Kimmo Järvinen, Juha Forsten and Jorma Skyttä, Efficient Circuitry for Computing τ-adic Non-Adjacent Form, in Proceedings of the 13th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2006, Nice, France, Dec. 10-13, 2006, pp. 232-235.

Efficient Circuitry for Computing τ-adic Non-Adjacent Form

Kimmo Järvinen, Juha Forsten and Jorma Skyttä
Helsinki University of Technology, Signal Processing Laboratory
Otakaari 5A, 02150 Espoo, Finland
Email: {Kimmo.Jarvinen, Juha.Forsten, Jorma.Skytta}@tkk.fi

***Abstract*— Elliptic curve point multiplication kP on an elliptic curve is required in every elliptic curve cryptosystem. The operation can be significantly accelerated by using a special type of elliptic curves called the Koblitz curves and by representing the integer k in τ-adic non-adjacent form (τNAF). Hardware-friendly modifications of existing τNAF conversion algorithms are presented and an efficient circuitry for the τNAF conversion is described with performance characteristics on an Altera Stratix-II S60C4 FPGA. To the authors' knowledge, this is the first published hardware implementation of the τNAF conversion.**

I. Introduction

Koblitz [1] and Miller [2] independently proposed the use of elliptic curves in cryptography in 1985. *Elliptic curve cryptography* (ECC) is a public-key cryptography method which offers a similar level of security than traditional public-key cryptography methods, such as RSA, with considerably shorter keys [3]. Hence, ECC has been studied much in the research community during the recent years.

The computational complexity of ECC computations can be considerably reduced by using a special type of elliptic curves called the Koblitz curves [4] and the τ-adic non-adjacent form (τNAF) representation for integers.

ECC, as public-key cryptography in general, is computationally expensive, and it commonly needs to be accelerated with dedicated hardware in order to meet the high requirements of modern communication systems. Hardware acceleration of ECC has been extensively studied in the community and many efficient accelerator architectures have been published. However, only few publications have considered the Koblitz curves and the τNAF method although it is one of the most efficient methods published so far. Lutz and Hasan [5] described an implementation for field programmable gate arrays (FPGAs) which used the τNAF method, but they calculated τNAF conversions with a C-program not a specific circuitry [6].

This paper considers the binary to τNAF conversion. The conversion algorithm was suggested by Solinas [7]. In addition to the actual conversion, also a reduction algorithm needs to be used in order to utilize the full potential of the method. Instead of the original reduction algorithm [7], an algorithm presented by Lutz in [6] is considered in this paper. Existing algorithms are modified in order to guarantee efficient hardware-based implementation and architectures for the τNAF conversion and reduction are described with analysis on their performance on FPGAs. To the best of the authors' knowledge, hardware-based implementations of the τNAF conversion have not been previously presented in the literature.

Performance characteristics and resource requirements are provided for an Altera Stratix-II EP2S60F1020C4 FPGA [8], which was used for prototyping. Although the prototyping was performed with FPGAs, VHDL (Very high-speed integrated circuit Hardware Description Language) describing the design is fully portable and could be used, e.g., for application specific integrated circuits (ASICs), too.

The remainder of the paper is organized as follows. The basics of ECC and the τNAF conversion are considered in Sec. II. Algorithm modifications and implementations are described in Sec. III. Finally, results are presented in Sec. IV and conclusions are drawn in Sec. V.

II. Elliptic Curves and τNAF Conversion

Elliptic curve point multiplication (ECPM) on an elliptic curve is the basic operation of any elliptic curve cryptosystem. Let $E(\mathbb{F}_{2^m})$ be an elliptic curve defined over a finite extension field $\mathbb{F}_{2^m}$. ECPM is defined on $E(\mathbb{F}_{2^m})$ as follows:

$$Q = kP = \underbrace{P + P + \ldots P}_{k \text{ times}} \tag{1}$$

where $Q, P \in E(\mathbb{F}_{2^m})$ and k is a large integer. The binary expansion of k is given by $k = \sum_{i=0}^{\ell-1} k_i 2^i$, where $k_i \in \{0, 1\}$. The length of k, ℓ, satisfies $\ell \leq m$. ECPM can be calculated with the binary method using consecutive elliptic curve point additions and doublings. A point doubling, $2P$, is performed for every k_i and a point addition, $P_1 + P_2$, is performed if $k_i = 1$. The average density of ones in k is $1/2$ and thus (1) requires ℓ point doublings and, on average, $\ell/2$ point additions. [3]

