An Area Aware Accelerator for Elliptic Curve Point Multiplication

Malik Imran
Dept. of Computer Systems
Tallinn University of Technology (TalTech)
Tallinn, Estonia
malik.imran@taltech.ee

Samuel Pagliarini
Dept. of Computer Systems
Tallinn University of Technology (TalTech)
Tallinn, Estonia
samuel.pagliarini@taltech.ee

Muhammad Rashid
Dept. of Computer Engineering
Umm Al-Qurra University (UQU)
Makkah, Saudi Arabia
mfelahi@uqu.edu.sa

***Abstract*— This work presents a hardware accelerator, for the optimization of latency and area at the same time, to improve the performance of point multiplication process in Elliptic Curve Cryptography. In order to reduce the overall computation time in the proposed 2-stage pipelined architecture, a rescheduling of point addition and point doubling instructions is performed along with an efficient use of required memory locations. Furthermore, a 41-bit multiplier is also proposed. Consequently, the FPGA and ASIC implementation results have been provided. The performance comparison with state-of-the-art implementations, in terms of latency and area, proves the significance of the proposed accelerator.**

Keywords*—elliptic curve cryptography, point multiplication, Montgomery algorithm, FPGA, ASIC*

I. INTRODUCTION

The main advantages of Elliptic Curve Cryptography (ECC), as compared to the commonplace Rivest Shamir Adleman algorithm, are shorter key lengths, lesser power consumption, and the lower hardware cost for an equivalent security [1]–[2]. Therefore, the ECC is implemented frequently in software or hardware [3]. While the software implementation is convenient and flexible, the hardware-based solutions display higher throughputs, low latency and inherent security [4]–[16]. However, the low latency hardware solutions typically demand precious hardware resources (area) that are not always available. Consequently, the architectures providing low latency with minimum area are challenging to devise.

A. Basis Parameters of ECC

The most important operation in ECC is the point multiplication (PM) [2]. Two types of fields are generally utilized to implement PM: the prime field, i.e., $GF(p)$ and the binary extension field, i.e., $GF(2^m)$. For each of the aforementioned fields, the National Institute of Standards and Technology (NIST) has provided various key length recommendations [17]. Furthermore, the implementations can adopt simple affine or projective coordinates [5]. In addition to the selection of field coordinates, another important issue in ECC is to choose between polynomial and normal basis [10].

The binary field is generally preferred over the primary filed for hardware implementations [4, 5]. In addition to this, the projective coordinates are more suitable than the general affine coordinates to attain effective latency/area architectures [3]. Similarly, the normal basis is valuable where the recurrent squarings are commonly required to be computed, while the polynomial basis is more convenient where the frequent multiplications are involved [2]. Consequently, in this paper, we have selected the binary field along with the projective coordinates to achieve some efficient latency/area results, whereas the polynomial basis has been selected to execute the finite field (FF) multiplications efficiently.

B. Related Work

Techniques toward latency optimization. In order to minimize the latency of ECC, different FF multipliers have been implemented. However, the three most frequently used multipliers are: digit level multipliers [4–8, 12, 15, 16], bit parallel multipliers [3, 9] and parallel Karatsuba multiplier [10, 11, 13, 14]. For each FF multiplication, the digit level multipliers require, $D = \frac{m}{n}$ clock cycles, where 'D' determines the total number of digits, 'm' defines the length of key and 'n' is the size of digit. There are two possibilities to implement the digit level multipliers, either by using the digit serial, implemented in [4–8], or digit parallel multiplier, utilized in [12, 16]. The digit serial multipliers require 'D' clock cycles for one FF multiplication while the digit parallel multipliers utilize one clock cycle for one FF multiplication, albeit with the higher requirements of required resources [12]. For both digit serial and digit parallel multipliers, the larger digit sizes are useful to reduce the number of clock cycles while the smaller digit sizes are more convenient to reduce the critical path [12, 15]. Therefore, to perform FF multiplication in one clock cycle, the bit parallel, implemented in [9], and Karatsuba multipliers, utilized in [10, 11, 13, 14] have been employed. These single-cycle multipliers have long critical paths that compromise the latency of the overall ECC computation. Apart from the FF multipliers, additional techniques related to the latency optimizations are pipelining [4, 5, 10, 12, 16] and instruction level parallelism [7].

Techniques toward area optimization. Several techniques have been adopted to reduce the hardware resources: the execution of Itoh Tsujii inversion algorithm by sharing the hardware resources of squarer and multiplier components [7, 9, 12, 16], the use of fewer temporary storage elements to keep the intermediate results during the computation of PM [12, 14, 16], the use of a single FF adder, multiplier and squarer components in the arithmetic and logic unit (ALU) of the crypto processor [12] and the use of digit serial multipliers [4–8].

C. Limitations of Existing Works

Section 1-B shows that the existing hardware accelerators for the PM process either target the optimization of latency or area [4, 6-9, 11, 13, 14, 16]. However, there are some application scenarios where the optimization of latency and area at the same time is critical [12]. Although, there exist some solutions which consider the optimization of both at the same time [5, 10, 12, 15], we believe that the performance can be further improved by using some efficient architectural techniques. Furthermore, another key issue in modern cryptographic applications is the scalability such that the solution can be configured for various key lengths, depending upon a particular requirement [16].

