Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms
Sujoy Sinha Roy, Junfeng Fan and Ingrid Verbauwhede, Fellow, IEEE

Abstract—Koblitz curves are a class of computationally efficient elliptic curves where scalar multiplications can be accelerated using $\tau$NAF representations of scalars. However conversion from an integer scalar to a short $\tau$NAF is a costly operation. In this paper we improve the recently proposed scalar conversion scheme based on division by $\tau^2$. We apply two levels of optimizations in the scalar conversion architecture. First we reduce the number of long integer subtractions during the scalar conversion. This optimization reduces the computation cost and also simplifies the critical paths present in the conversion architecture. Then we implement pipelines in the architecture. The pipeline splitting increases the operating frequency without increasing the number of cycles. We have provided detailed experimental results to support our claims made in this paper.

Index Terms—Koblitz Curve, Lazy Reduction, Scalar Multiplication, Cryptography, FPGA, Architecture, Pipelining.

I. INTRODUCTION
Elliptic curve cryptography (ECC) is the modern standard for public key cryptography thanks to its higher bit security and implementation friendliness on embedded platforms. Elliptic curve scalar multiplication (ECSM) is the soul of any ECC processor. In ECSM, a point $P$ on an elliptic curve is multiplied by a large scalar $k$ to get $kP$. The standard way to perform ECSM is to use the double and add algorithm [1], [2], [3] where point doublings are performed for every key bit and point additions are performed for every nonzero key bit. As the number of point additions is determined by the Hamming weight of the scalar, computation time of an ECSM depends on the scalar $k$. Numerous works are present in the literature on improving the computation time by applying various optimizations such as efficient representation of scalars, efficient group operations, use of projective coordinate systems, and use of computation friendly class of elliptic curves etc. Architectural optimizations depending on the platforms add further acceleration [4], [5], [6], [7].

Koblitz curves [8] are a special class of elliptic curves, where the Frobenius endomorphism can be utilized to represent an integer scalar in a $\tau$adic form. With this special representation of a scalar, costly point doublings are replaced by cheaper Frobenius operations. Solinas extended the work and proposed $\tau$adic non-adjacent-form known as $\tau$NAF [9].

However, the length of both $\tau$adic and $\tau$NAF representations of a scalar are almost twice the length of the integer scalar. Thus, increase in the length (thus increase in the number of point additions) becomes a negating factor in achieving the acceleration offered by the Frobenius endomorphism. Length reduction schemes were first introduced by Meier and Staffelback in [10] and were later improved by Solinas in [9]. The length reduction algorithms proposed by Solinas are efficient in software, but are not amiable to hardware implementations due to the presence of multi-precision integer multiplications and divisions.

On the hardware side, very few research papers exist in the literature on designing efficient scalar conversion algorithms. In [11], [12], the conversions are performed in a software processor, while the ECSMs are performed on a dedicated hardware. Such an approach is well suited when the scalar multiplier architecture is slow. However, the present speed records for ECSMs in hardware have made such an approach a bottleneck [13], [6]. The first hardware implementation of scalar conversion was reported in [14]. Later [15], [16] keep the conversion units in hardware along with the scalar multipliers. Still the hardware versions of the converters are slow and have a large area.

The first hardware implementation friendly scalar conversion scheme was reported by Brumley and Järvinen in [17]. Their conversion algorithm repeatedly divides the scalar by $\tau$ to generate a length reduced scalar. Due to its sequential nature, the authors call the algorithm lazy reduction. The algorithm is very simple as it requires only integer addition/subtraction and shifting. In [18] Adikari, Dimitrov, and Järvinen proposed an improvement over [17]. Since division by $\tau^2$ is cheap in hardware, their double lazy reduction scheme reduces the computation time to nearly half by using division by $\tau^2$ instead of $\tau$.

In this paper we propose acceleration techniques for the double lazy reduction algorithm [18]. We observe that several additions and subtractions can be eliminated during the scalar reduction and the $\tau$NAF generation. Subtraction or addition of the nonzero remainders are replaced by alteration of the low order bits in the operands, use of one’s complement of the operands and by considering borrow/carry inputs in subtracter and adder circuits. We eliminate unnecessary subtractions from zero using iterative property of the conversion steps. Such optimizations reduce the number of integer adder and subtracter circuits from the critical paths of the conversion architecture without affecting the cycle requirement. With the proposed acceleration techniques, we achieve a improvement in the operating frequency by at least 12.5% and 17.9% compared
to [18] for the fields $F_{2^{233}}$ and $F_{2^{283}}$ respectively.

Next we perform efficient pipelining in the proposed conversion architecture. Using the iterative property of the conversion steps, we pipeline the conversion architecture in such a way that the pipelined stages are always utilized. Due to its bubble free nature, almost no cycles are wasted to satisfy the data dependencies between the pipeline stages. Our two-stage pipelined conversion architecture achieves acceleration by 35.5% and 40% compared to [18] for the fields $F_{2^{233}}$ and $F_{2^{283}}$ respectively.

The organization of the paper is: Section II has a brief mathematical background on the Koblitz curves and scalar conversion techniques. In Section III, computational optimizations for the double lazy reduction algorithm are discussed. Section IV shows optimizations in the $\tau$NAF generation steps. A hardware architecture for the scalar conversion is designed in Section V and an efficient pipeline strategy is presented in Section VI. Experimental results are presented in Section VII. The final section draws the conclusions.