The density of non-zero coefficients in k can be reduced to 1/3 by representing k in *non-adjacent form* (NAF) by using the signed bit representation, i.e. $k_i \in \{0, 1, \bar{1}\}$, where $\bar{1} = -1$ [3]. For example, if $k = 7 = 111_b$, its NAF is $100\bar{1}_{\text{NAF}}$. As the name suggests, there are no adjacent non-zero values in the NAF of k, i.e., $|k_i| + |k_{i+1}| \leq 1$. If k is represented in NAF, only $\ell/3$ point additions or subtractions, $P_1 \pm P_2$, and ℓ point doublings are required in (1) on average [3].

The computational complexity of a point doubling is smaller than the complexity of a point addition. Nevertheless, because more point doublings than point additions are required in computing (1), performance can be dramatically increased by replacing point doublings with less expensive operations. Methods replacing doublings with cheaper operations include point halving techniques and Frobenius maps on the Koblitz curves [3]. This paper considers the latter one.

The *Koblitz curves* are a special type of curves for which the Frobenius endomorphism can be used for improving the performance of computing (1). The Koblitz curves are defined over $\mathbb{F}_2$ as follows:

$$E_a : y^2 + xy = x^3 + ax^2 + 1 \tag{2}$$

where $a \in \{0, 1\}$ [4]. The *Frobenius map* $\tau : E_a(\mathbb{F}_{2^m}) \mapsto E_a(\mathbb{F}_{2^m})$ for a point $P = (x, y)$ on $E_a(\mathbb{F}_{2^m})$ is defined by

$$\tau(x, y) = (x^2, y^2), \quad \tau(\mathcal{O}) = \mathcal{O} \tag{3}$$

where $\mathcal{O}$ is the point at infinity [3]. It stands that $(\tau^2 + 2)P = \mu\tau(P)$ for all $P \in E_a(\mathbb{F}_{2^m})$, where $\mu = (-1)^{1-a}$. Thus, it follows that the Frobenius map can be considered as a multiplication with a complex number $\tau = \frac{\mu + \sqrt{-7}}{2}$. [7]

It can be seen in (3) that only two squarings in $\mathbb{F}_{2^m}$ are required in computation of τP. If projective coordinates are used in order to reduce expensive inversions in $\mathbb{F}_{2^m}$, three squarings in $\mathbb{F}_{2^m}$ are required. Squarings are cheap to perform, and they are practically free in hardware if a normal basis is used for $\mathbb{F}_{2^m}$. [3]

Point doublings can be replaced by Frobenius maps if the integer k is represented with τ-adic expansion, i.e. $k = \sum_{i=0}^{\ell_u - 1} u_i \tau^i$, where

1-4244-0395-2/06/$20.00 ©2006 IEEE.

$u_i \in \{0, 1\}$. In analogy with the binary expansion, the density of non-zero bits can be reduced to $1/3$ by using a NAF representation, where $u_i \in \{0, 1, \bar{1}\}$. An algorithm for computing *τ-adic non-adjacent form* (τNAF) for $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$, where $\mathbb{Z}[\tau]$ is the ring of polynomials in τ with integer coefficients, was suggested by Solinas [7] and it is presented in Alg. 1. [3]

Algorithm 1 Computing the τNAF of an element in $\mathbb{Z}[\tau]$ [7]

INPUT: $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$
OUTPUT: $u = \tau\text{NAF}(\kappa)$
  $i \leftarrow 0$
  while $r_0 \neq 0$ and $r_1 \neq 0$ **do**
    if r_0 is odd **then**
      $u_i \leftarrow 2 - (r_0 - 2r_1 \mod 4)$
      $r_0 \leftarrow r_0 - u_i$
    else
      $u_i \leftarrow 0$
    $t \leftarrow r_0$; $r_0 \leftarrow r_1 + \mu r_0/2$; $r_1 \leftarrow -t/2$; $i \leftarrow i + 1$
  Return $(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$