D. Proposed Solution

We have proposed an area aware latency optimized hardware accelerator architecture with $m = 163, 233, 283, 409$ and 571 for the PM computation of ECC. To reduce the latency of the proposed accelerator, we have performed:

- an exploration for different stages of pipelining
- an efficient rescheduling of PM instructions

Towards area reduction, we have proposed:

- an efficient use of required memory locations
- a digit parallel least significant digit level (DP-LSD) multiplier, with the digit size of 41 $bits$.

The proposed accelerator/architecture is described in Verilog HDL and synthesized for a Virtex-7 FPGA as well as for a 16nm ASIC technology. On FPGA, the results after synthesis reveal that the proposed accelerator can operate up to maximum frequencies of 383, 379, 377, 342 and 340 MHz when implemented with $m = 163, 233, 283, 409$ and 571 bit key lengths, respectively. As ASIC, a maximum operational frequency of 2.2GHz for $m = 571$ bit key length is achieved, and it requires only 5.5µs for one PM computation.

The performance in terms of ratio of 1 over latency times area is provided on FPGA (Virtex 7) and ASIC (16nm) platforms. As compared to the most recent state-of-the-art solution, the proposed accelerator achieves 36%, 65% and 57% higher performance ratio for $m = 163$, 233 and 283. Moreover, as expected, the ECC computation in the ASIC implementation takes 6.53 times less PM time than in the FPGA implementation.

The remainder of this paper is structured as follows: Theoretical background for the computation of PM on ECC over $GF(2^m)$ is provided in Section II. The proposed accelerator architecture is described in Section III. The implementation results and the comparison with state-of the-art are discussed in Section IV. Finally, the conclusions are given in Section V.

II. Point Mutiplication on ECC over GF(2^M)

For $GF(2^m)$, a Lopez Dahab projective form of elliptic curve is described as a set of points $P(X:Y:Z)$, satisfying the following Eq. (1):

$$E: Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \quad (1)$$

As presented in Eq. (1), the terms 'X', 'Y' and 'Z' are the Lopez Dahab projective components of initial point $P(X:Y:Z)$ where $Z \neq 0$, 'a' and 'b' are the curve constants, with $b \neq 0$. Then, PM is the addition of 'k' copies of point 'P', i.e., $Q = k \times (P + P + \cdots + P)$, where '$P$' is an initial point with 'x' and 'y' coordinates, 'k' is an integer (equals to the size of field) and 'Q' is a final point on the elliptic curve. Due to its simplicity, several algorithms are available for the computation of PM. We have selected the Montgomery algorithm [18] due to its inherent resilience against side channel attacks [5, 9, 16].

Algorithm 1: Montgomery algorithm over $GF(2^m)$ [18]

Input: $k = (k_{n-1}, \ldots, k_1, k_0)$ with $k_{n-1} = 1$, $P = (x_p, y_p) \in GF(2^m)$

Output: $Q(x_q, y_q) = k.P$

1. **Initialization:** $X_1 = x_p, Z_1 = 1, X_2 = x_p^4 + b, Z_2 = x_p^2$
2. **Point Multiplication:** for (i from $n-2$ down to 0) do

2.1. $if(k_i = 1)$, *then*	2.2. *else*
2.1.1. ***(P = P + Q)***	2.2.1. ***(P = P + Q)***
2.1.1.1. $Z_1 = X_2 \times Z_1$	2.2.1.1. $Z_2 = X_1 \times Z_2$
2.1.1.2. $X_1 = X_1 \times Z_2$	2.2.1.2. $X_2 = X_2 \times Z_1$
2.1.1.3. $T = X_1 + Z_1$	2.2.1.3. $T = X_2 + Z_2$
2.1.1.4. $X_1 = X_1 \times Z_1$	2.2.1.4. $X_2 = X_2 \times Z_2$
2.1.1.5. $Z_1 = T^2$	2.2.1.5. $Z_2 = T^2$
2.1.1.6. $T = x_p \times Z1$	2.2.1.6. $T = x_p \times Z_2$
2.1.1.7. $X_1 = X_1 + T$	2.2.1.7. $X_2 = X_2 + T$
2.1.1.8. $Return: P(X_1, Z_1)$	2.2.1.8. $Return: P(X_2, Z_2)$
2.1.2. ***(P = P + P = 2P)***	2.2.2. ***(P = P + P = 2P)***
2.1.2.1. $Z_2 = Z_2^2$	2.2.2.1. $Z_1 = Z_1^2$
2.1.2.2. $T = Z_2^2$	2.2.2.2. $T = Z_1^2$
2.1.2.3. $T = b \times T$	2.2.2.3. $T = b \times T$
2.1.2.4. $X_2 = X_2^2$	2.2.2.4. $X_1 = X_1^2$
2.1.2.5. $Z_2 = X_2 \times Z_2$	2.2.2.5. $Z_1 = X_1 \times Z_1$
2.1.2.6. $X_2 = X_2^2$	2.2.2.6. $X_1 = X_1^2$
2.1.2.7. $X_2 = X_2 + T$	2.2.2.7. $X_1 = X_1 + T$
2.1.2.8. $Return: Q(X_2, Z_2)$	2.2.2.8. $Return: Q(X_1, Z_1)$

end if

end for

3. **Reconversion:**

3.1. $x_q = \frac{X_1}{Z_1}$ and

3.2. $y_q = (x_p + \frac{X_1}{Z_1}) \times \left\{ [(X_1 + x_p \times Z_1)(X_2 + x_p \times Z_2) + (x_p^2 + y)(Z_1 \times Z_2)] \times (x_p \times Z_1 \times Z_2)^{-1} \right\} + y_p$