II. Preliminaries

The Koblitz curves over $F_{2^m}$ have the following form.

$$E_a : y^2 + xy = x^3 + a \cdot x^2 + 1, \quad a \in \{0, 1\}$$

We denote the Koblitz curve group by $E_a(F_{2^m})$. In $E_a(F_{2^m})$ the Frobenius mapping can be applied to reduce the complexity of ECSM. The Frobenius mapping $\tau : E_a(F_{2^m}) \to E_a(F_{2^m})$ is defined below.

$$\tau(O) = O, \quad \tau P(x, y) = Q(x^2, y^2)$$

Application of $\tau$ on any point $P$ squares the coordinates and gives another point $Q$ on the curve. Since squaring in $F_{2^m}$ is cheap [1], [19], computing the Frobenius map of a point is also easy. The map operator $\tau$ satisfies the relation $\tau^2 + 2 = \mu \tau$, where $\mu = (-1)^{1-a}$.

The ring of polynomials in $\tau$ with integer coefficients is denoted by $\mathbb{Z}[\tau]$. For any polynomial $u_{t-1} \tau^{t-1} + \ldots + u_1 \tau + u_0 \in \mathbb{Z}[\tau]$ with $u_i \in \{0, 1\}$, and any base point $P$ on a Koblitz curve, we see the following relation.

$$[u_{t-1} \tau^{t-1} + \ldots + u_0]P = [u_{t-1}] \tau^{t-1}P + \ldots + [u_0]P$$  (1)

In the above equation, the base point $P$ is multiplied by a scalar which is an element of $\mathbb{Z}[\tau]$. During ECSM, only point addition and Frobenius operations are performed.

Solinas in [9] proposed algorithms to convert integer scalars into polynomials in $\tau$ with coefficients $u_i \in \{0, 1\}$ ($\tau$adic form). He also showed that number of point additions can be reduced using non-adjacent form which is known as a $\tau$NAF of a scalar. Calculation of the $\tau$NAF from a scalar requires iterative divisions by $\tau$. Here are two theorems from [9] related to the division of any element $\alpha = (d_0 + d_1 \tau) \in \mathbb{Z}[\tau]$ by $\tau$.

**Theorem 1** : $\alpha$ is divisible by $\tau$ when $d_0$ is even. The result of the division when stored in $(d_0, d_1)$ is $(d_0, d_1) \leftarrow (\mu d_0/2 + d_1, -d_0/2)$.

**Theorem 2** : $\alpha$ is divisible by $\tau^2$ when $d_0 \equiv 2d_1 \pmod{4}$. The result of the division when stored in $(d_0, d_1)$ is $(d_0, d_1) \leftarrow ((-d_0 + 2\mu d_1)/4, -(\mu d_0 + 2d_1)/4)$.

A. Length of $\tau$NAF and Reduction Schemes

The length $(l)$ of a $\tau$NAF for an integer scalar $k$ is approximately $2\log_2 k$, which is almost double the length of the scalar. Length reduction schemes were initially proposed by Solinas in [9]. An integer scalar $k$ is first reduced to $k \pmod{\delta}$, where $\delta = 2^{m-1}$ and then a $\tau$NAF is generated from the reduced scalar. The maximum length of the generated $\tau$NAF is $m + a$ in $F_{2^m}$. The scalar reduction scheme involves multi-precision integer division. Solinas proposed another reduction scheme [9] where the scalar is partially reduced to avoid integer division at the cost of integer multiplication. Since multi-precision division and multiplication operations are complicated, hardware implementations are inefficient.

In [17], Brumley and Järvinen presented the lazy reduction algorithm, where the scalar $k$ is repeatedly divided by $\tau$ for $m$ number of times to get the following relation.

$$k = (d_0 + d_1 \tau) \tau^m + (b_0 + b_1 \tau)$$  

$$= (d_0 + d_1 \tau) (\tau^m - 1) + (b_0 + d_0) + (b_1 + d_1) \tau$$

The authors use $\gamma$ as the reduced scalar and show that the length of the $\tau$NAF generated from $\gamma$ is at most $m + 4$ in $F_{2^m}$. Since division by $\tau$ is a simple operation (Theorem 1), the algorithm is suitable for hardware.

In [18], Adikari, Dimitrov and Järvinen proposed an improvement of the lazy reduction which they call the double lazy reduction. In the double lazy reduction, the scalar is divided by $\tau^2$ for $(m-1)/2$ number of times. Finally one division by $\tau$ is performed to obtain the reduced scalar. The cycle count for the scalar reduction reduces to nearly half compared to the lazy reduction. The computational steps in the double lazy reduction are shown in Algorithm 1. We advise the readers of this paper to study the double lazy reduction algorithm from [18], as this will be helpful to understand the acceleration techniques proposed in our paper.

III. Improved Reduction Algorithm

In this section we propose optimization steps to reduce the number of long integer additions and subtractions in Algorithm 1. Throughout this discussion we consider $\mu = -1$. Similar optimizations for $\mu = 1$ are shown in the appendix.