If the length of $\tau\text{NAF}(\kappa)$, ℓ_u, is greater than 30, it satisfies

$$\log_2(N(\kappa)) - 0.55 < \ell_u < \log_2(N(\kappa)) + 3.52 \quad (4)$$

where $N(\kappa)$ is the norm of κ which is given by

$$N(r_0 + r_1\tau) = r_0^2 + \mu r_0 r_1 + 2r_1^2. \quad (5)$$

The average length of $\tau\text{NAF}(\kappa)$ is $\log_2(N(\kappa))$. Hence, the length of $\tau\text{NAF}(k)$ is approximately $2\log_2(k)$ and $\ell_u \approx 2\ell$. Although the density of u is 1/3, (1) would require as many as $\ell_u/3 \approx 2\ell/3$ point additions and $\ell_u \approx 2\ell$ Frobenius maps. It is, however, possible to reduce the length to approximately ℓ because, if $n \equiv k \mod (\tau^m - 1)$, then $kP = nP$, because $(\tau^m - 1)P = \mathcal{O}$. It can be also shown that, if $\rho \equiv k \mod \delta$, where $\delta = \frac{\tau^m - 1}{\tau - 1}$, then $kP = \rho P$. [7]

There are two different approaches to reduce the length of $\tau\text{NAF}(k)$. The first one is to find $\rho = s_0 + s_1\tau$ with a minimum norm $N(\rho)$, and then apply Alg. 1 to ρ for finding u with length $\ell_u \approx \ell$. This approach is here referred to as the *pre-processing* method. In the second approach, Alg. 1 is applied directly to k and the result is then processed so that the length reduces approximately to ℓ. This method is referred to as the *post-processing* method. This paper concentrates on the post-processing method, but the pre-processing method is shortly considered as well.

Pre-processing can be done by performing a modular division in $\mathbb{Z}[\tau]$ [7]. This method always results in ρ with $N(\rho)$ as small as possible but requires computation of two multiprecision integer divisions, which makes it very difficult and slow to implement on many platforms [3]. In order to circumvent this problem, a partial reduction algorithm was developed [7]. This algorithm does not require any multiprecision divisions but may result in sub-optimal results, i.e. $N(\rho)$ not as small as possible. The partial reduction requires several integer multiplications [7].

Post-processing introduced in [6] is presented in Alg. 2, which combines two algorithms from [6]. The post-processing is performed in two phases. First, a *reduction*, where the ℓ_u-bit result of $u = \tau\text{NAF}(k)$ is reduced into an m-bit v, takes place. Second, NAF, which was lost in the reduction, is recovered in a *regeneration*. The reduction takes advantage of the fact that any τ-adic integer, u, can be reduced modulo $(\tau^m - 1)$ without changing the result of uP. Thus, τ^m can be factored out from u [6]. The result of the reduction, v, is no longer in NAF but NAF can be regenerated by applying Alg. 1 to each $(v_i + v_{i+1}\tau)\tau^i$ starting from $i = 0$ [6] (while-loop in Alg. 2). The length of the result of a regeneration is m bits, on average [6]. This method does not require any multiprecision divisions or multiplications, and it is easily implementable on various platforms. The downside of the post-processing method is that Alg. 1 requires on average $2m$ cycles while it requires only m cycles on average if the pre-processing is utilized. Post-processing may also result in sub-optimal representations similarly as the partial reduction.

Algorithm 2 Reduction and regeneration algorithm [6]

INPUT: $u = \sum_{i=0}^{\ell_u - 1} u_i\tau^i$, $u_i \in \{0, 1, \bar{1}\}$, m
OUTPUT: $w = \sum_{i=0}^{\ell_w - 1} w_i\tau^i$, $w_i \in \{0, 1, \bar{1}\}$
  $(v_{m-1}, v_{m-2}, \ldots, v_1, v_0) \leftarrow (0, 0, \ldots, 0, 0)$
  if $\ell_u > 2m$ **then**
    $(v_{m-1}, \ldots, v_0) \leftarrow (0, 0, \ldots, 0, u_{\ell_u - 1}, \ldots, u_{2m})$
  if $\ell_u > m$ **then**
    $(v_{m-1}, \ldots, v_0) \leftarrow (v_{m-1}, \ldots, v_0) + (u_{2m-1}, \ldots, u_m)$
  $(v_{m-1}, \ldots, v_0) \leftarrow (v_{m-1}, \ldots, v_0) + (u_{m-1}, \ldots, u_0)$
  $i \leftarrow 0$
  while $v_j \neq 0$ for some $j > i$ **do**
    if $v_i = 0$ **then**
      $i \leftarrow i + 1$
    else
      $(t_1, t_0) \leftarrow (v_{i+1}, v_i)$; $(v_{i+1}, v_i) \leftarrow (0, 0)$
      $v \leftarrow v + \tau\text{NAF}(t_0 + t_1\tau)$; $w_i \leftarrow v_i$; $i \leftarrow i + 1$
  Return $(w_{i-1}, w_{i-2}, \ldots, w_1, w_0)$