As shown in Algorithm 1, the Montgomery algorithm consists of three steps for the computation of PM. A scalar multiplier 'k' and an initial point 'P' with its coordinates (x_p, y_p) are the inputs. The coordinates (x_q, y_q) of the final point 'Q' are the outputs. Initialization step of Algorithm 1 ensures the conversions from affine to Lopez Dahab projective form, whereas point multiplication step calculates point addition (PA) and point doubling (PD) considering the inspected value of the scalar multiplier k_i. At the end, the reconversion step reverts the Lopez Dahab projective to affine conversion.

III. Architecture of Proposed Accelerator

The proposed 2-stage pipelined accelerator architecture is composed of a register file (RF), an arithmetic and logic unit (ALU), pipeline registers and an FSM-based control unit (CU), as presented in Fig. 1. The parameters for the architecture have been chosen from NIST [17].

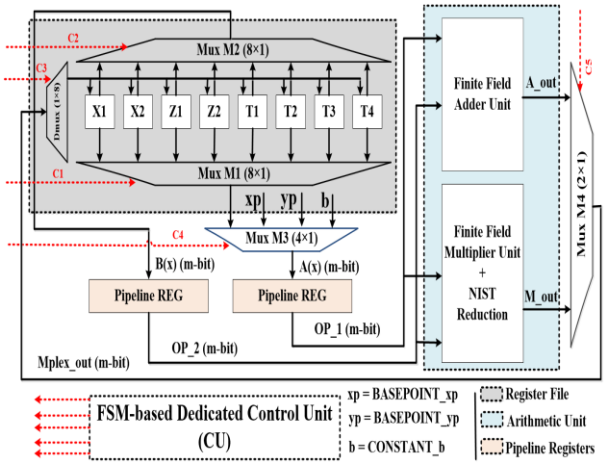

Fig. 1: Proposed hardware accelerator architecture

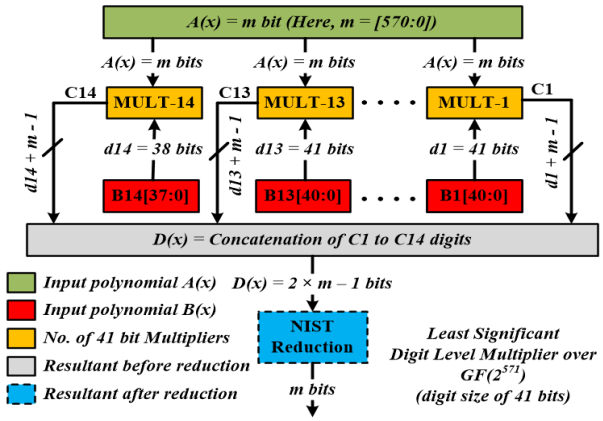

Fig. 2: Proposed digit parallel least significant digit (DP-LSD) multiplier with digit size of 41-bits

A. Register File

It contains an '$8 \times m$' register array, as illustrated in Fig. 1. The numeric digit '8' determines the total number of locations while variable 'm' determines the range of each particular location. In this paper, 'm' is the ECC key length, i.e., 163, 233, 283, 409 and 571. The purpose of RF is to keep intermediate results X_1, X_2, Z_1, Z_2, T_1, T_2, T_3 and T_4 of Algorithm 1. The multiplexers $M1$ (8×1) and $M2$ (8×1) are used to fetch the operands from RF to the ALU while a single de-multiplexer $Dmux$ (1×8) updates the contents of the RF by using the $Mplex\_out$ signal.

B. Arithmetic and Logic Unit

The ALU of the proposed hardware accelerator contains a total of two finite field operators, i.e., an adder and a multiplier, as presented in Fig. 1. Input to both operators are two 'm' bit polynomials $A(x)$ and $B(x)$. Each operator produces a single output polynomial, i.e., $A\_out(x)$ or $M\_out(x)$. The first input to both operators is polynomial $A(x)$, an output of routing multiplexer $M3$, while the second input is polynomial $B(x)$, an output of RF. Both output polynomials are inputted to mux $M4$ for write back on the RF. In order to compute the addition of polynomials $A(x)$ and $B(x)$ over $GF(2^m)$, bitwise exclusive-OR gates have been utilized.

