,0)$. Suppose now that $\vec{Y}_n = (b_{i+2n}, r_{i+2n}, r_{i+2n-1}) = (0,1,0)$ for some $0 \leq n \leq k-2$. We must then have $b_{i+2n+1} = 0$ because $b_{i+2n+1} = 1$ implies $b_{i+2n} = 1$, a contradiction. This, in turn, implies $r_{i+2n+1} = 0$ (otherwise, by Lemma 3, we would get $b_{i+2n+1} = r_{i+2n+1} = 1$). We also have $r_{i+2n+2} = 1$ because $r_{i+2j} = 1$ for all $0 \leq j \leq k-1$. Finally, since $b_{i+2n+1} = 0$, we must have $b_{i+2n+2} = 0$. Consequently, $\vec{Y}_{n+1} = (b_{i+2n+2}, r_{i+2n+2}, r_{i+2n+1}) = (0,1,0)$ and Eq. (*) is proven.
   Moreover, from Eq. (*), we have $b_{i+2j-1} = 0$ for all $0 \leq j \leq k-1$. Similarly, from $\vec{Y}_{k-1}$, we can prove that $b_{i+2k} = b_{i+2k-1} = 0$ and $r_{i+2k-1} = 0$. In short, when $r_{i-3} = 1$, we have:

r_{i+3}	r_{i+2}	r_{i+1}	r_i	r_{i-1}	r_{i-2}	r'_{i+3}	r'_{i+2}	r'_{i+1}	r'_i	r'_{i-1}	r'_{i-2}
·	·	0	1	0	1	·	·	0	1	1	0

   Since $(r_{i+2k}, \ldots, r_{i-1}, r_{i-2}) = (0, \langle 0,1 \rangle^{k+1})$ and $b_{i+j} = 0$ for all $0 \leq j \leq 2k$, we obtain $(r'_{i+2k}, \ldots, r'_i) = (0, \langle 0,1 \rangle^k)$. Moreover, $r'_{i-1} = 1$ and $r'_{i-2} = 0$. Therefore, the output is $(r'_{i+2k}, \ldots, r'_{i-1}, r'_{i-2}) = (0, \langle 0,1 \rangle^k, 1, 0) \equiv_{\text{SD2}} (0,1, \langle 0, \bar{1} \rangle^k, 0)$ by Lemma 2. Since the latter is sparse (note the beginning 0 and the ending 0), the output $(r'_{i+2k}, \ldots, r'_{i-1}, r'_{i-2})$ is optimal by Proposition 1.

3. $\boxed{(b_i, r_i, r_{i-1}, r_{i-2}) = (1,0,1,0)}$

   In this case, we have $(b_{i-1}, r_{i-1}, r_{i-2}, r_{i-3}) = (1,1,0,r_{i-3})$ and $r'_i = \bar{1}$.

(a) If $r_{i-3} = 1$ then $r'_{i-1} = 0$ and thus $r'_i \cdot r'_{i-1} = 0$.
(b) If $r_{i-3} = 0$, analogously to Case (2.b), we obtain:

r_{i+3}	r_{i+2}	r_{i+1}	r_i	r_{i-1}	r_{i-2}	r'_{i+3}	r'_{i+2}	r'_{i+1}	r'_i	r'_{i-1}	r'_{i-2}
·	·	1	0	1	0	·	·	0	$\bar{1}$	$\bar{1}$	0

Using similar arguments to those used in Case (2.b), we can prove that the output must be of the form $(r'_{i+2k}, \ldots, r'_{i-1}, r'_{i-2}) = (0, \langle 0, \bar{1}\rangle^k, \bar{1}, 0)$ for some $k \geq 1$. From Lemma 2, we see that this output is optimal since equivalent to the sparse representation $(0, \bar{1}, \langle 0, 1\rangle^k, 0)$.

4. $\boxed{(b_i, r_i, r_{i-1}, r_{i-2}) = (1, 0, 1, 1)}$

Similarly to Case (1.), $(b_{i-1}, r_{i-1}, r_{i-2}, r_{i-3}) = (1, 1, 1, r_{i-3})$; so $r'_{i-1} = 0$ and thus $r'_i \cdot r'_{i-1} = 0$.