III. Hardware Architecture

An efficient hardware architecture for the τNAF conversion is presented in this section. The architecture implements Algs. 1 and 2, but both of the algorithms are modified in order to guarantee efficient hardware-based implementation. The architectures for Algs. 1 and 2 are presented in Secs. III-A and III-B, respectively.

A. τNAF Conversion

Implementation of Alg. 1 is straightforward. However, certain observations result in very efficient implementations. First, $r_0 \leftarrow r_0 - u_i$ can be moved out from the if-statement because, if r_0 is even, $u_i = 0$ resulting $r_0 - u_i = r_0$. Furthermore, the if-statement can be removed entirely by computing u_i as presented next.

The computation of u_i can be performed very efficiently by observing the two least significant bits (LSBs) of r_0 and r_1. If r_0 is odd in Alg. 1, then $r_0 - 2r_1 \mod 4$ is also odd, i.e. either 1 or 3, which gives $u_i = \pm 1$. Furthermore, if r_0 is even, then $r_0 - 2r_1 \mod 4$ is also even, i.e. either 0 or 2 resulting $u_i = 2$ or $u_i = 0$, respectively. By setting that u_i is the two LSBs of the result of $r_0 - 2r_1$ considered as a 2-bit signed integer, the operation of Alg. 1 does not change if the exception case $u_i = 2$ is handled correctly. Let r_{i_j} denote the jth bit of r_i. Obviously, $(r_{0_1}, r_{0_0}) - (r_{1_0}, 0) = (r_{0_1} \oplus r_{1_0}, r_{0_0})$, where $\oplus$ denotes exclusive-or (XOR). One additional AND-operation, denoted by $\otimes$, must be introduced in order to force u_i to zero if r_0 is even, i.e., if $r_{0_0} = 0$. Thus, the sign-bit of u_i is given by $(r_{0_1} \oplus r_{1_0}) \otimes r_{0_0}$.

A modification of Alg. 1 which is especially suitable for hardware implementations is presented in Alg. 3. A block diagram implementing Alg. 3 is given in Fig. 1. The adders and the registers are $D + 1$ bits wide, where D must satisfy $D \geq \max(\ell) = m$. The two's complement representation is used for r_0 and r_1. The implementation outputs one u_i starting from u_0 in one clock cycle, and $\tau\text{NAF}(k)$ requires on average $2m$ clock cycles if $\ell = m$.

Algorithm 3 Modification of Alg. 1

INPUT: $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$
OUTPUT: $u = \tau\text{NAF}(\kappa)$
  $i \leftarrow 0$
  while $r_0 \neq 0$ and $r_1 \neq 0$ **do**
    $u_i \leftarrow ((r_{0_1} \oplus r_{1_0}) \otimes r_{0_0}, r_{0_0})$
    $r_0 \leftarrow r_0 - u_i$
    $t \leftarrow r_0$; $r_0 \leftarrow r_1 + \mu r_0/2$; $r_1 \leftarrow -t/2$; $i \leftarrow i + 1$
  Return $(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$

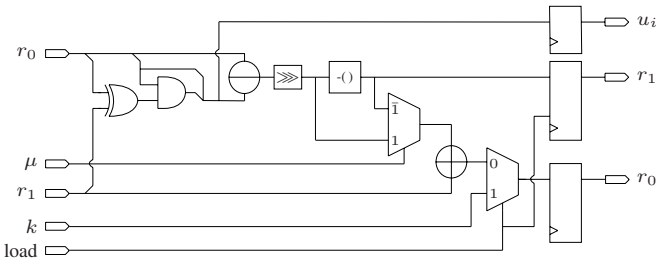

Fig. 1. Circuitry for the τNAF conversion, i.e., an implementation of Alg. 3. The widths of r_0 and r_1 are $D+1$ bits and u_i is a 2-bit signed integer, whose sign-bit is the output of the AND-gate and the magnitude bit is r_{0_0}. Division by 2 is a shift to the right ($\ggg$). The inputs of the XOR-gate are the LSB of r_1 and the second LSB of r_0. The second input for the AND-gate is the LSB of r_0. The load signal loads k into r_0 and clears r_1, i.e. $r_0 = k$ and $r_1 = 0$. Also comparators for r_0 and r_1, which are not depicted, are required in order to determine when the conversion is ready.