Proposed multiplier architecture. For the multiplication of two 'm' bit polynomials, a DP-LSD multiplier with a digit size of 41 bits is utilized as depicted in Fig. 2. The digits with '$d = 41$' bits of input polynomial $B(x)$ are created $(B1-B14)$ by generating simple partial products. Parallel execution of each created digit $(B1-B14)$ for the polynomial multiplication is then performed with the input polynomial $A(x)$. For the computation of finite field multiplication over $GF(2^{571})$, a total of 14 digits are required, as shown in Fig. 2. Out of these 14 digits, 13 digits $(B1-B13)$ are with size of 41 bits, whereas the remaining one digit $(B14)$ is with 38 bit size. The parallel multiplication of each '$B1-B14$' digit with an 'm' bit polynomial $A(x)$ results '$d + m - 1$' bits of polynomials and these resultant polynomials are represented as '$C1-C14$'. After multiplication of each 'd' bit digit with an 'm' bit polynomial, the resultant polynomial of $D(x) = 2 \times m - 1$ bit is constructed using shift and add operation of '$C1-C14$'.

We have provided identical inputs to the DP-LSD multiplier for the computation of FF squaring, i.e., $A(x)^2 = A(x) \times A(x)$, which is required for the implementation of Algorithm 1. Both multiplication of two polynomials $(A(x) \times B(x))$ and squaring of one polynomial $(A(x)^2 = A(x) \times A(x))$, produce resultant polynomial of degree almost '$2 \times \mathrm{m} - 1$' bits. Consequently, reduction is essential to execute after the computation of each FF multiplication and squaring. For this purpose, NIST reduction algorithms over $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$ are implemented as described in Algorithms 2.41, 2.42, 2.43, 2.44 and 2.45 of [3]. Finally, to perform inversion over $GF(2^m)$, a square Itoh Tsujii algorithm [19] is implemented by repeatedly squaring along with FF multiplication operation, using only the multiplier unit.

C. Pipeline Registers and Instructions Rescheduling

To minimize the latency and to shorten the critical path, we have first sought an appropriate strategy for the deployment of pipeline registers. Subsequently, the architecture is divided into three portions: operands read $[R]$ using multiplexers $M1$, $M2$ and $M3$, execution of operands $[E]$ using the ALU and write back $[WB]$ of results into the RF unit.

There are three possible solutions with the aforementioned division. First, without the pipeline registers; hence, read $[R]$, execute $[E]$ and write back $[WB]$ in one clock cycle (CC). Second, 2-stage pipelined architecture (ALU containing the registers at their inputs), executing $[R]$ in first clock cycle and $[E, WB]$ in second cycle. Lastly, 3-stage pipelined processor (ALU accommodating registers both at the input and output), thus executing $[R]$, $[E]$ and $[WB]$ in three clock cycles.

For each PA and PD computation, Algorithm 1 in its actual structure requires a total of 14 instructions ($inst_1$ to $inst_{14}$), as shown in column 2 of Table I. Out of these 14 instructions, seven instructions are for PA ($inst_1$ to $inst_7$) and remaining instructions are for PD. However, Algorithm 1 in its presented form, can present read after write (RAW) hazards, as detailed in column 3 of Table I. The term hazard means that whenever one of the read operands of current instruction are dependent upon the write operand of the previous instruction. In such cases, the execution of current instruction ($inst_i$) is stalled until the result of previous instruction ($inst_{i-1}$) is written back. In order to