This concludes the proof. □

5 Hardware Implementation

Evidently, the proposed minimum-weight signed-digit converter (Fig. 7) is much more regular and simpler for hardware implementation. To implement the transformation algorithm, we encode r'_i ($r'_i \in \{0, 1, \bar{1}\}$) as

$$\{0 \triangleq (\mathrm{X}, 0)_2;\ 1 \triangleq (0, 1)_2;\ -1 \triangleq (1, 1)_2\},$$

and we denote r'_i by a two-bit representation $(r'_{i,H}, r'_{i,L})_2$.

After some logic manipulations and minimizations, the following Boolean equations for b_{i-1}, $r'_{i,H}$ and $r'_{i,L}$ can be obtained. The outputs (b_{i-1}, r'_i) corresponding to the missing entries $(b_i, r_i, r_{i-1}, r_{i-2}) = (0, 1, 1, \mathrm{X})$ and $(1, 0, 0, \mathrm{X})$ of Table 2 were optimally assigned to $(0, 1)$ and $(1, \bar{1})$, respectively.

$$\begin{cases} b_{i-1} = \overline{b_i} \cdot r_{i-1} \cdot r_{i-2} + b_i \cdot r_{i-1} + b_i \cdot r_{i-2} \\ r'_{i,H} = b_i \\ r'_{i,L} = \overline{b_i} \cdot r_{i-1} \cdot r_{i-2} + \overline{b_i} \cdot r_i + b_i \cdot \overline{r_i} + b_i \cdot \overline{r_{i-1}} \cdot \overline{r_{i-2}} \end{cases}.$$

A hardware realization of these equations is given in Figure 8. Initially, the *shift-left registers* $\{r_i, r_{i-1}, r_{i-2}\}$ are loaded with $\{0, r_{m-1}, r_{m-2}\}$ and the *latch* is reset to logic "0". At the end of each iteration, the outputs $r'_{i,H}$ and $r'_{i,L}$ are used to encode the transformed SD2 digit r'_i. At this moment, the output b_{i-1} is fed into the latch, the lower bit r_{i-3} is prepared to be fed into the shift-left registers, and the registers shift one bit to the left.

Based on this minimum-weight signed-digit transformer, an hardware architecture for fast exponentiation, α^r, is sketched in Figure 9. In that exponentiation hardware, the outputs of the signed-digit transformer, $r'_{i,H}$ and $r'_{i,L}$, are used to control the computational data flow and the register fetching. Signal $r'_{i,H}$ is used

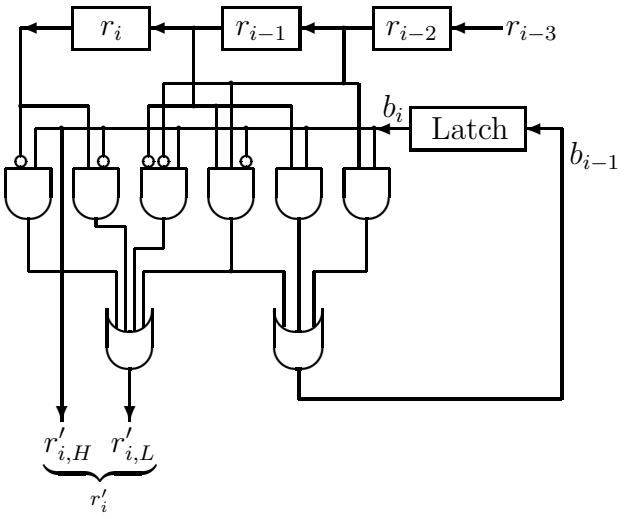

Fig. 8. Hardware implementation of the proposed transformer.

to determine which of α or α^{-1} has to be selected by the *multiplexer* **MUX-1**, and signal $r'_{i,L}$ determines whether or not the multiplication $A \leftarrow A \times B$ (where A means the *accumulator* and B means the output of **MUX-1**) needs to be performed. In the case of $r'_{i,L} = 1$ (and thus $M' =$ "$0\|1$", where $\|$ means concatenation), both the squaring and the multiplication are required. On the other hand, when $r'_{i.L} = 0$ (and thus $M' = 0$), only the squaring operation is required. Using this architecture, the computation of α^r needs, on average, $\frac{4}{3}|r|$ multiplications (if we assume that multiplication and squaring are equally time-consuming operations). For example, α^r is evaluated after only 684 multiplications for a 512-bit integer r.