B. Reduction and Regeneration

The key idea in implementing Alg. 2 is to perform $\tau\text{NAF}(t_0 + t_1\tau)$ by using a fixed *look-up table* (LUT). This way the $\tau\text{NAF}(t_0 + t_1\tau)$ operation can be done entirely in one clock cycle while, if also this would be performed by using the circuitry presented in Sec. III-A, each $\tau\text{NAF}(t_0+t_1\tau)$ computation would require as many clock cycles as is the length of that particular $\tau\text{NAF}(t_0 + t_1\tau)$.

The construction of the LUT is considered in Sec. III-B.1 and the implementation of the rest of the reduction and regeneration circuitry is presented in Sec. III-B.2.

1) Look-Up Table for τNAF: All possible values of t_0 and t_1 must be known in order to construct a LUT for $\tau\text{NAF}(t_0 + t_1\tau)$. It is known based on the definition of τNAF that $u_i \in \{0, 1, \bar{1}\}$ and $|u_i| + |u_{i+1}| \leq 1$. Because $\log_2(k) < m$, it follows from (4) that $\ell_u \leq 2m + 3$. Thus, $v_i \in \{0, \pm1, \pm2, \pm3\}$, if $0 \leq i < 3$, and, if $i \geq 3$, $v_i \in \{0, \pm1, \pm2\}$. Because of the non-adjacency of u, it also follows that $|v_i| + |v_{i+1}| \leq 3$, if $0 \leq i < 3$, and $|v_i| + |v_{i+1}| \leq 2$, if $i \geq 3$. These are referred to as the *initial condition* of v.

Let h denote the result of $\tau\text{NAF}(t_0 + t_1\tau)$. It can be easily checked that, for v satisfying the initial condition, the maximum length of h is 7. Adding h to v sometimes results in values which do not fulfill the initial condition. All possible values of v_i and v_{i+1} can be determined as follows. Take all v of length L satisfying the initial condition and pad them with zeros to infinity resulting $a = (\ldots, 0, 0, 0, v_{L-1}, v_{L-2}, \ldots, v_1, v_0)$. Let A denote the set including all possible a. Run regeneration for the first $L-1$ coefficients of each a. Coefficients a_i where $i > L-1$ may now have non-zero values. Let B be the set of all different $b = (\ldots, a_{L+2}, a_{L+1}, a_L, a_{L-1})$ resulted by the regenerations. Define A' to include all $a' = (\ldots, 0, 0, 0, v_{2L-1}, v_{2L-2}, \ldots, v_L, 0)$ satisfying the initial condition. Notice that $A' \subset A$. Now, for each element of B, compute $c = b + a'$ for all elements of A' and perform the regeneration for the first $L-1$ coefficients of c. If the result $(\ldots, c_{L+2}, c_{L+1}, c_L, c_{L-1})$ is not included in B, add it there. Continue this process until the entire B has been processed. The above process goes through all possible combinations of v during a regeneration. When the process with $L = 9$ was run by using a C-program, it was verified that $t_i \in \{0, \pm1, \pm2, \pm3, \pm4\}$ and $|t_0 + t_1| \leq 4$ for all possible v satisfying the initial condition. The maximum length of h is 7 also in this case.

The construction of the LUT is straightforward when all possible values of t_0 and t_1 are known. The LUT is presented in Fig. 2. It can be seen that $\tau\text{NAF}(t_0+t_1\tau) = -\tau\text{NAF}(-t_0-t_1\tau)$ and, based on this symmetry, the size of the LUT can be halved. Let σ denote the sign of t_1. Then, $h = \tau\text{NAF}(t_0 + t_1\tau)$ can be calculated with a LUT having half the size of the original one by setting

$$h = \sigma \cdot \tau\text{NAF}(\sigma t_0 + |t_1|\tau). \tag{6}$$

Certain additional logic needs to be implemented in order to compute $|t_1|$ and multiplications with σ but, still, the overall area requirement of the $\tau\text{NAF}(t_0 + t_1\tau)$ computation reduces.