TABLE I. PROPOSED INSTRUCTION SCHEDULING USING SINGLE FINITE FIELD ADDER AND MULTIPLIER UNIT

Clock cycles	Original sequence of instructions ($inst_i$) for if part of Algorithm 1	RAW Hazard	Proposed rescheduling of instructions ($inst_i$) for if part of Algorithm 1 (for non-pipelined) $inst_i[R,E,WB]$	RAW Hazard	Status of proposed rescheduled instructions ($inst_i$) in pipelining for if part of Algorithm 1 (for 2-stage pipelined) $inst_i[R]$	(for 2-stage pipelined) $inst_i[E,WB]$	(for 3-stage pipelined) $inst_i[R]$	(for 3-stage pipelined) $inst_i[E]$	(for 3-stage pipelined) $inst_i[WB]$
1	$inst_1 \rightarrow Z_1 = X_2 \times Z_1$	–	$inst_1 \rightarrow Z_1 = X_2 \times Z_1$	–	$\text{inst}_1[R]$	–	$\text{inst}_1[R]$	–	–
2	$inst_2 \rightarrow X_1 = X_1 \times Z_2$	–	$inst_2 \rightarrow X_1 = X_1 \times Z_2$	–	$\text{inst}_2[R]$	$\text{inst}_1[E,WB]$	$inst_2[R]$	$inst_1[E]$	–
3	$inst_3 \rightarrow T = X_1 + Z_1$	X_1	$inst_{11} \rightarrow X_2 = X_2 \times X_2$	–	$\text{inst}_{11}[R]$	$inst_2[E,WB]$	$inst_{11}[R]$	$inst_2[E]$	$inst_1[WB]$
4	$inst_4 \rightarrow X_1 = X_1 \times Z_1$	–	$inst_3 \rightarrow T_1 = X_1 + Z_1$	–	$\text{inst}_3[R]$	$\text{inst}_{11}[E,WB]$	–	$inst_{11}[E]$	$inst_2[WB]$
5	$inst_5 \rightarrow Z_1 = T \times T$	–	$inst_4 \rightarrow X_1 = X_1 \times Z_1$	–	$\text{inst}_4[R]$	$\text{inst}_3[E,WB]$	$inst_3[R]$	–	$inst_{11}[WB]$
6	$inst_6 \rightarrow T = x_p \times Z_1$	Z_1	$inst_5 \rightarrow Z_1 = T_1 \times T_1$	–	$\text{inst}_5[R]$	$\text{inst}_4[E,WB]$	$inst_4[R]$	$inst_3[E]$	–
7	$inst_7 \rightarrow X_1 = X_1 + T$	T	$inst_8 \rightarrow Z_2 = Z_2 \times Z_2$	–	$\text{inst}_8[R]$	$\text{inst}_5[E,WB]$	–	$inst_4[E]$	$inst_3[WB]$
8	$inst_8 \rightarrow Z_2 = Z_2 \times Z_2$	–	$inst_6 \rightarrow T_1 = x_p \times Z_1$	–	$\text{inst}_6[R]$	$\text{inst}_8[E,WB]$	$inst_5[R]$	–	$inst_4[WB]$
9	$inst_9 \rightarrow T = Z_2 \times Z_2$	Z_2	$inst_9 \rightarrow T_2 = Z_2 \times Z_2$	–	$\text{inst}_9[R]$	$\text{inst}_6[E,WB]$	$inst_8[R]$	$inst_5[E]$	–
10	$inst_{10} \rightarrow T = b \times T$	T	$inst_7 \rightarrow X_1 = X_1 + T_1$	–	$\text{inst}_7[R]$	$\text{inst}_9[E,WB]$	–	$inst_8[E]$	$inst_5[WB]$
11	$inst_{11} \rightarrow X_2 = X_2 \times X_2$	–	$inst_{10} \rightarrow T_2 = b \times T_2$	–	$\text{inst}_{10}[R]$	$\text{inst}_7[E,WB]$	$inst_6[R]$	–	$inst_8[WB]$
12	$inst_{12} \rightarrow Z_2 = X_2 \times Z_2$	X_2	$inst_{12} \rightarrow Z_2 = X_2 \times Z_2$	–	$\text{inst}_{12}[R]$	$\text{inst}_{10}[E,WB]$	$inst_9[R]$	$inst_6[E]$	–
13	$inst_{13} \rightarrow X_2 = X_2 \times X_2$	–	$inst_{13} \rightarrow X_2 = X_2 \times X_2$	–	$\text{inst}_{13}[R]$	$\text{inst}_{12}[E,WB]$	–	$inst_9[E]$	$inst_6[WB]$
14	$inst_{14} \rightarrow X_2 = X_2 + T$	X_2	$inst_{14} \rightarrow X_2 = X_2 + T_2$	X_2	–	$inst_{13}[E,WB]$	$inst_7[R]$	–	$inst_9[WB]$
15	–	–	–	–	$\text{inst}_{14}[R]$	–	$inst_{10}[R]$	$inst_7[E]$	–
16	–	–	–	–	–	$\text{inst}_{14}[E,WB]$	$inst_{12}[R]$	$inst_{10}[E]$	$inst_7[WB]$
17	–	–	–	–	–	–	$inst_{13}[R]$	$inst_{12}[E]$	$inst_{10}[WB]$
18	–	–	–	–	–	–	–	$inst_{13}[E]$	$inst_{12}[WB]$
19	–	–	–	–	–	–	–	–	$inst_{13}[WB]$
20	–	–	–	–	–	–	$inst_{14}[R]$	–	–
21	–	–	–	–	–	–	–	$inst_{14}[E]$	–
22	–	–	–	–	–	–	–	–	$inst_{14}[WB]$

reduce RAW hazards, we reschedule the instructions of Algorithm 1. The proposed instruction rescheduling was achieved by considering the following:

- Parallel execution of PA and PD instructions for PM computation. For example, in the proposed rescheduling (shown in column 4 of Table I), $inst_3$ cannot be read until the result of $inst_2$ is available, resulting in one cycle delay. We have instead scheduled $inst_{11}$ when $inst_2$ is in the write back stage. In the next cycle, $inst_3$ is scheduled when $inst_{11}$ is in the write back stage.
- Efficient replacement of temporary storage element, denoted as 'T', shown in column 2 of Table I, with 'T_1' for PA instructions and 'T_2' for PD instructions, shown in column 4. This results in decrease of one clock cycle for one PA and PD computation. Therefore, for m bit key length, m number of clock cycles are reduced.

The rescheduled instructions for non-pipelined case ($[R], [E], [WB]$ in one CC) are presented in column 4 of Table I, resulting in a single RAW hazard, shown in column 5 of Table I (the original sequence of instruction had seven hazards, marked in column 3). The corresponding sequences carried out in 2- and 3-stage pipelining architectures are shown in the right hand side of Table I. For each PA and PD computation, the last instruction, i.e., $inst_{14}$ for non-pipelined, 2-stage, and 3-stage pipelined cases are written back in 14, 16 and 22 clock cycles, as shown in Table I. As described earlier, the proposed rescheduling has only one RAW hazard, resulting in 1 and 6 clock cycles delay when executed in the 2-stage and 3-stage pipelined architectures, respectively. For the 2-stage pipeline, this delay is offset by the higher frequency enabled by the pipelining technique. On the other hand, moving from 2- to 3-stage brings only a slight increase in clock frequency that is not beneficial. Consequently, in the remainder of this paper, we discuss only 2-stage pipeline case.