A. Elimination of Long Subtractions for Nonzero Remainders

In line 6 of Algorithm 1, remainders $u_0$ and $u_1 \in \{0, 1\}$ are subtracted from $d_0$ and $d_1$. We observe that the subtractions are easy in some cases. For example, when $d_0 \equiv 1 \pmod{4}$ and $2d_1 \equiv 0 \pmod{4}$ (i.e. $u_0 = 1$ and $u_1 = 0$), the subtraction of $u_0$ from $d_0$ is equivalent to changing the least significant bit of $d_0$ from 1 to 0. Hence, in this case the long subtraction can be replaced by a bit alteration. However, when carry propagations are involved with long subtractions, alteration of few specific bits do not work as a replacement. For example, when $d_0 \equiv 3 \pmod{4}$ and $2d_1 \equiv 0 \pmod{4}$ (i.e. when $u_0 = 1$ and $u_1 = 1$), a long subtraction appears. Use of signed remainders $u_0$ and $u_1 \in \{0, \pm 1\}$ helps to some extent in eliminating the long subtractions of nonzero remainders for such cases. Table
I show how the signed remainders are generated during the reduction steps depending on the low bits of $d_0$ and $2d_1$. In the table, the $\%$ operator represents modulus reduction operation. Except Case 4, subtractions of the $u_0$ and $u_1$ from $d_0$ and $d_1$ involve no carry propagation and thus can be performed by altering the low order bits of $d_0$ and $d_1$.

For Case 4, if we perform the subtraction of $u_0 = -1$ in line 7 of Algorithm 1 instead of line 6 (i.e. we put $d_0 + 1$ in place of $d_0$), then we have the following observation.

$$(d_0, d_1) \leftarrow \left(-\frac{2d_1 + d_0 + 1}{2}, -\frac{2d_1 - (d_0 + 1)}{2}\right)$$

This is equivalent to taking carry/borrow inputs in the adder/subtractor circuits as shown below.

$$d_0 \leftarrow -\frac{2d_1 + d_0 + \text{(Carry input) = 1}}{4}$$

$$d_1 \leftarrow -\frac{2d_1 - d_0 - \text{(Borrow input) = 1}}{4}$$

From the above observations we conclude that the long integer subtractions of the nonzero remainders in Algorithm 1 can be eliminated using cheaper alternatives.

### Algorithm 1: Double Lazy Reduction

| Input: integer $k$ |
| Output: reduced scalar $\gamma$ |
| begin |
| for $i = 1$ to $(m - 1)/2$ do |
| $d_0 \leftarrow d_0 - u_0$, $d_1 \leftarrow d_1 - u_1$ |
| $(d_0, d_1) \leftarrow \left((-d_0 + 2d_1)/4, -(\mu d_0 + 2d_1)/4\right)$ |
| if $u_0 > 0$ then |
| $b_0 \leftarrow (b_0 + u_0 a_0 - 2u_1 a_1)$ |
| $b_1 \leftarrow (b_1 + u_0 a_1 + u_1 (a_0 + \mu a_1))$ |
| $(a_0, a_1) \leftarrow (-2(a_0 + \mu a_1), \mu a_0 - a_1)$ |
| end |
| if $d_0 \equiv 1$ (mod 2) then |
| $u \leftarrow 1$, $d_0 \leftarrow d_0 - 1$ |
| $(b_0, b_1) \leftarrow (b_0 + a_0, b_1 + a_1)$ |
| $(d_0, d_1) \leftarrow (\mu d_0/2 + d_1, -d_0/2)$ |
| $\gamma \leftarrow (b_0 + d_0, b_1 + d_1)$ |
| end |

### B. Elimination of Subtractions from Zero

In line 7 of Algorithm 1, a subtraction from zero is required for $d_0$ after computing $2d_1 + d_0$ (Eq. 2).

$$(d_0, d_1) \leftarrow \left(-\frac{2d_1 + d_0}{4}, -\frac{2d_1 - d_0}{4}\right)$$

We eliminate the subtraction from zero using the following scheme. Instead of Eq. 2, we compute Eq. 3.

$$(d_0, d_1) \leftarrow \left(-\frac{2d_1 + d_0}{4}, 2d_1 - d_0\right)$$

The results from Eq. 2 and 3 have opposite signs, but same magnitudes. So, when Eq. 2 and 3 are applied iteratively for an even number of times, the results of the equations are same. However for an odd number of iterations, the results have opposite signs.

We replace the computation in line 7 by Eq. 3. Thus $(d_0, d_1)$ has wrong sign after any odd number of iterations and correct sign after any even number of iterations of the for-loop. The loop iterates for $(m - 1)/2$ number of times. So, when $(m - 1)/2$ is odd, only one subtraction from zero is required at the end to make $(d_0, d_1)$ of correct sign. We can also compensate this subtraction in line 18 or 19 by performing subtractions in place of additions or vice-versa.

The same trick is applied to eliminate the subtractions from zero during the computation of $(a_0, a_1)$ in line 12 of Algorithm 1. Instead of computing Eq. 4

$$(a_0, a_1) \leftarrow (-2(a_0 - a_1), -(a_0 + a_1))$$

we compute Eq. 5.

$$(a_0, a_1) \leftarrow (2(a_0 - a_1), (a_0 + a_1))$$

After the completion of the for-loop, one subtraction from zero is required to correct the signs of $(a_0, a_1)$ when $(m - 1)/2$ is odd. This subtraction can be eliminated if we compute $(b_0, b_1) \leftarrow (b_0 - a_0, b_1 - a_1)$ in line 16 of Algorithm 1.