t_0 \ t_1	−4	−3	−2	−1	0	1	2	3	4
−4					$0\bar{1}00\bar{1}00$	$0\bar{1}0\bar{1}0\bar{1}0$			
−3				$0001001\bar{1}$	$0\bar{1}00\bar{1}01$	$0000\bar{1}0\bar{1}$	$0\bar{1}0\bar{1}001$		
−2			$0001\bar{1}0$	0001000	$0001\bar{1}10$	0000100	$00\bar{1}00\bar{1}0$	$00\bar{1}0000$	
−1		$\bar{1}0\bar{1}0\bar{1}01$	$001\bar{1}01\bar{1}$	0001001	$000\bar{1}001$	0000101	$00\bar{1}0\bar{1}0\bar{1}$	$00\bar{1}0001$	$\bar{1}00\bar{1}00\bar{1}$
0	$0000\bar{4}0$	$0000\bar{3}0$	$0000\bar{2}0$	$0000\bar{1}0$	0000000	0000010	0000020	0000030	0000040
1	$\bar{1}00\bar{1}001$	$001000\bar{1}$	0010101	$0000\bar{1}0\bar{1}$	0000001	$000\bar{1}00\bar{1}$	$00\bar{1}0\bar{1}01$	$1010\bar{1}0\bar{1}$	
2		0010000	$0010\bar{1}10$	$0000\bar{1}00$	$000\bar{1}0\bar{1}0$	$000\bar{1}000$	$000\bar{1}\bar{1}10$		
3			$0\bar{1}0\bar{1}00\bar{1}$	$0000\bar{1}01$	$0\bar{1}00\bar{1}0\bar{1}$	$000\bar{1}001$			
4				$0\bar{1}0\bar{1}0\bar{1}0$	$0\bar{1}00\bar{1}00$				

$\mu = 1$

t_0 \ t_1	−4	−3	−2	−1	0	1	2	3	4
−4					$0\bar{1}00\bar{1}00$	$\bar{1}0\bar{1}00\bar{1}0$			
−3				$0000\bar{1}0\bar{1}$	$0\bar{1}00\bar{1}01$	$000\bar{1}00\bar{1}$	$\bar{1}0\bar{1}0001$		
−2			$00\bar{1}0\bar{1}0$	0000100	$000\bar{1}0\bar{1}0$	$000\bar{1}000$	$000\bar{1}0\bar{1}0$	$\bar{1}0\bar{1}0\bar{1}00$	
−1		$00\bar{1}0001$	$00\bar{1}0\bar{1}0\bar{1}$	0000101	$0000\bar{1}0\bar{1}$	$000\bar{1}001$	$0010\bar{1}0\bar{1}$	$\bar{1}0\bar{1}0\bar{1}0\bar{1}$	$\bar{1}00\bar{1}00\bar{1}$
0	$0000\bar{4}0$	$0000\bar{3}0$	$0000\bar{2}0$	$0000\bar{1}0$	0000000	0000010	0000020	0000030	0000040
1	$\bar{1}00\bar{1}001$	$\bar{1}0\bar{1}0\bar{1}0\bar{1}$	$00\bar{1}0\bar{1}01$	$0001\bar{1}0\bar{1}$	$0000\bar{1}01$	$0000\bar{1}0\bar{1}$	$0010\bar{1}0\bar{1}$	$001000\bar{1}$	
2		$\bar{1}0\bar{1}0\bar{1}00$	$000\bar{1}0\bar{1}0$	0001000	$000\bar{1}\bar{1}10$	$0000\bar{1}00$	$00\bar{1}00\bar{1}0$		
3			$\bar{1}0\bar{1}0000\bar{1}$	0001001	$0\bar{1}00\bar{1}0\bar{1}$	$0000\bar{1}01$			
4				$\bar{1}0\bar{1}0010$	$0\bar{1}00\bar{1}00$				

$\mu = -1$

Fig. 2. Look-up table for $h = \tau\text{NAF}(t_0 + t_1\tau)$ calculation. The values are presented as $h_6h_5h_4h_3h_2h_1h_0$. Dark grey areas are never used and need not to be implemented. Light grey areas can be reduced from the LUT based on symmetry. Notice that the row $t_0 = 0$ is not $\tau\text{NAF}(t_1\tau)$ because in that case no changes in v should occur (see Alg. 2).