D. Control Unit

The proposed hardware accelerator contains an FSM-based control unit, described in the following:

In order to implement Algorithm 1, the initialization step requires only 6 clock cycles. The proposed rescheduling of PA and PD requires a total of 33 cycles. Out of these 33 cycles, 32 cycles are required for the computation of 'if' and '$else$' parts of Algorithm 1, while the remaining one cycle is required to inspect the key bit. Finally, the reconversion step requires two FF inversions (Inv) for an additional 34 cycles. Clock cycles for each Inv operation are computed by implementing $m-1$ times repeated squares followed with 9 (for $m = 163$), 10 (for $m = 233$), 10 (for $m = 283$), 11 (for $m = 409$) and 12 (for $m = 571$) FF multiplications. The total number of required cycles for the proposed accelerator can be calculated by using Eq. (2), as detailed in Table II. It becomes clear that the PM calculation takes most of the computation cycles.

$$\underbrace{\text{Initial}}_{\text{Initialization}} + \underbrace{17 \times (\text{m}-1)}_{\text{Point Multiplication}} + \underbrace{2 \times (\text{Inv}) + 34}_{\text{Reconversion}} \qquad (2)$$

TABLE II. CLOCK CYCLES INFORMATION

Field (m)	Initial	PA + PD $17 \times (m-1)$	Inv.	Recon.	Total cycles
163	6	2754	502	1038	3798
233	6	3944	709	1452	5402
283	6	4794	867	1768	6568
409	6	6936	1239	2512	9454
571	6	9690	1300	2633	12329

Initial = initialization, PA + PD = point additions & doublings, Inv = inversion, Recon=reconversion

IV. RESULTS AND COMPARISONS

This section describes the implementation results for the proposed accelerator architecture on ASIC and FPGA platforms (Section IV-A). The achieved results are then compared with most recent state-of-the-art solutions on FPGA (Section IV-B).

A. *Results after synthesis on FPGA and ASIC library*

We have created a Verilog HDL model for each field size over $GF(2^{163})$, $GF(2^{233})$, $GF(2^{283})$, $GF(2^{409})$ and $GF(2^{571})$ The HDL models are then synthesized for Virtex-7 (XC7VX690T) FPGA, using Xilinx ISE tool [20]. For ASIC implementation, the HDL model for $GF(2^{571})$ is synthesized targeting a 16nm FinFET technology, using Cadence Genus tool and a commercial cell library. The circuit was implemented with a nominal voltage of 0.8V and results are reported for the typical corner (TT). The clock uncertainty and clock tree delays were carefully modelled to achieve a tapeout-quality result. The circuit is fully routed and passes DRC with no violations. Metals 1 through 7 are used for signal routing, while the power is distributed in M8/M9. The actual layout of the accelerator, as shown in Fig. 3, is obtained from Cadence Innovus [21].

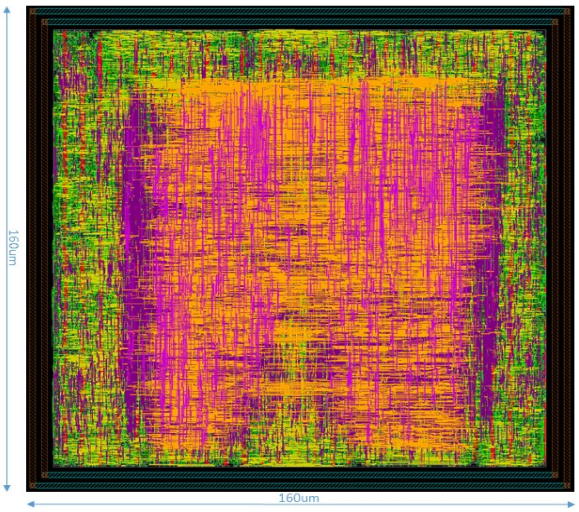

Fig. 3: Layout of the proposed accelerator for $m = 571$ bits key length

The synthesis results on ASIC and FPGA are given in Table III and Table IV, respectively. The numerical figures of achieved latency for the proposed hardware accelerator on both FPGA and ASIC are 36.26µs and 5.55µs over $GF(2^{571})$ respectively. Consequently, and as expected, the ECC computation in the ASIC implementation takes 6.53 times less time than in the FPGA implementation.