6 Other Considerations

For some special cases, the proposed algorithms may bring further advantages.

1. One of the main concerns of the proposed algorithms was to reduce the Hamming weight of exponent r for the computation of α^r. While the number of squarings still remains the same (i.e., $|r|$), the average number of multiplications was reduced from $|r|/2$ to $|r|/3$. Over GF(2^k), squaring operations can be achieved via a simple circular shift if the elements are represented in the so-called *normal basis* [28]. Their computational costs can therefore be ignored comparing to multiplications. In this case, the real expected gain over the binary methods is then $(\frac{1}{2} - \frac{1}{3})/\frac{1}{2} \approx 66.66\,\%$ instead of 11.11% as aforementioned in Section 2.
2. Because the expression of variable b, $b_{i-1} \leftarrow \lfloor (b_i + r_{i-1} + r_{i-2})/2 \rfloor$ (see Fig. 7), is similar to that of carry c in Reitwiesner's algorithm (Fig. 3), $c_{i+1} \leftarrow$

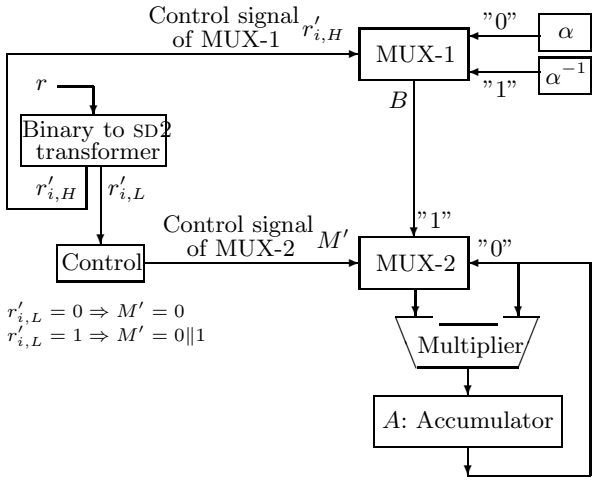

Fig. 9. Hardware architecture for the computation of α^r.

$\lfloor (c_i + r_i + r_{i+1})/2 \rfloor$, the *parallel* recoding method proposed by Koç [29] readily extends to our minimum-weight left-to-right recoding algorithm.

7 Conclusions

To improve the performance of the square-and-multiply exponentiation for the evaluation of α^r, the Hamming weight of exponent r should be reduced. This can be achieved by adopting a signed-digit representation for exponent r. Assuming that α^{-1} is provided along with α (or can be cheaply evaluated, as for elliptic curves), a minimum-weight signed-digit representation for r can improve quite a large the computation of α^r. However, in order to produce the minimum-weight signed-digit representation, existing solutions must initiate the recoding process from *right to left* while only the modified *left-to-right* (MLR) exponentiation algorithm is suitable. The *heterogeneous* processings bring some time and memory inefficiencies for hardware realization. Indeed, the recoding has first to be performed, the resulting representation for r must be saved (which requires *twice* more memory space than its binary representation). Then and only then, the exponentiation process may start in order to obtain α^r.

In this paper, new and *homogeneous* approaches for *minimum-weight* signed-digit representation are presented. Our methods are homogeneous in the sense that both the proposed recoding algorithms and the MLR exponentation algorithm initiate from the most significant position of the exponent. The signed-digit representation of exponent r is consequently obtained in real-time and does not need to be stored. This better memory usage is especially useful for small devices like smart-cards. Our first algorithm gives the canonical representation for the

exponent. This algorithm is then modified into another minimum-weight recoding algorithm so that table look-up is possible. A hardware implementation of this second converter and the corresponding architecture for exponentiation are also presented.

Acknowledgments

We are grateful to Dan Gordon for sending a preprint of [16]. Many thanks also go to the anonymous reviewers for their excellent work.

This work was partly supported by the National Science Council of the Republic of China under contracts NSC89-2213-E-008-049, NSC87-2213-E-032-012, and NSC87-2811-E-032-0001.

References

1. A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Trans. Electronic Computers*, vol. EC-10, pp. 389–400, 1961. Reprinted in [30, vol. II, pp. 54–65].
2. I. Koren, *Computer arithmetic algorithms*, Prentice-Hall, 1993.
3. N. Takagi, H. Yasuura, and S. Yajima, "High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. Computers*, vol. C-34, no. 9, pp. 789–796, 1985.
4. S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," in *Proc. 8th Symp. Computer Arithmetic*, pp. 80–86, 1987.
5. Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi, "A high-speed multiplier using a redundant binary adder tree," *IEEE J. Solid-State Circuits*, vol. 22, no. 1, pp. 28–34, 1987.
6. A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer, and P.G.A. Jespers, "A new carry-free division algorithm and its application to a single-chip 1024-b RSA processor," *IEEE J. Solid-State Circuits*, vol. 25, no. 3, pp. 748–755, 1990.
7. K. Hwang, *Computer arithmetic, principles, architecture and design*, John Wiley & Sons, 1979.
8. S.M. Yen, C.S. Laih, C.H. Chen, and J.Y. Lee, "An efficient redundant-binary number to binary number converter," *IEEE J. Solid-State Circuits*, vol. 27, no. 1, pp. 109–112, 1992.
9. R.L. Rivest, A. Shamir, and L.M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.
10. Ç.K. Koç, "High-speed RSA implementations," Tech. Rep. TR 201, RSA Laboratories, Nov. 1994.
11. D.E. Knuth, *The art of computer programming/Seminumerical algorithms*, vol. 2, Addison-Wesley, 2nd edition, 1981.
12. P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM J. Computing*, vol. 10, pp. 638–646, 1981.
13. J. Bos and M. Coster, "Addition chain heuristics," in *Advances in Cryptology — CRYPTO '89*, vol. 435 of *Lecture Notes in Computer Science*, pp. 400–407, Springer-Verlag, 1990.

14. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Information Theory*, vol. IT-31, no. 4, pp. 469–472, 1985.
15. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997.
16. E.F. Brickell, D.M. Gordon, K.S. McCurley, and D.R. Wilson, "Fast exponentiation with precomputation: Algorithms and lower bounds," Preprint, Mar. 1995. An earlier version appeared in *Proc. of EUROCRYPT '92*.
17. F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," *Theoretical Informatics and Applications*, vol. 24, pp. 531–543, 1990.
18. A.D. Booth, "A signed binary multiplication technique," *The Quaterly J. Mechanics and Applied Mathematics*, vol. 4, pp. 236–240, 1951. Reprinted in [30, vol. I, pp. 100–104].
19. G.W. Reitwiesner, "Binary arithmetic," *Advances in Computers*, vol. 1, pp. 231–308, 1960.
20. J. Jedwab and C. Mitchell, "Minimum weight modified signed-digit representations and fast exponentiation," *Electronics Letters*, vol. 25, pp. 1171–1172, 1989.
21. H. Cohen, *A course in computational algebraic number theory*, Springer-Verlag, 1993.
22. Ö. Eğecioğlu and Ç.K. Koç, "Exponentiation using canonical recoding," *Theoretical Computer Science*, vol. 129, no. 2, pp. 407–417, 1994.
23. B.S. Kaliski, "The Montgomery inverse and its applications," *IEEE Trans. Computers*, vol. C-44, no. 8, pp. 1064–1065, 1995.
24. S. Arno and F.S. Wheeler, "Signed digit representations of minimal Hamming weight," *IEEE Trans. Computers*, vol. C-42, no. 8, pp. 1007–1010, 1993.
25. D.M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, pp. 129–146, 1998.
26. W.E. Clark and J.J. Liang, "On arithmetic weight for a general radix representation of integers," *IEEE Trans. Information Theory*, vol. IT-19, pp. 823–826, 1973.
27. H. Wu and M.A. Hasan, "Efficient exponentiation of a primitive root in GF(2^m)," *IEEE Trans. Computers*, vol. C-46, no. 2, pp. 162–172, 1997.
28. J. Omura and J. Massey, "Computational method and apparatus for finite field arithmetic," U.S. Patent # 4,587,627, 1986.
29. Ç.K. Koç, "Parallel canonical recoding," *Electronics Letters*, vol. 32, pp. 2063–2065, 1996.
30. E.E. Swartzlander, Jr., Ed., *Computer arithmetic*, vol. I and II, IEEE Computer Society Press, 1990.