Since the remainders are generated by observing the low order bits of $d_0$ and $d_1$ (Table I), use of $(d_0, d_1)$ which has wrong sign should be justified for the correctness of the reduction algorithm. Let after any odd number of iterations of the for-loop in Algorithm 1, we have the pairs $(a_{i,0}, a_{i,1})$, $(b_{i,0}, b_{i,1})$ and $(d_{i,0}, d_{i,1})$. Since wrong sign is assigned to $(d_{i,0}, d_{i,1})$ after any odd number of iterations, we have the following relation.

$$k = -(d_{i,0} + \tau d_{i,1})\tau^{2i} + (b_{i,0} + \tau b_{i,1})$$

In the next iteration, remainders $u_0$ and $u_1$ are generated from $d_{i,0}$ and $d_{i,1}$ instead of the correct values $-d_{i,0}$ and $-d_{i,1}$. If $(d_{i+1,0}, d_{i+1,1})$ are the quotients, then the relation between $(d_{i,0}, d_{i,1})$ and $(d_{i+1,0}, d_{i+1,1})$ is shown below.

$$(d_{i,0} + \tau d_{i,1}) - (u_0 + \tau u_1) = -(d_{i+1,0} + \tau d_{i+1,1})\tau^2$$

After plugging in Eq. 7 in Eq. 6, we get the following equation.

$$k = (d_{i+1,0} + \tau d_{i+1,1})\tau^{2i+2} - (u_0 + \tau u_1)\tau^{2i} + (b_{i,0} + \tau b_{i,1})$$

In the above equation the actual remainders are $(-u_0, -u_1)$. Interestingly, after any odd number of iterations, wrong sign is also assigned to $(a_0, a_1)$. Since, both $(u_0, u_1)$ and $(a_0, a_1)$

| TABLE I |
| Signed remainders during reduction of scalar |

<table>
<thead>
<tr>
<th>Cases</th>
<th>$d_0/4$</th>
<th>$2d_1/4$</th>
<th>$u_0$</th>
<th>$u_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>


Algorithm 2: New Reduction Algorithm

Input: integer k
Output: reduced scalar γ
1 begin
2 (a0, a1) ← (1, 0), (b0, b1) ← (0, 0), (d0, d1) ← (k, 0)
3 /* Iterative divisions by τ² start here */
4 for i = 1 to (m − 1)/2 do
5    u ← (d0 − 2d1) mod 4
6    (u0, u1) ← Table 1
7    (d0, d1) ← AlterLowBits(d0, d1)
8    if Case 4 is True then
9        (B, C) ← (1, 1) /* Borrow and Carry Inputs */
10    end
11 else
12    (B, C) ← (0, 0)
13 end
14 (d0, d1) ← ((2d1 + d0 + C)/4, (2d1 − d0 − B)/4)
15 if u > 0 then
16    b0 ← (b0 + u0a0 − 2u1a1)
17    b1 ← (b1 + u0a1 + u1(a0 − a1))
18 end
19 (a0, a1) ← (2(a0 − a1), a0 + a1) ;
20 /* Iterative divisions by τ² finish here */
21 if d0 ≡ 1 (mod 2) then
22    d0 ← AlterLeastBit(d0)
23 (b0, b1) ← (b0 + a0, b1 + a1)
24 end
25 (d0, d1) ← ((2d1 − d0)/2, d0/2) /* Division by τ */
26 if 2d1/2 ≡ 0 (mod 2) then
27    γ ← (b0 + d0, b1 − d1)
28 end
29 else
30    γ ← (b0 − d0, b1 + d1)
31 end
32 end

are of same sign in any iteration, computation of (b0, b1) is insensitive to the wrong sign of operands. Thus, (b1+1,0, b1+1,1) is computed as shown below.

(b1+1,0, b1+1,1) ← (u0 + τu1)(a1,0 + τa1,1) + (b1,0 + τb1,1)

This justifies the correctness of the reduction during assignment of wrong sign to the variables (d0, d1) and (a0, a1).

Throughout this section we have discussed how the number of long integer additions/subtractions can be reduced during the scalar reduction steps. Our proposed improvements over the double lazy reduction are described in Algorithm 2. We see that only one addition or subtraction is performed during the computations of d0, d1, a0, a1 and b0 in every iteration. For b1, at most two additions/subtractions are performed per iteration. Thus, if implemented in hardware, critical path contains only one adder/subtractor circuit of width m + 1 bit. In the previous reduction architectures [17], [18], critical paths are through two cascaded adder and subtractor circuits of data width m + 1. Since integer adder and subtractor circuits have large delay, removal of such circuits from critical paths help improves frequency. In the next section, we further look into τNAF generation algorithm and discuss how long subtractions of nonzero remainders can be eliminated during the τNAF generation steps.

IV. IMPROVED DOUBLE DIGIT τNAF GENERATION

In [18] two consecutive τNAF digits are generated in a single step from the reduced scalar d0 + τd1 by performing divisions by τ². The authors call the NAF as double τNAF. Table II describes the generation of the consecutive τNAF digits r0 and r1 from d0 and d1. Similar to Section III, we eliminate subtractions of nonzero remainders from d0 and d1 during the τNAF generation.