2) Circuitry: Because the τNAF conversion block outputs u_i in serial, the reduction is modified to take u_i in serial instead of in parallel as in Alg. 2. This way the reduction of Alg. 2 can be performed concurrently with the execution of Alg. 3. When u_i are input in serial starting from u_0, v can be calculated by first setting v to zero and then by adding u_i to $v_{(i \mod m)}$.

An efficient algorithm for performing the reduction and regeneration in hardware is presented in Alg. 4. Notice that all operations are

Algorithm 4 Modification of Alg. 2

INPUT: $u = \sum_{i=0}^{\ell_u - 1} u_i\tau^i$, $u_i \in \{0, 1, \bar{1}\}$, m, μ
OUTPUT: $w = \sum_{i=0}^{\ell_w - 1} w_i\tau^i$, $w_i \in \{0, 1, \bar{1}\}$
  $(v_{m-1}, v_{m-2}, \ldots, v_1, v_0) \leftarrow (0, 0, \ldots, 0, 0)$
  for $i = 0$ to $\ell_u - 1$ **do**
    $v_{(i \mod m)} \leftarrow v_{(i \mod m)} + u_i$
  $i \leftarrow 0$
  while $i < m$ or $v_j \neq 0$ for some $j < 7$ **do**
    $(h_6, \ldots, h_0) \leftarrow \tau\text{NAF}_{\text{LUT}}(v_0 + v_1\tau)$
    $(v_6, \ldots, v_2) \leftarrow (v_6, \ldots, v_2) + (h_6, \ldots, h_2)$; $v_1 \leftarrow h_1$
    $w_i \leftarrow h_0$; $v \leftarrow \text{ShiftRight}(v)$; $i \leftarrow i + 1$
  Return $(w_{i-1}, w_{i-2}, \ldots, w_1, w_0)$

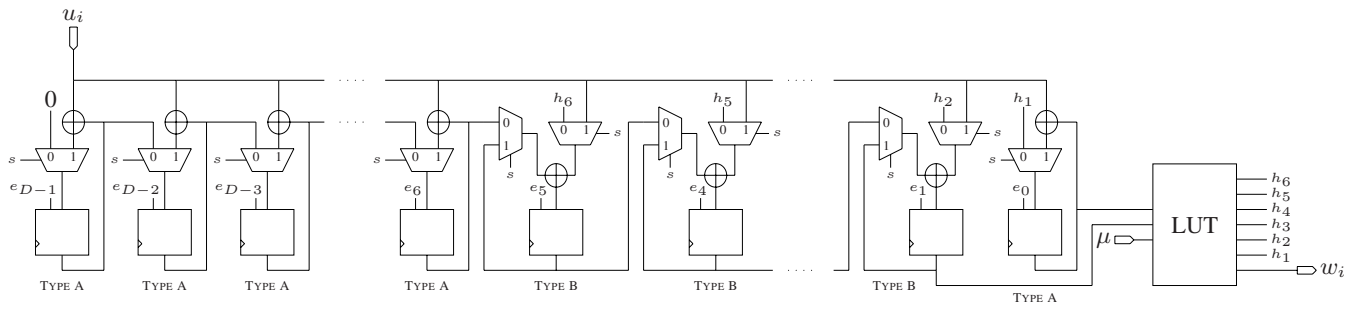

Fig. 3. Circuitry for the reduction and regeneration, i.e., an implementation of Alg. 4. If the mode select signal $s = 1$, the circuit performs the reduction of Alg. 4 and, if $s = 0$, the register chain acts as a shift register and the regeneration is performed. The enable signals e_j operate as follows: during the reduction ($s = 1$), $e_j = 1$ for $j = i \mod m$ and $e_j = 0$ otherwise and, during the regeneration ($s = 0$), $e_j = 1$ for all j. Type A registers are 3 bits wide, excluding the register for v_0 which is a 4-bit register. All Type B registers are 4 bits wide. In total, $D - 5$ Type A and 5 Type B registers are required.

directed to v and, therefore, both reduction and regeneration can be performed with only one register chain.