TABLE III. ARCHITECTUR RESULTS ON 16NM ASIC OVER GF(2^{571})

Clk Prd. (ps)	Fq. (MHz)	Inst	Area (µm²)	Latency (µs)	Power (nW)	
					Lkg	Dyn
450	2220	34281	22441	5.55	39039	30529403

Clk Prd: Clock Period, Inst: Instances, Lkg: leakage, Dyn: Dynamic

TABLE IV. RESULTS AND COMPARISON ON VIRTEX-7 FPGA

Ref year	m	Slices	LUTs	FFs	Fq. (MHz)	Latency (µs)
[12] 2019	163	2207	9965	1981	369	10.73
	233	5120	18953	2764	357	15.78
	283	5207	20202	3210	337	20.32
[9] 2018	163	3670	–	–	–	2.50
	233	5616	–	–	–	4.09
	283	7738	–	–	–	5.81
	409	12290	–	–	–	9.50
	571	20291	–	–	–	18.51
[16] 2018	163	3107	10955	1968	351	28.53
	233	5674	19753	2764	343	10.73
[15] 2015	163	1476	4721	1886	397	10.51
	233	2647	7895	2832	370	16.01
	283	3728	11593	3973	345	20.96
	409	6888	20881	6038	316	32.72
	571	12965	38547	10066	250	57.61
This work	163	1529	4162	1832	383	9.91
	233	2048	6407	2524	379	14.25
	283	2623	6753	3018	377	17.42
	409	3373	10083	4229	342	27.64
	571	4560	12691	5871	340	36.26

Comparing ASIC implementations with one another is challenging since technology nodes can be widely different, not only in transistor dimensions (i.e., technology node) but also in transistor structure. Our work utilizes a commercial "1st generation" FinFET technology, which cannot be fairly compared to older bulk technologies. Hence, we have selected only the most relevant FPGA-based solutions [10, 12, 15] for our comparison of results.

B. *Comparison with FPGA-based solutions*

Figure-of-Merit (FoM). To evaluate the performance of proposed accelerator and to perform a fair comparison with recent state-of-the-art solutions, we have defined a FoM (ratio of 1 over latency times slices) that captures both latency and area (slices) characteristics at the same time. The term latency is the time required for the computation of one PM $(k.P\ in\ \mu s)$ and is calculated by using Eq. (3). Therefore, the defined FoM is calculated by using Eq. (4) and the calculated values are given in Fig. 4.

$$latency = k.P(in\ \mu s) = \frac{\#\ of\ clock\ cycles}{clock\ frequency\ (MHz)} \quad (3)$$

$$\frac{1}{latency \times slices} = \frac{1}{k.p\ (in\ \mu s) \times slices} \quad (4)$$

Ccomparison using the individual latency and area: As far as only the latency is concerned, the proposed accelerator over $GF(2^{163})$ to $GF(2^{283})$ is 8%, 10% and 15% faster than the solution provided in [12]. The comparison is not possible for higher key lengths, as the authors in [12] do not target them. Similarly, for $GF(2^{163})$ to $GF(2^{571})$, this work provides 6%,

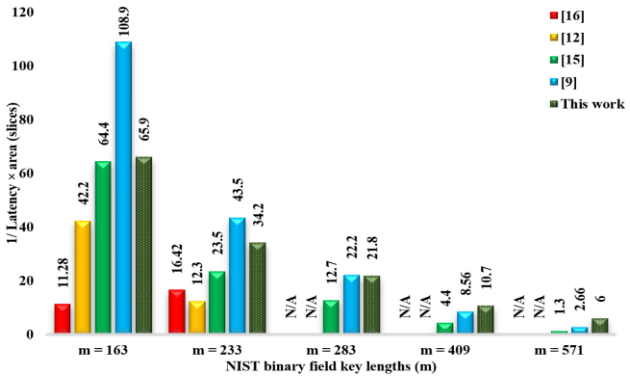

Fig. 4: Results of 1/latency×area for different key lengths

11%, 17%, 16% and 38% speedup with respect to [15]. However, the improvement in latency is 66% for $GF(2^{163})$, as compared to [16] while over $GF(2^{233})$ the solution described in [16] is 25% faster than the proposed accelerator. Finally, for $GF(2^{163})$ to $GF(2^{571})$, the latency achieved in [9] is 75%, 72%, 67%, 66% and 49% better than the proposed one.

On the other hand, when considering only the area for comparison, the work in [12] over $GF(2^{163})$ to $GF(2^{283})$ utilizes 69%, 40% and 50% more FPGA slices than this work. As compared to [15] over $GF(2^{163})$, the proposed accelerator consumes 4% more hardware slices while for the remaining field sizes, i.e., $GF(2^{233})$ to $GF(2^{571})$, the proposed accelerator utilizes 23%, 30%, 52% and 65% less slices. While the work in [9] shows better results in terms of latency for $GF(2^{163})$ to $GF(2^{571})$, the proposed accelerator utilizes 59%, 64%, 67%, 73% and 78% lower FPGA slices than [9]. Finally, as compared to [16], the proposed accelerator consumes 51% and 64% lower slices over $GF(2^{163})$ and $GF(2^{233})$.

Comparison using the defined FoM: As compared to [12], the proposed accelerator achieves 36%, 65% and 57% higher FoM values over $GF(2^{163})$ to $GF(2^{283})$. Similarly, for $GF(2^{163})$ to $GF(2^{571})$, the proposed accelerator provides 3%, 32%, 42%, 59% and 78% higher FoM values as compared to [15]. For $GF(2^{163})$ to $GF(2^{233})$, the proposed accelerator obtains 83% and 52% higher FoM values as compared to [16]. As compared to [9] over $GF(2^{163})$ to $GF(2^{283})$, the proposed accelerator achieves 39%, 21% and 2% lower values while for the remaining field sizes, i.e., $GF(2^{409})$ to $GF(2^{571})$, the it achieves 20% and 56% higher values as compared to [9]. These results strongly suggest that our accelerator is better suited for longer key lengths (for $m = 571$ bits).

V. CONCLUSIONS

This paper has presented an efficient 2-stage pipelined accelerator, in terms of latency and area, over $GF(2^{163})$ to $GF(2^{571})$. The accelerator uses an LSD based multiplier with digit size of 41bits to execute the FF multiplication in one clock cycle. Moreover, the pipeline registers are efficiently placed at the input of ALU to shorten the critical path(s). Furthermore, an efficient rescheduling of PA and PD computations is presented. Finally, the proposed accelerator provides a best-in-class figure-of-merit of 6 (for $m = 571$) as compared to state-of-the-art FPGA implementations. Our ASIC implementation pushes the performance envelope further, as it displays a speedup of 6.53 as compared to the same FPGA implementation.

ACKNOWLEDGEMENTS

This work was supported by the EC through the European Social Fund in the context of the project "ICT programme".

REFERENCES

[1] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Communication*, vol. 48, no. 177, pp. 203–209,1987.

[2] M. Rashid, et al., "Flexible architectures for cryptographic algorithms - A Systematic Literature Review, *Journal of Circuits, Systems and Computers*, vol. 28, no. 3, 2019.

[3] D. Hankerson, A. Menezes, and S. Vanstone, "Guide to elliptic curve cryptography," New York: Springer-Verlag, 2004.

[4] W. Chelton and M. Benaissa, "Fast elliptic curve cryptography on FPGA," *IEEE Trans. on VLSI System.*, vol. 16, no. 2, pp. 198–205, 2008.

[5] Z. Khan and M. Benaissa, "High speed and low latency ECC processor implementation over $GF(2^m)$ on FPGA," *IEEE Trans. on VLSI Systems*, vol. 25, pp. 165–176, 2017.

[6] B. Ansari and M. Hasan, "High-performance architecture of elliptic curve scalar multiplication," *IEEE Trans. on Computers.*, vol. 57, no. 11, pp. 1443–1453, 2008.

[7] Y. Zhang et al., "A high performance ECC hardware implementation with instruction-level parallelism over $GF(2^{163})$," *Microprocessor and Microsystems.*, vol. 34, no. 6, pp. 228–236, 2010.

[8] G. Sutter, J. Deschamps, and J. Imana, "Efficient elliptic curve point multiplication using digit serial binary field operations," *IEEE Trans on Industrial Electronics.*, vol. 60, no. 1, pp. 217–225, 2013.

[9] L. Li and S. Li, "High-performance pipelined architecture of point multiplication on koblitz curves," *IEEE Trans. on Circuits and Systems-II*, vol. 65, no. 11, pp. 1723–1727, 2018.

[10] R. Salarifard, S. Bayat-Sarmadi and H. Mosanaei-Boorani, "A low-latency and low-complexity point-multiplication in ECC," *IEEE Trans on Circuits and Systems-I*, vol. 65, no. 9, pp. 2869–2877, 2018.

[11] S. Roy, C. Rebeiro, and D. Mukhopadhyay, "Theoretical modeling of elliptic curve scalar multiplier on LUT-based FPGAs for area and speed," *IEEE Trans on VLSI System.*, vol. 21, no. 5, pp. 901–909, 2013.

[12] M. Imran et al., "Throughput/area optimised pipelined architecture for elliptic curve crypto processor," *IET Computers and Digital Techniques*, vol. 13, no. 5, pp. 361–368, 2019.

[13] G. Sutter, J. Deschamps, and J. Imana, "Efficient elliptic curve point multiplication using digit serial binary field operations," *IEEE Trans. on Industrial Electronics.*, vol. 60, no. 1, pp. 217–225, 2013.

[14] M. Imran, M. Kashif and M. Rashid "Hardware design and implementation of scalar multiplication in elliptic curve cryptography (ECC) over $GF(2^{163})$ on FPGA," *IEEE Int. Conf. on Information and Communication Technologies (ICICT)*, Pakistan, 2015, pp. 1–4.

[15] Z. Khan and M. Benaissa, "Throughput/area-efficient ECC processor using Montgomery point multiplication on FPGA," *IEEE Trans. on Circuits and Systems-II.*, vol. 62, no. 11, pp. 1078–1082, 2015.

[16] M. Imran et al., "ACryp-Proc: flexible asymmetric crypto processor for point multiplication," *IEEE Access*, vol. 6, pp. 22778–22793, 2018.

[17] NIST recommended elliptic curves for federal government use, July 1999. http://csrc.nist.gov/CryptoToolkit/dss/ecdsa/NISTReCur.pdf.

[18] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[19] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases," *Journal of Information and Computation*, vol. 78, no. 3, pp 171–177.

[20] Xilinx ISE Design Suite, [Online] Available at: https://www.xilinx.com/products/design-tools/ise-design-suite.html.

[21] Cadence, [online] Available at: https://www.cadence.com/en_US/home.html.