From Table II, we see that for the cases 2, 3.B, 3.C, 3.D, 5.A, 5.B, 5.D, 6 and 8, the subtractions of nonzero remainders from d0 or d1 affect only the low order bits of d0 and d1. For the above cases, the long subtractions are replaced by cheaper bit alterations in d0 and d1. Subtraction of r0 = −1 in Case 4 in Table II can be handled in the same way we did for Case 4 in Table I (Section III-A).

In Case 3.A, the subtraction of r1 = 1 from d1 involves borrow propagation and thus may affect all the bits of d1. If we incorporate this subtraction in the next step where we perform the division by τ², then by putting d1 − 1 in place of d1 in Eq. 2, we have the following observation.

\[
(d_0, d_1) \leftarrow (\frac{-2d_1 + (d_0 - 2) + 4}{4}, \frac{-2d_1 - (d_0 + 2)}{4})
\] (8)

We see that the subtraction of r1 from d1 is equivalent to the addition/subtraction of 2 with d0. Since d0 ≡ 2 (mod 8), the addition/subtraction changes only the three least bits of d0.

In Case 5.C, the subtraction of r1 = −1 from d1 involves carry propagation. When we put d1 + 1 in place of d1 in Eq. 2, we have the following observation.

\[
(d_0, d_1) \leftarrow (\frac{2d_1 - d_0}{4}, \frac{2d_1 + d_0}{4})
\] (9)

Using one’s complement of d1 in Case 5.C we eliminate the long subtraction of r1 from d1.

Algorithm 3 describes the new τNAF generation technique. Only one addition or subtraction is performed on d0 and d1.
during division by $\tau^2$ in any iteration. Thus, during the $\tau$NAF generation, presence of only one adder/subtractor circuit in the critical paths of $d_0$ and $d_1$ is sufficient.

**Algorithm 3:** New $\tau$NAF Generation Algorithm

**Input:** Reduced Scalar $\gamma = d_0 + \tau d_1$

**Output:** $\tau$NAF($\gamma$)

1. $S \leftarrow \langle \rangle$ /* Used to store $\tau$NAF */
2. $\text{Sign} \leftarrow 0$ /* Used to keep sign of $(d_0, d_1)$ */
3. while $d_0 \neq 0$ or $d_1 \neq 0$
   4. $(r_0, r_1) \leftarrow \text{Table 2}$
   5. $(d_0, d_1) \leftarrow \text{AlterLowBits}(d_0, d_1)$ in Section IV
   6. $(B, C) \leftarrow (0, 0)$ /* Borrow and Carry Inputs */
   7. if $\text{Sign} = 1$ then
      8. $(r_0, r_1) \leftarrow (-r_0, -r_1)$
   9. end
   10. $\text{Prepend}(r_1, r_0)$ to $S$ /* $\tau$NAF digits */
11. if Case 4 True then
   12. $(B, C) \leftarrow (1, 1)$
13. end
14. if Case 5.C True then
   15. $(d_0, d_1) \leftarrow (\frac{2d_1 + d_0 + C}{4}, \frac{2d_1 - d_0 - B}{4})$
   16. $\text{Sign} \leftarrow \text{Sign}$
17. end
18. else
   19. $(d_0, d_1) \leftarrow (\frac{2d_1 + d_0 + C}{4}, \frac{2d_1 - d_0 - B}{4})$
   20. $\text{Sign} \leftarrow 1 \oplus \text{Sign}$
21. end
22. end
23. end
24. end

V. HARDWARE ARCHITECTURE

The hardware architecture for performing scalar conversion (for $\mu = -1$) using the proposed acceleration techniques is shown in Fig. 1. Similar to [17], [18], our conversion architecture is capable of performing both scalar reduction and double digit $\tau$NAF generation.

In any iteration of Algorithm 2 and 3, the variables $a_0$, $a_1$, $b_0$, $b_1$, $d_0$ and $d_1$ have data dependencies on their values in the previous iteration. So, storage registers are used for each of the data variables. The data paths of the variables are kept in parallel to each other as there are no immediate data dependencies between them in any particular iteration.

The component Bit Alteration and NAF Generation is a combinational circuit which scans the low order three bits of $d_0$ and two bits of $d_1$. Remainder digits are generated as per Table I and II during the reduction and the $\tau$NAF generation phases of the scalar conversion. This component also subtracts the nonzero remainders from $d_0$ and $d_1$ using the bit alteration technique discussed in Section III and IV. An input signal `mode` is used to distinguish between the scalar reduction and the $\tau$NAF generation phases of the conversion. After the subtraction of $u_0$ from $d_0$ as bit alteration, two different outputs $d_0\#A1$ and $d_0\#A2$ are generated for the two different data paths through A1 and A2 adder/subtractor circuits. This happens because of Case 3.A in Table II, where 2 is subtracted and added for the two different data paths through A1 and A2 respectively.

During the $\tau$NAF generation phase, only the data paths for $d_0$ and $d_1$ remain active. An one bit register `sign` takes account of the sign of $d_0$ and $d_1$ (Algorithm 3). The remainders generated by the Bit Alteration and NAF Generation unit are first assigned the sign and then expressed as the two consecutive $\tau$NAF digits in each cycle.

A controller is used to generate the control signals for the multiplexers and the adder/subtractor circuits. The control block also generates the carry and borrow inputs for the
addition and subtraction operations. Let the data width for both
adders be at most \( m \). The adder and subtractor circuits are
split into two equal stages (Fig. 2) of width \( m/2 \) by putting
registers in the carry and borrow propagation paths.

Due to the data dependency of stage 2 on stage 1, the
computations in the stage 2 lag by one cycle. Timing diagram
of the two stages is described in Fig. 3 for the first five cycles.
Iteration numbers are indicated by the superscripts. As per
the timing diagram, first four iterations of the consecutive
additions and subtractions complete after the fifth cycle. It
is straightforward to understand that for \( I \) number of
iterations (Eq. 10), the two stage architecture takes \( I + 1 \) cycles.

In comparison, a non-pipelined architecture takes \( I \) number
of cycles to finish \( I \) rounds. The advantage of the pipelined
architecture is in the reduction of overall delay by half (ideally)
compared to the non-pipelined architecture at the cost of only
two flip-flops and one cycle.

**B. Pipelining the conversion architecture**

We apply the same concept in pipelining the conversion
hardware. However data dependencies get more complicated
due to the presence of shifter circuits and due to the different
data widths of the registers present in the conversion
architecture (Fig. 1). Additionally, synchronization between
the parallel data paths is essential to maintain functional
correctness of the conversion hardware.

Fig. 4 shows the two stage pipelined conversion architecture
for \( \mu = -1 \). Data paths are split in almost symmetric stages to
achieve best operating frequency for the two stage architecture.
In the figure, suffix \#1 and \#2 indicate the parts of different
components in the first and second stages of the pipelined
architecture respectively. The registers \( a_0, a_1, b_0, b_1, d_0 \)
and \( d_1 \) are split into two equal halves between the two stages.
The lower half of a register gets updated by the computations in
the first stage of the pipeline and the upper half gets updated by
the second stage of the pipeline. In the first stage of the data
path for \( d_0 \), the adder/subtractor A1\#1 has width \( \frac{m+1}{2} + 2 \) bits
due to the presence of division by four and two (right shift)
circuits. After a division by four, the most significant bit, i.e.,
the \((\frac{m+1}{2})^{th}\) bit of the output from A1\#1 is written into
the \((\frac{m+1}{2})^{th}\) bit, i.e. the most significant bit position of \( d_0 \).

To match the data width requirement for A1\#1, the bits from
position \( \frac{m+1}{2} + 1 \) and \( \frac{m+1}{2} + 2 \) of \( d_0 \) are needed. However, these
two bits belong to the upper half of \( d_0 \) register (\( d_0 \#2 \))
present in the second stage. As the second stage lags the first stage
by one cycle, we can not use the two least significant bits from
\( d_0 \#2 \). The output from the multiplexer M1\#2 leads by one
cycle over \( d_0 \#2 \) (because in any positive cycle transition, data
in the output of M1\#2 gets written into \( d_0 \#2 \)). We perform
data forwarding of the two least significant bits from the output
of the multiplexer M1\#2 to the input of A1\#1. Similar data
forwarding strategies are applied in the first stage of data path
for \( d_1 \) register due to the presence of division by two and four
circuits. Merging of wires are indicated by the horizontal and
vertical \( \sim \) symbols in Fig. 4.
Control signals for the adder/subtractor circuits and the multiplexers present in the second stage are lagged by one cycle to maintain the lag of the second stage in the pipelined data path. Data paths for $a_0$, $a_1$, $b_0$ and $b_1$ are also split in two stages to maintain synchronization between all parallel data paths present in the conversion hardware. The second stage of the data path for $b_0$ has data dependency on the bit position $\lceil \frac{m}{2} \rceil$ of register $a_1$. This particular bit is the most significant bit of the register $a_1$#1. Due to the lag of the second stage of register $b_0$, we apply data lagging of the required bit using an edge triggered flip-flop.

VII. EXPERIMENTAL RESULTS

We have evaluated the proposed acceleration techniques for the NIST recommended Koblitz curves [24] K-233 and K-283 on the Xilinx Virtex 4 FPGA xcvlx200-11ff1513. All these curves have $\mu = -1$ and support the present security standards. Table III shows performance results of the proposed pipelined and non-pipelined conversion architectures. Results are obtained from the Xilinx ISEv12.2 tool after place and route analysis with optimization for speed. Comparisons with other reported conversion architectures are also presented in the table. In [17], Brumley et. al. has provided experimental results for only Altera FPGAs. For fair comparison with the Brumley et. al. architecture, we have considered Xilinx FPGA based results of the same architecture from [18].

In the table, $T_{\text{Red}}$ and $T_{\text{Conv}}$ represents the scalar reduction

<table>
<thead>
<tr>
<th>Work</th>
<th>Curve</th>
<th>Slices</th>
<th>Freq MHz</th>
<th>$T_{\text{Red}}$ $\mu$s</th>
<th>$T_{\text{Conv}}$ $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brumley et. al.[18]</td>
<td>K-233</td>
<td>1380</td>
<td>76</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Adikari et. al.[18]</td>
<td></td>
<td>1777</td>
<td>75.3</td>
<td>1.55</td>
<td>3.1</td>
</tr>
<tr>
<td>Non-pipelined*</td>
<td></td>
<td>1661</td>
<td>81.5</td>
<td>1.4</td>
<td>2.8</td>
</tr>
<tr>
<td>2-Pipelined*</td>
<td></td>
<td>1582</td>
<td>119</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>Brumley et. al.[18]</td>
<td></td>
<td>1671</td>
<td>65.9</td>
<td>4.3</td>
<td>8.6</td>
</tr>
<tr>
<td>Adikari et. al.[18]</td>
<td></td>
<td>1998</td>
<td>65.1</td>
<td>2.2</td>
<td>4.4</td>
</tr>
<tr>
<td>Non-pipelined*</td>
<td>K-283</td>
<td>1910</td>
<td>70.2</td>
<td>2.0</td>
<td>4.1</td>
</tr>
<tr>
<td>2-Pipelined*</td>
<td></td>
<td>1814</td>
<td>107</td>
<td>1.3</td>
<td>2.6</td>
</tr>
<tr>
<td>3-Pipelined*</td>
<td></td>
<td>1812</td>
<td>122.6</td>
<td>1.16</td>
<td>2.32</td>
</tr>
</tbody>
</table>

![Fig. 4. Pipelined Conversion Architecture for $\mu = -1$](image-url)
time and the scalar conversion time respectively. The scalar conversion time is the sum of scalar reduction time and τNAF generation time. Our non-pipelined conversion architecture and the architecture in [18] have same cycle count of m + 6 for a scalar conversion in $F_{2^{m}}$. The pipelined conversion architecture takes only two extra cycles and thus requires $m + 8$ cycles in $F_{2^{m}}$. In [17], the conversion architecture uses division by $τ$ and takes $2m + 7$ cycles.

Operating frequencies achieved for the conversion architectures depend on the type of integer adders and also on the optimization settings in the synthesis tools. Our implementation uses generic adders and subtracters. For fare comparison of the frequencies, we have implemented a small circuit which is same as the data path for $d_0$ register in [18] and uses carry propagation subtracter circuits. With the same optimization parameters in the Xilinx ISE tool, we achieved frequencies 72MHz and 59.5MHz for the fields $F_{2^{233}}$ and $F_{2^{283}}$ respectively. When we consider implementation of the conversion hardware in [18] using generic adder and subtracter circuits, the frequencies will be limited by the above mentioned values due to the increased circuit complexities. Under this fare comparison scenario, our non-pipelined architectures achieve improvement in frequencies by at least 12.5% and 17.9% for K-233 and K-283 respectively. Area of the non-pipelined architectures are slightly lesser than the architectures in [18]. The proposed optimizations reduce the number of adder and subtracter circuits but increase the number of multiplexers.

Use of the pipeline strategy improves the frequency drastically. Since there are no bubbles in the pipeline data-path, the cycle requirements for scalar reduction and conversion remain almost same. Our two-stage pipeline architectures achieve 35.5% and 40% reduction in the overall conversion time compared to [18] for the curves K-233 and K-283 respectively. Further, to show that the proposed pipeline strategy is not limited to only two stages, we have implemented a three stage pipeline architecture for K-283. The three-stage architecture improves the frequency by around 14% compared to the two-stage architecture and thus reduces computation time.

An interesting observation is that the pipeline architectures have smaller area compared to the non-pipeline architectures. In one side, the pipeline strategy is very cost effective as it requires very few flip-flops. While on the other side, the ISE tool performs lesser logic replications due to shorter critical paths [25]. The combined effect reduces the overall area.

VIII. CONCLUSION

In this paper we have proposed acceleration techniques for scalar conversions required in the Koblitz curve based cryptoprocessors. The scalar conversion time is improved by reducing the number of costly integer additions and subtractions and by implementing pipelines in the data-path. The proposed reduction in the arithmetic cost simplifies the critical paths in the conversion architecture. Further, an efficient pipeline strategy is used to drastically improve the frequency of the conversion architecture without increasing the cycle count.

The circuit has inputs $w_0$, $u_1$, $k$ and $d_1$. The most significant bit of $d_0$ register is the output from the circuit. The critical path consists of two $m + 1$ bit subtracter circuits and one 4:1 multiplexer.

In this paper we have implemented up to three stage pipeline architecture. We remark that more number of pipeline stages can be implemented in the conversion architecture to achieve faster computation time. However the percentage of reduction in the computation time will lessen with the increase in the number of pipeline stages. The actual number of pipeline stages in the conversion architecture could be fixed to match the speed of the binary field primitives used in the Koblitz curve cryptoprocessor.

REFERENCES

Sujoy Sinha Roy is a PhD student in the Department of Electrical Engineering (ESAT), KU Leuven. He has a BS degree in Electronics and Telecommunication Engineering from BESU, Shibpur (2007) and a MS degree in Computer Science from Indian Institute of Technology, Kharagpur (2012). His research area has been broadly in the field of hardware for public key cryptography.

Junfeng Fan received his Bachelor and Master degrees in electrical engineering from Zhejiang University, China, in 2003 and 2006, respectively. In 2012, he received his PhD degree at the Department of Electrical Engineering (ESAT) of K.U. Leuven. His research interests include efficient arithmetic for public key cryptography, low-power design for ubiquitous security and physical attack resistant implementations.

Ingrid Verbauwhede received the electrical engineering degree and PhD degree from the KU Leuven, Belgium, in 1991. From 1992 to 1994, she was a postdoctoral researcher and visiting lecturer at the University of California, Berkeley. From 1994 to 1998, she worked for TCSI and ATMEL in Berkeley, California. In 1998, she joined the faculty of University of California, Los Angeles (UCLA). She is currently a professor at the KU Leuven and an adjunct professor at UCLA. At KU Leuven, she is a co-director of the Computer Security and Industrial Cryptography (COSIC) Laboratory. Her research interests include circuits, processor architectures and design methodologies for real-time embedded systems for security and cryptography. This includes the influence of new technologies and new circuit solutions on the design of next-generation systems on chip. She is a Fellow of the IEEE.

APPENDIX A

Here we present computational optimizations for the curve parameter \( \mu = 1 \). We compute \( (d_0, d_1) \) as per Equation (11) to avoid long subtractions from zero (Section III-B).

\[
(d_0, d_1) \leftarrow \left( \frac{2d_1 + d_0}{4}, \frac{2d_1 - d_0}{4} \right)
\]  

(11)

During the iterative divisions by \( \tau^2 \), wrong sign is assigned to either \( d_0 \) or \( d_1 \) in any iteration. Assignment of the wrong sign alternates in every consecutive iteration. We find Equation 11 is same as Equation 3, only with the difference in the relative positions of \( d_0 \) and \( d_1 \) in the left-hand-side. During the reduction of the scalar, the nonzero remainders are generated as per Table I for both \( \mu = 1 \) and \( \mu = -1 \). Thus, the computational optimizations we followed in Section III for \( \mu = -1 \), are also applicable for \( \mu = 1 \).

Double digit \( \tau \) NAF is generated as per Table VIII for \( \mu = 1 \). Comparing with Table II, we see only the cases 3.A-3.D are different in Table VIII. For the cases which are same in both the tables, we apply the same computational optimizations discussed in Section IV for \( \mu = -1 \). Subtractions of remainders from \( d_1 \) are performed by altering low order bits of \( d_1 \) for the cases 3.A, 3.C and 3.D. However the subtraction of remainder in case 3.B involves carry propagation. We eliminate this long subtraction by incorporating it in the next step where we perform division by \( \tau^2 \). This is shown in Equation 12. Subtraction of 2 or addition of 1 with \( d_0 \) is easy as it requires only alteration of low order bits of \( d_0 \). We also consider a borrow input to the adder/subtractor circuit in the critical path of \( d_1 \) (Equation 12).

\[
(d_0, d_1) \leftarrow \left( \frac{2d_1 + (d_0 - 2)}{4}, \frac{2d_1 - (d_0 + 1)}{4} \right)
\]  

(12)

### APPENDIX TABLE IV

<table>
<thead>
<tr>
<th>Cases</th>
<th>( d_0%4 )</th>
<th>( 2d_1%4 )</th>
<th>( d_0%4 )</th>
<th>( 2d_1%8 )</th>
<th>( r_0 )</th>
<th>( r_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3.A</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>3.B</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3.C</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3.D</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>5.A</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5.B</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>5.C</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>5.D</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### APPENDIX A

Here we present computational optimizations for the curve parameter \( \mu = 1 \). We compute \( (d_0, d_1) \) as per Equation (11) to avoid long subtractions from zero (Section III-B).

\[
(d_0, d_1) \leftarrow \left( \frac{2d_1 + d_0}{4}, \frac{2d_1 - d_0}{4} \right)
\]  

During the iterative divisions by \( \tau^2 \), wrong sign is assigned to either \( d_0 \) or \( d_1 \) in any iteration. Assignment of the wrong sign alternates in every consecutive iteration. We find Equation 11 is same as Equation 3, only with the difference in the relative positions of \( d_0 \) and \( d_1 \) in the left-hand-side. During the reduction of the scalar, the nonzero remainders are generated as per Table I for both \( \mu = 1 \) and \( \mu = -1 \). Thus, the computational optimizations we followed in Section III for \( \mu = -1 \), are also applicable for \( \mu = 1 \).

Double digit \( \tau \) NAF is generated as per Table VIII for \( \mu = 1 \). Comparing with Table II, we see only the cases 3.A-3.D are different in Table VIII. For the cases which are same in both the tables, we apply the same computational optimizations discussed in Section IV for \( \mu = -1 \). Subtractions of remainders from \( d_1 \) are performed by altering low order bits of \( d_1 \) for the cases 3.A, 3.C and 3.D. However the subtraction of remainder in case 3.B involves carry propagation. We eliminate this long subtraction by incorporating it in the next step where we perform division by \( \tau^2 \). This is shown in Equation 12. Subtraction of 2 or addition of 1 with \( d_0 \) is easy as it requires only alteration of low order bits of \( d_0 \). We also consider a borrow input to the adder/subtractor circuit in the critical path of \( d_1 \) (Equation 12).

\[
(d_0, d_1) \leftarrow \left( \frac{2d_1 + (d_0 - 2)}{4}, \frac{2d_1 - (d_0 + 1)}{4} \right)
\]  

(12)