The reduction requires registers and circuitry for $v_{(i \mod m)} + u_i$ computations. The regeneration requires a shift register, adders for $v+h$, and the LUT. Both of the operations can be performed with the circuitry shown in Fig. 3. It consists mainly of two kind of register blocks called Type A and Type B registers. The *Type A registers* have a register for storing v_i and logic for saving a new value to the register and for accumulating the value of the register. The *Type B registers* also have a register for storing v_i but, in them, the value to be stored is a sum of two values selected from four by using two multiplexors. Both Type A and Type B registers are arranged so that they perform similarly during the reduction. On the other hand, during the regeneration, Type A registers act as a shift register while Type B registers both shift v and compute and store the result of $v + h$. The circuitry outputs w_i in serial starting from w_0.

The latency of the reduction is approximately $2m$ clock cycles. However, the reduction can be performed concurrently with the τNAF(k) computation and its effective latency reduces to one clock cycle. Once the reduction is performed, the regeneration takes m clock cycles on average. Thus, the average latency of computing and reducing the length of τNAF(k) becomes $3m + 2$ clock cycles on the given circuitry, including loading k into the circuit.

IV. Results

The designs were described in VHDL and synthesized by using Altera Quartus-II 5.1 SP1 design software. Results of the implementations on an Altera Stratix-II EP2S60F1020C4 FPGA [8] are presented in Table I. The maximum clock frequencies $f_{\max}$ and resource requirements, presented as the number of consumed adaptive look-up tables (ALUTs) [8] and registers, were given by Quartus-II. A generic implementation capable of performing conversions for any Koblitz curve $E_a(\mathbb{F}_{2^m})$, where $m \leq 256$, and curve specific implementations for $E_1(\mathbb{F}_{2^{163}})$ and $E_0(\mathbb{F}_{2^{233}})$ are included in Table I. The curves $E_1(\mathbb{F}_{2^{163}})$ and $E_0(\mathbb{F}_{2^{233}})$ are included in the NIST recommended elliptic curves for federal government use [9]. Fixing the curve reduces area and increases $f_{\max}$ because smaller adders and fewer registers are needed and the size of the LUT can be halved, i.e., only $\mu = 1$ or $\mu = -1$ needs to be implemented. In all cases, the critical path determining $f_{\max}$ consists of the computation of τNAF(k) as presented in Sec. III-A.

TABLE I
Implementation results on a Stratix-II S60C4

Curve	D	ALUTs	Regs.	$f_{\max}$ (MHz)
$E_a(\mathbb{F}_{2^m})$	256	2,285	1,564	55.72
$E_0(\mathbb{F}_{2^{233}})$	233	1,800	1,198	58.89
$E_1(\mathbb{F}_{2^{163}})$	163	1,433	988	80.44

V. Conclusions

An efficient hardware circuitry for converting integers to τNAF was presented. The circuitry suits well for implementations on various platforms including constrained environments, because it is compact and does not require any large multiplications or divisions. To the best of the authors' knowledge, this was the first hardware-based τNAF implementation presented in the literature so far.

It is concluded that the τNAF conversion, including the reduction of the length, can be efficiently computed with a small amount of resources by using the circuitry presented in this paper.

Acknowledgements

This research was conducted within the Packet Level Authentication (PLA) project at the Helsinki University of Technology. The PLA project is funded by Tekes — Finnish funding agency for technology and innovation.

References

[1] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[2] V. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85*, ser. Lecture Notes in Computer Science, vol. 218, Santa Barbara, California, USA, Aug. 18–22, 1985, pp. 417–426.

[3] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. New York, USA: Springer-Verlag, 2004.

[4] N. Koblitz, "CM-curves with good cryptographic properties," in *Advances in Cryptology — CRYPTO '91*, ser. Lecture Notes in Computer Science, vol. 576, Santa Barbara, California, USA, Aug. 11–15, 1991, pp. 279–287.

[5] J. Lutz and A. Hasan, "High performance FPGA based elliptic curve cryptographic co-processor," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004)*, vol. 2, Las Vegas, Nevada, USA, Apr. 5–7, 2004, pp. 486–492.

[6] J. Lutz, "High performance elliptic curve cryptographic co-processor," Master's thesis, University of Waterloo, Canada, 2003.

[7] J. Solinas, "Efficient arithmetic on Koblitz curves," *Designs, Codes and Cryptography*, vol. 19, no. 2–3, pp. 195–249, Mar. 2000.

[8] "Stratix II device handbook," vol. 1–2, ver. 4.1, Altera Corporation, Apr. 2006. [Online]. Available: http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf

[9] National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS)*, Federal Information Processing Standard, Std. FIPS PUB 186-2, Jan. 27, 2000. [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf