A Generic Approach to the Development of Coprocessors for Elliptic Curve Cryptosystems

Rabia Shahid, Ted Winograd and Kris Gaj
ECE Department
George Mason University
Fairfax, VA 20151
Email: {rshahid, twinogra, kgaj}@gmu.edu

***Abstract*—The mathematical complexity involved in ECC requires engineers to delve into advanced concepts related to algebra and number theory to achieve optimized designs. We present the design of a configurable and generic execution unit for ECC that serves as a coprocessor to perform operations involved during a scalar multiplication. The execution unit is supported by a software static scheduler to automate the cumbersome process of manual scheduling of operations involved in ECC. The arithmetic unit performs the operations at the lowest level of hierarchy, i.e., prime field arithmetic. We focus on optimizing the overall performance of the cryptoprocessor by using an optimal number of multiplier units, capable of taking full advantage of the parallelism present in the algorithm and a single modular adder/subtractor, working in parallel with multipliers. An instruction set architecture capable of supporting all required instructions is designed, along with the coprocessor that can process multiple batches of instructions using the arithmetic unit. We report results for an entire scalar multiplication in terms of latency in clock cycles and in absolute time units. We also demonstrate that the entire setup is generalizable to any cryptosystem that involves modular multiplications and modular additions/subtractions at the lowest level of hierarchy.**

I. Introduction and Motivation

Cryptographic engineering is a field that combines cryptology, algebraic geometry, and number theory with methods from digital system design using two most important technologies of today: ASICs and FPGAs. Unfortunately, most of the researchers working in this area are either mathematicians/cryptographers or hardware engineers, specializing in their respective fields. The theoretical complexities related to number theory and abstract algebra in any cryptosystem can easily make things a lot harder for a hardware engineer with background in only basic modular arithmetic required for most algorithms. Modern public key cryptosystems other than ECC, such as pairing-based schemes [5], [6], also rely heavily on these complex abstract algebra concepts. The theory behind ECC [14] is also quite rich and allows realization of various schemes to provide different cryptographic services. Therefore, a significant gap exists between cryptographers/mathematicians, who design these algorithms, and engineers, who implement them. For correct choice of parameters, one is required to have sufficient background in concepts related to Galois fields (mainly prime fields) and sometimes even extension fields. One level higher are operations on the group of points on an elliptic curve and elliptic curve discrete logarithm (ECDLP) problem.

The novelty of our approach is the development of a new methodology and tools that help to bridge this gap. Based on the platform and application, engineers can use our approach and tools to choose which set of parameters is optimal and determine how this set will perform in terms of timing and resource utilization. We have developed a configurable execution unit that serves as our coprocessor, to execute instructions grouped into batches by the scheduler. More generally, if applied to any other cryptosystem, where the main operations are composed of finite field arithmetic, our cryptoprocessor can serve as an engine to process a sequence of instruction batches, generated by the scheduler, by dispatching operations to the underlying field arithmetic units. The overall goal is to achieve fast computation time and minimized storage, combined with configurability of arithmetic units.

For ECC implementation, there are many possible design options, e.g., the choice of elliptic curve, use of different coordinate systems, the underlying finite field, etc. The time required for manual scheduling of basic underlying operations prevents hardware designers from exploring any significant subset of these design options. By following our design methodology, the user can generate a schedule of operations automatically for multiple design choices without any need to modify a universal and parametrizable execution unit. The scheduler also predicts the overall execution time in terms of clock cycles and gives an estimate of the memory required by the hardware architecture, which can help in early prototyping to come up with the most suitable set of options to be deployed in real applications.

II. Related Work

A lot of research has been done to develop hardware cryptoprocessors to speed up scalar multiplication operations on elliptic curves for a wide range of applications and platforms.

An improved design to perform scalar multiplication, capable of supporting different standards and curves by a scalable architecture, is presented in [2]. To exploit parallelism by using two multiplier units, manual scheduling of operations is performed in their design. In [3], although scheduling is performed through a program, the schedule is generated only for one multiplier unit, so there is no flexibility to perform independent operations at once to improve the overall throughput of the design. In [8], an Altera FPGA-based implementation is presented. To perform scheduling the same manual way is adopted, however they implement carry-free arithmetic using Residue Number System (RNS).

[20] presented a superscalar architecture for curve-based cryptosystems by using multiple processing cores. Since their cores can perform only operations specific to ECC/HECC, their coprocessor is not generalizable to any other cryptosystem composed of modular multiplications and modular additions/subtractions. In [21], a dual field ECC coprocessor is presented to show that the use of multiple processing elements can help improve the performance of their design. Guo et al. [9] developed a hardware coprocessor for ECC, with local storage and local controller, which performs finite field arithmetic by manual scheduling of operations. Ghosh et al. in [7], and Hamilton et al. in [10] present a hardware accelerator for ECC over prime fields for arbitrary size fields.

All above mentioned designs are implemented by generating manual sequence of operations to perform ECC. Any change in the design choices will require manual scheduling of the operations from scratch. Unlike all other related work available in the literature, adding the capability to generate an automated schedule along with a generic execution unit, allows us to implement any cryptosystem which is composed of modular multiplications and modular additions/subtractions.

The related work based on architectures to perform modular arithmetic also focuses towards decreasing the computation time to minimize latency of the entire operation. In 1985 Montgomery proposed modular multiplication without trial division. A novel number representation, and a novel basic arithmetic operation, were named the numbers in the Montgomery domain and the modular Montgomery multiplication, respectively. Multiple different, hardware-supporting, bit-oriented versions of this algorithm were analyzed in [15]. Tenca et al. [24] proposed the very first scalable architecture for Montgomery Multiplication. Harris et al. in [12], and later on Huang et al. in [13] have improved this design in terms of latency and latency*area by factor of two. Further improvement of the aforementioned architectures was possible when radix-4 architectures were introduced. They were demonstrated for the Tenca et al., Harris et al. and Huang et al. designs in [25], [11] and [13], respectively.

Suzuki in [22], [23] combined the Multiple Word Radix-2 Montgomery Multiplication (MWR2MM) together with the quotient pipelining technique and proposed an architecture which can be mapped onto a modern high-performance DSP-oriented FPGA structure. We use a Montgomery multiplication algorithm based on quotient pipelining technique developed by Orup in 1995 [18].

III. Mathematical Background on ECC

An elliptic curve E over GF(p) can be defined by the following equation in Weierstrass form:

$$y^2 = x^3 + ax + b$$

where a and b are curve parameters. The point on an elliptic curve can be expressed using affine coordinates or using projective coordinates. We use modified Jacobian coordinates which are a subclass of projective coordinates to allow fast point doubling. The use of projective coordinates over the traditional affine coordinates is also preferred to avoid modular inversions. As a result, a point on elliptic curve is described as $P = (X, Y, Z, aZ^4)$. The coordinates of the input point P are converted into projective coordinates at the beginning. Once the elliptic curve scalar multiplication is done, the resulting point is converted back to affine space at the end. There is only one inversion required for the conversion from projective to affine space. We perform the modular inversion using Fermat's little theorem implemented using square-and-multiply modular exponentiation.

To calculate the scalar multiplication we use the Double-and-Add Algorithm as described in Algorithm 1.

Algorithm 1 Elliptic Curve Point Multiplication

1: **Input**: P - point of an elliptic curve, k - integer k, $k = k_{l-1}, k_{l-2}, ...k_0$, $k_{l-1} = 1$
2: **Output**: Q - point of an elliptic curve
3: $Q \leftarrow P$
4: **for** i from $l-2$ downto 0 **do**
5: $\quad Q \leftarrow 2Q$
6: $\quad$ **if** $(k_i = 1)$ **then**
7: $\quad\quad Q \leftarrow Q + P$
8: $\quad$ **end if**
9: **end for**
10: **return** Q.

IV. Background on Modular arithmetic

A. Modular Multiplication

The Montgomery product MP computed as a result of Montgomery multiplication is in the form of $S = ABR^{-1} \pmod{M}$, where A and B are the multiplication arguments, M is the modulus, S is the final result, and $R = 2^n$, where n is equal to the number of bits of M.

In 1995 Orup proposed a quotient pipelining technique in [18]. His algorithm, similar to the Montgomery's counterpart, produces the final result of multiplication in the form of $S = ABR^{-1} \bmod M$, where A and B are the multiplication arguments, M is the modulus, S is the final result, and $R = 2^n$, where n is equal to the number of bits in M.

Algorithm 2 Modular Multiplication with Quotient Pipelining as described in [18]

1: **Setting**: radix : 2^k; delay parameter : d = 1; no. of blocks: n; multiplicand : A; multiplier : B; modulus: M, $M > 2$, gcd$(M, 2)$ = 1, $(-MM' \mod 2^{k(d+1)})$ = 1, $\tilde{M}$ = $(M' \mod 2^{k(d+1)})M$, $4M < 2^{kn}$ = R, M'' = $(\tilde{M}+1)/2^{k(d+1)}$, $0 \leq A$, $B \leq 2\tilde{M}$, $B = \sum_{i=0}^{n+d}(2^k)^i b_i$, $b_i \in 0, 1, ..., 2^k - 1$, and b_i = 0 for $i \geq n$.
2: **Input**: A, B, M''
3: **Output**: OMP(A, B)=$S_{n+d+2} = ABR^{-1} \mod M$, where $0 \leq S_{n+d+2} \leq 2\tilde{M}$
4: $S_0 \leftarrow 0; q_{-d} \leftarrow 0; ...; q_{-1} \leftarrow 0$
5: **for** $i \leftarrow 0, n+d$ **do**
6: $\quad q_i \leftarrow S_i \mod 2^k$
7: $\quad S_{i+1} \leftarrow S_i/2^k + q_{i-d}M'' + b_i A$
8: **end for**
9: $S_{n+d+2} \leftarrow 2^{kd} S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1} 2^{kj}$
10: **return** S_{n+d+2}.

The major differences come from the fact that the bit-oriented conditional additions of Montgomery multiplication were replaced by the word-oriented multiplication in Orup's proposition. The modulus M, is replaced by $\tilde{M}$ (called

$Mwave$ in all subsequent sections) and is given by $\tilde{M} = (M' \mod 2^{k(d+1)})M$, where k is the radix, d is the delay parameter and $-MM' \mod 2^{k(d+1)} = 1$ as described in [22], [23].

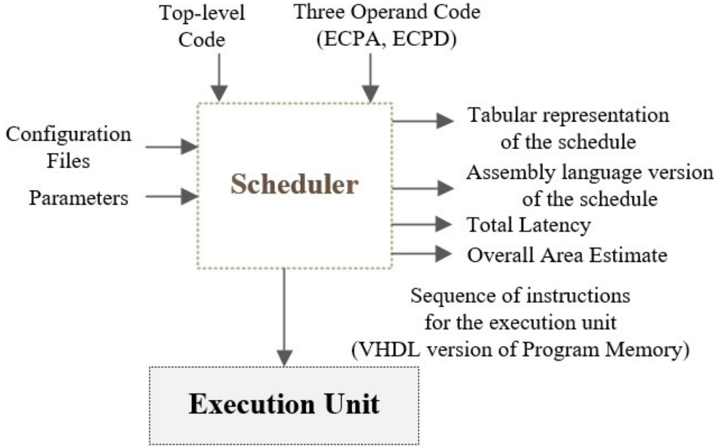

Fig. 1: Overall Design Methodology

$Mwave$ is computed only once before the computations, it remains constant for a given modulus M and radix. For efficient DSP-based implementation of Orup's algorithm [22], [23], we adopt Suzuki's interpretation of the algorithm. The use of memories to store input operands is replaced by registers in our design due to smaller operand sizes as compared to RSA. Also, to reduce the clock cycle count by half, we use two DSP blocks per column to process one 2*k-bit word.

B. Modular Adder/subtractor

Modular addition on long integers is an operation that is extensively used in public key cryptography. Current FPGA families allow fast propagation of the carry signal. Detailed knowledge of the architecture of FPGAs can help the designer to exploit these features and get an efficient design. The parallel prefix networks are used to calculate the generate propagate pairs. Their low latency and regular structure makes them suitable for pipelining. The two most common PPN networks reported in literature are Kogge Stone and Brent Kung PPN adder networks. The idea of PPN adders is generalized to high radix, and implemented using fast carry chains present in modern FPGAs [19].

We implement the Brent-Kung PPN adder based on [19]. The adder architecture is converted into an adder/subtractor unit to fulfill the requirements of our execution unit. As PPN networks are considered viable for pipelined architectures, we intend to investigate the design methodology from [19] further by applying pipelining.

V. Design Methodology

The entire program flow of our design has the following stages

1) Normal to Montgomery Conversion (N2M)
2) Elliptic Curve Scalar Multiplication (ECSM)
3) Projective to Affine Conversion (P2A), including Modular Multiplicative Inversion (MMI) using Fermat's Theorem
4) Montgomery to Normal Conversion (M2N)

A. Scheduler

For this effort, we have developed an automated scheduler that accepts as input a three-operand code (op3) file from the Hyperelliptic.org Explicit Formulas Database [4]. The scheduler accepts the following additional parameters:

1) Number of multipliers: the number of modular multipliers available in the Execution Unit.
2) Assembly file: the name of the assembly file to which the scheduler writes the assembly language version of the program.
3) VHDL ROM file: the name of the VHDL file to which the scheduler writes the machine language version of the program.
4) CSV file: the name of the CSV file to which the scheduler writes a tabular representation of the schedule.

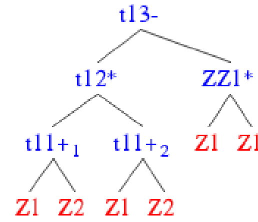

Fig. 2: Tree representation of the operations

The scheduler also provides a detailed output, which contains a description of a tree representation of the three-operand code (in a format that can be supported by phpSyntaxTree – a visual display tool [1]) and a textual description, containing every step in which a multiplication, addition, or subtraction occurs.

1) Detailed operation of the scheduler: First, the scheduler processes the op3 file, and develops a representation of the code in a tree format that can contain multiple roots. The leaves of the tree form the inputs, with each node representing the result of a given operation against the two leaves. The root nodes represent the final operations in the formula (in general, all root nodes are outputs, but not all outputs are root nodes). For example, the operations $t13 = t12 - ZZ1$, $t12 = t11^2$, $t11 = Z1 + Z2$, and $ZZ1 = Z1^2$ are represented in Figure 2. $Z1$ and $Z2$ are inputs, $t13$ and $ZZ1$ are outputs.

For each node, its output name is followed by the associated operation on the two child nodes (e.g., t11+ is the addition of its child nodes). In the PHP syntax tree scheme, a subscript is produced for each unique time a specific node is shown in the tree. This represents node reuse: each node in the tree is represented by a single memory location in the schedule (e.g., t11$+_2$ refers to the second instance of this node in the graphical representation of the tree). Once the tree representation is complete, the scheduler develops three lists:

Point Doubling

Three-operand Code

$P_1 = (X1, Y1, Z1, T1=aZ_1^4)$

$P_3 = 2P_1 = (X3, Y3, Z3, T3=aZ_3^4)$

```
XX = X1^2
t0 = Y1^2
A = 2*t0
AA = A^2
U = 2*AA
t1 = X1+A
t2 = t1^2
t3 = t2-XX
S = t3-AA
t4 = 3*XX
M = t4+T1
t5 = M^2
t6 = 2*S
X3 = t5-t6
t7 = S-X3
t8 = M*t7
Y3 = t8-U
t9 = Y1*Z1
Z3 = 2*t9
t10 = U*T1
T3 = 2*t10
```

Point Addition

Three-operand Code

$P_1 = (X1, Y1, Z1, T1=aZ_1^4)$

$P_2 = (X2, Y2, Z2, T2=aZ_2^4)$

$P_3 = P_1 + P_2 = (X3, Y3, Z3, T3=aZ_3^4)$

```
ZZ1 = Z1^2
ZZ2 = Z2^2
U1 = X1*ZZ2
U2 = X2*ZZ1
t0 = Z2*ZZ2
S1 = Y1*t0
t1 = Z1*ZZ1
S2 = Y2*t1
H = U2-U1
t2 = 2*H
I = t2^2
J = H*I
t3 = S2-S1
r = 2*t3
V = U1*I
t4 = r^2
t5 = 2*V
t6 = t4-J
X3 = t6-t5
t7 = V-X3
t8 = S1*J
t9 = 2*t8
t10 = r*t7
Y3 = t10-t9
t11 = Z1+Z2
t12 = t11^2
t13 = t12-ZZ1
t14 = t13-ZZ2
Z3 = t14*H
ZZ3 = Z3^2
t15 = ZZ3^2
T3 = a*t15
```

Fig. 3: Three Operand Code for Elliptic Curve Point Doubling (ECPD) and Elliptic Curve Point Addition (ECPA) using Modified Jacobian Coordinates

- Operations to be completed: a list containing the operation node, the maximum number of additions between the node and a root node, the maximum number of multiplications between the node and a root node. This list is sorted in descending order according to the number of remaining multiplications necessary to reach a root node. In case the number of multiplications is the same, the nodes with the largest number of remaining additions are listed first
- Operations that have been completed: a list containing values that have already been calculated by the scheduled operations (when the scheduler launches, this list only contains the labels for inputs)
- Registers: a list that represents the memory locations for each value that is still used in the remaining nodes of the tree

With the initial setup complete, the scheduler performs the Algorithm 3. This algorithm is intended to mimic the manual techniques used by the authors to develop schedules for any cryptographic formula. Where manually optimizing a schedule for an elliptic curve sequence of basic Galois Field operations can take minutes or hours, the scheduler can generate the equivalent result within milliseconds or seconds. Tables I, III, II, IV show the full schedules generated manually and using the automated scheduler, respectively. The tables also show attempts to minimize memory usage. For the automatically generated schedule, the scheduler maps variables in the OP3 code to memory locations. In the tables, each variable name is prefixed with the memory location that holds its data (e.g., for X1, the table will show 0:X1).

TABLE I: Manually generated schedule for point doubling

Multiplier 1	Multiplier 2	Adder
$5{:}XX{=}0{:}X1*0{:}X1$	$4{:}t0{=}1{:}Y1*1{:}Y1$	
		$4{:}A{=}4{:}t0+4{:}t0$ $0{:}t1{=}0{:}X1+4{:}A$
$4{:}AA{=}4{:}A*4{:}A$	$0{:}t2{=}0{:}t1*0{:}t1$	$6{:}b2{=}5{:}XX+5{:}XX$ $6{:}t4{=}6{:}b2+5{:}XX$ $6{:}M{=}6{:}t4+t3{:}T1$
$7{:}t5{=}6{:}M*6{:}M$	$1{:}t9{=}1{:}Y1*2{:}Z1$	$5{:}t3{=}0{:}t2-5{:}XX$ $5{:}S{=}5{:}t3-4{:}AA$ $0{:}t6{=}5{:}S+5{:}S$ $4{:}U{=}4{:}AA+4{:}AA$
		$0{:}X3{=}7{:}t5-0{:}t6$ $5{:}t7{=}5{:}S-0{:}X3$
$6{:}t10{=}4{:}U*3{:}T1$	$5{:}t8{=}6{:}M*5{:}t7$	$2{:}Z3{=}1{:}t9+1{:}t9$
		$1{:}Y3{=}5{:}t8-4{:}U$ $3{:}T3{=}6{:}t10+6{:}t10$

TABLE II: Automatically generated schedule for point doubling

Mutliplier 1	Multiplier 2	Adder
$5{:}XX{=}0{:}X1*0{:}X1$	$4{:}t0{=}1{:}Y1*1{:}Y1$	
		$4{:}A{=}4{:}t0+4{:}t0$ $0{:}t1{=}0{:}X1+4{:}A$
$4{:}AA{=}4{:}A*4{:}A$	$0{:}t2{=}0{:}t1*0{:}t1$	$6{:}b2{=}5{:}XX+5{:}XX$ $6{:}t4{=}5{:}XX+6{:}b2$ $6{:}M{=}6{:}t4+3{:}T1$
$1{:}t9{=}1{:}Y1*2{:}Z1$	$5{:}t5{=}6{:}M*6{:}M$	$0{:}t3{=}0{:}t2-5{:}XX$ $7{:}U{=}4{:}AA+4{:}AA$ $2{:}S{=}0{:}t3-4{:}AA$ $4{:}t6{=}2{:}S+2{:}S$
		$0{:}X3{=}5{:}t5-4{:}t6$ $5{:}t7{=}2{:}S-0{:}X3$
$2{:}t8{=}6{:}M*5{:}t7$	$4{:}t10{=}7{:}U*3{:}T1$	$3{:}Z3{=}1{:}t9+1{:}t9$
		$5{:}T3{=}4{:}t10+4{:}t10$ $1{:}Y3{=}2{:}t8-7{:}U$

To optimize memory utilization, the memory locations used to store input coordinates are also used as temporary variables when they are no longer required to be read during a computation. For the point addition, we use the memory locations of coordinates of point Q, whereas coordinates of point P are treated as constants. Out of the 14 locations required for point addition, 8 locations already reserved for P and Q, thus, only 6 additional variables are used. For point doubling as 4 locarions are reserved for Q, only 4 additional variables are required. The exact assignment of these memory locations to particular local memory addresses is determined by a header of the OP3 file.

Algorithm 3 Automated Scheduler

1: **Input**: Any OP3 Code in tree form and a list containing each node in the tree that represents the list of operations to be completed
2: **Output**: Efficient scheduler for the formula
3: **Pre-processing**: The list of operations is sorted by multiplication depth–the maximum number of multiplications from each node to a root. The list is sorted in this order.
4: **while** Number of operations to be completed > 0 **do**
5: IdealMults $\leftarrow$ the multiplication depth of the first N multiplications where N is the number of multipliers
6: onlyAdders $\leftarrow$ the list of adders whose operands have been calculated that have an adder as a parent
7: waitingMultipliers $\leftarrow$ the first N multiplications whose operands have been calculated
8: **if** the multiplication depth of waitingMultipliers and idealMults are the same **then**
9: IsIdeal$\leftarrow$ True
10: **else**
11: IsIdeal $\leftarrow$ False
12: **end if**
13: **if** Multipliers are not in use **then**
14: **if** The size of waitingMulitpliers $< N$ and $<$ the number of remaining possible multiplications **then**
15: pauseMultipliers $\leftarrow$ True
16: **else if** The size of onlyAdders > 0 and not IsIdeal **then**
17: pauseMultipliers $\leftarrow$ True
18: **else**
19: pauseMultipliers $\leftarrow$ False
20: **end if**
21: **end if**
22: **if** pauseMultipliers **then**
23: Append the first node in onlyAdders to the list to be performed
24: **else**
25: **for** For each node in the list of possible operations **do**
26: **if** this node is a multiplication and multiplication is not occurring and the node's operands have been calculated **then**
27: Append this node to the list to be processed unless there would be too many simultaneous multiplications
28: **else if** this node is an addition or subtraction and the node's operands have been calculated **then**
29: Append this node to the list to be performed
30: **end if**
31: **end for**
32: **end if**
33: Increment the simulated clock and process all operations to be processed in parallel
34: Add all nodes that have been completed to the completed list. Record the starting and ending time.
35: Determine if existing registry entries are used by any remaining operations
36: Determine if a new registry entry is needed and add if necessary
37: Assign the outputs of completed operations to registry entries.
38: **loop**

TABLE III: Manually generated schedule for point addition

Mutliplier 1	Multiplier 2	Adder
8:$ZZ1$=2:$Z1*$2:$Z1$	9:$ZZ2$=6:$Z2*$6:$Z2$	10:$t11$=2:$Z1+$6:$Z2$
0:$U1$=0:$X1*$9:$ZZ2$	3:$U2$=4:$X2*$8:$ZZ1$	
11:$t0$=6:$Z2*$9:$ZZ2$	2:$t1$=2:$Z1*$8:$ZZ1$	12:H=3:$U2-$0:$U1$ 13:$t2$=12:$H+$12:H
11:$S1$=1:$Y1*$11:$t0$	3:$S2$=5:$Y2*$2:$t1$	
1:I=13:$t2*$13:$t2$	10:$t12$=10:$t11*$10:$t11$	2:$t3$=3:$S2-$11:$S1$ 2:r=2:$t3+$2:$t3$
8:J=12:$H*$1:I	9:V=0:$U1*$1:I	10:$t13$=10:$t12-$8:$ZZ1$ 10:$t14$=10:$t13-$9:$ZZ2$
3:$t4$=2:$r*$2:r	2:$Z3$=12:$H*$10:$t14$	13:$t5$=9:$V+$9:V
11:$t8$=11:$S1*$8:J	10:$ZZ3$=2:$Z3*$2:$Z3$	3:$t6$=3:$t4-$8:J 0:$X3$=3:$t6-$13:$t5$ 12:$t7$=9:$V-$0:$X3$
12:$t10$=2:$r*$12:$t7$	8:$t15$=10:$ZZ3*$10:$ZZ3$	11:$t9$=11:$t8+$11:$t8$
3:$T3$=$a*$8:$t15$		1:$Y3$=12:$t10-$11:$t9$

TABLE IV: Automatically generated schedule for point addition

Mutliplier 1	Multiplier 2	Adder
8:$ZZ2$=6:$Z2*$6:$Z2$	9:$ZZ1$=2:$Z1*$2:$Z1$	10:$t11$=2:$Z1+$6:$Z2$
3:$U2$=4:$X2*$9:$ZZ1$	0:$U1$=0:$X1*$8:$ZZ2$	
2:$t1$=2:$Z1*$9:$ZZ1$	11:$t0$=6:$Z2*$8:$ZZ2$	3:H=3:$U2-$0:$U1$ 12:$t2$=3:$H+$3:H
2:$S2$=5:$Y2*$2:$t1$	1:$S1$=1:$Y1*$11:$t0$	
10:$t12$=10:$t11*$10:$t11$	12:I=12:$t2*$12:$t2$	11:$t3$=2:$S2-$1:$S1$ 2:r=11:$t3+$11:$t3$
13:$t4$=2:$r*$2:r	0:V=0:$U1*$12:I	11:$t13$=10:$t12-$9:$ZZ1$ 10:$t14$=11:$t13-$8:$ZZ2$
11:J=3:$H*$12:I	8:$Z3$=10:$t14*$3:H	9:$t5$=0:$V+$0:V
10:$ZZ3$=8:$Z3*$8:$Z3$	3:$t8$=1:$S1*$11:J	12:$t6$=13:$t4-$11:J 1:$X3$=12:$t6-$9:$t5$ 13:$t7$=0:$V-$1:$X3$
12:$t15$=10:$ZZ3*$10:$ZZ3$	11:$t10$=2:$r*$13:$t7$	9:$t9$=3:$t8+$3:$t8$
3:$T3$=7:$a*$12:$t15$		2:$Y3$=11:$t10-$9:$t9$
	0:$X3$=1:$X3$	
	1:$Y3$=2:$Y3$	
	2:$Z3$=8:$Z3$	

B. Organization of the Execution Unit

The proposed architecture of the execution unit of our cryptoprocessor is composed of the global control unit (GCU), an

arithmetic unit (AU) containing several multipliers ($MULT_0$, $MULT_1$, ..., $MULT_{r-1}$) and an adder/subtractor (A/S) unit, a shared local coprocessor memory, an instruction memory and an input/output interface as shown in Figure 4. The scheduler implemented using a Python script yields a sequence of instructions in the form of a VHDL ROM file. This instruction sequence is assumed to be preloaded into a program memory implemented as a ROM. Then, the Instruction Decoder dispatches these instructions to the Arithmetic Unit to perform prime field arithmetic.

The coprocessor executes a sequence of multiple additions during one multiplication. To exploit parallelism even further, several multipliers are instantiated in the design to perform independent multiplications at one time. The scheduler can be used to determine an optimal number of multipliers required to achieve either minimum execution time or minimum execution time x area product.

The input and output buses of the coprocessor are 32-bit wide to support external communication through AXI4-Stream interface or a FIFO based interface. The use of two dual-port block memories in the Input and Output Storage is required to accomodate three addresses, one used for input (memory write), and two other used for output (memory read).

The internal datapath width is $2k$-bit, as each Orup multiplier operates on $2k$-bit words at a time. According to the local memory map shown in Table VII, all inputs and parameters required to perform Elliptic Curve Scalar Multiplication (ECSM) are loaded into the local memory initially. As the control unit reads instructions from the program memory, data is loaded from the local memory to the multiplier units one by one. Only after all operands are loaded into the multiplier units, the execution starts. As the local memory is dual-port, both operands of the multipliction or addition/subtraction can be read in one clock cycle. As all multiplications reqired in a batch start execution at the same time, we can use a single control unit for these operations. To utilize the architecture to its fullest, once all multipliers are loaded, and multiplications started, we perform all additions/subtractions that are part of a batch of instructions.

In our modular multiplier design based on [22], [23], α is the number of $2k$ words, where k is 17. For 256-bit operands, α is 9, whereas for 512-bit operands α is 17. For a 256-bit Orup multiplier, after 42 clock cycles the multiplier starts generating k-bit words in each of the following $2*\alpha$ clock cycles to compute the final result. In Figure 4, registers are required at the output of each multiplier to store partial results, as the local memory can support write operation only for one multiplier's results.

C. Instruction Set Architecture of the Execution Unit

In our design, the number of instructions stored in the memory depends on the schedule generated by the Python scheduler. The input to the scheduler can be a three-operand file describing an arbitrarily long sequence of multiplications, additions, and subtractions that can be scheduled statically, before run time, e.g., elliptic curve point addition or point doubling. The instructions are stored in the ROM in the form of batches. Each batch starts with a Batch Begin (BB) instruction to indicate the number of independent multiplications and additions/subtractions that can be executed concurrently.

The instructions stored in the instruction ROM are decoded by an instruction decoder which is part of the global control unit. The decoded information contains the addresses of operands and result, and also the control signals for the arithmetic unit. All the inputs and parameters are loaded into the local coprocessor memory in the beginning. After that instructions are read from the memory with a priority assigned to independent multiplications. During the execution of a multiplication, modular additions/subtractions are performed and their result is also written back to the local memory.

As shown in Table V, the instructions are divided into arithmetic and control flow instructions. We use different notations for the widths of the respective fields, dependent on the size of the local memory, 16-bit register set, and program memory. Number of bits required to represent the number of multiplications in a batch is denoted by *s*. We use *s_lm*, *s_reg* and *s_pm* to represent the number of bits required to represent local memory address, register set address and instruction memory address respectively.

VI. Results and Comparison

All results presented in this paper are generated using Vivado Design Suite 2016.2. The evaluation board used is ZedBoard Zynq-7000 ARM/FPGA SoC Development Board, based on the xc7z020clg484-1 Zynq device. The design has been functionally verified using Vivado simulator. Software implementation for test vector generation was developed using Python.

We present results for three variants of the execution units, with one, two, and four multipliers, respectively. In each case, a single adder/subtractor is included in the execution unit as well. The supported ECC schemes have a field of 256 bits. The execution unit is configurable to perform scalar multiplication for even higher security level with bigger operand sizes. The scheduler generates a sequence of operations divided into batches of independent multiplications that can be executed in parallel. Each of these batches also has multiple modular addition/subtractions which can be performed during the multiplications. Let N_m be the number of multiplications in a given batch, and N_{as} be the number of additions/subtractions in a given batch. The execution time of N_m multiplications in a batch in terms of number of clock cycles can be calculated as

$$CC_{Nm} = N_m * M_{rd} + M_{comp} + N_m * M_{wr}$$

where M_{rd} and M_{wr} are the number of clock cycles required to read and write the operands from/to the local memory respectively, and M_{comp} is the number of clock cycles required to perform an entire modular multiplication.

Similarly, the execution time of N_{as} addition/subtractions in terms of number of clock cycles can be calculated as

$$CC_{Nas} = N_{as} * (AS_{rd} + AS_{comp} + AS_{wr})$$

where AS_{rd} and AS_{wr} are the number of clock cycles required to read and write the operands from/to the local memory respectively, and AS_{comp} is the number of clock cycles required to perform an addition/subtraction. In our implementation, M_{rd}, M_{wr}, AS_{rd}, and AS_{wr} are all equal to 1, independently of the

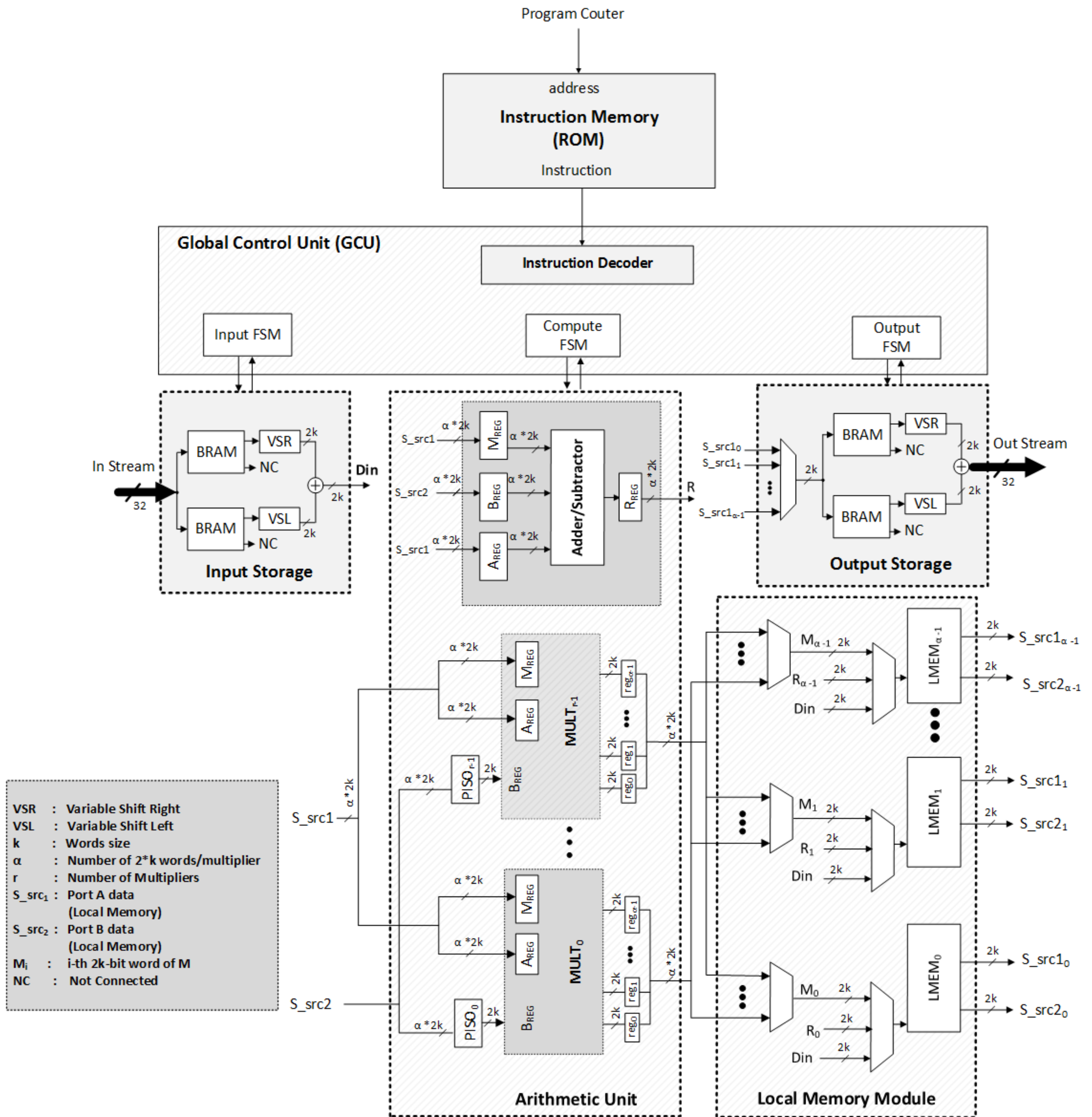

Fig. 4: Detailed Block Level Diagram of the Execution Unit

TABLE V: Instruction Set Architecture

Mnemonic and Operands	**Opcode**	**Field Widths**	**Description**
		Arithmetic Instructions	
BB Nm, Nas	0000	Nm:s, Nas:s	Batch begin. Indicates number of multiplications and addition/subtractions in a batch
MUL dst, src1, src2	0001	dst:s_lm, src1:s_lm, src2:s_lm	Multiply src1 by src2 and store the result in dst
ADD dst, src1, src2	0010	dst:s_lm, src1:s_lm, src2:s_lm	Add src1 to src2 and store the result in dst
SUB dst, src1, src2	0011	dst:s_lm, src1:s_lm, src2:s_lm	Subtract src2 from src1 and store the result in dst
		Control Flow Instructions	
MOV dst, src	0100	dst: s_lm, src: s_lm	Move data from src to dst
BIT src1, src2	0101	src1:s_lm, src2:s_reg	C flag = value of bit src2 of operand src1
JMP NC addr	0110	addr:s_pm	Jump to a label if C=0, else PC = PC + 1
DEC src1	0111	src1:s_reg	Decrement by 1 (Sets Z flag when the result is '0')
JMP NZ addr	1000	addr:s_pm	Jump to a label if Z = 0, else PC = PC + 1
CALL addr	1001	addr:s_pm	Call a subroutine
RET	1010		Return from the subroutine

TABLE VI: Assembly language version of the top-level code (without division into arithmetic instruction batches)

Stage	Label	Instr.	Op1	Op2	Op3
Normal to Montgomery Conversion (N2M)		MUL	0	0	6
		MUL	1	1	6
		MUL	2	2	6
		MUL	3	3	6
Elliptic Curve Scalar Multiplication (ECSM)		MOV	9	0	
		MOV	10	1	
		MOV	11	2	
		MOV	12	3	
	L1:				
		CALL	ECPD		
		BIT	8	R0	
		JMP NC	M1		
		CALL	ECPA		
	M1:				
		DEC	R0		
		JMP NZ	L1		
Projective to Affine Conversion (P2A) Part 1: Modular Multiplicative Inversion (MMI)		MAS	7	4	7
		MOV	13	11	
	L2:				
		MUL	13	13	13
		BIT	7	R1	
		JMP NC	M2		
		MUL	13	13	11
	M2:				
		DEC	R1		
		JMP NZ	L2		
Projective to Affine Conversion (P2A) Part 2:		MUL	14	13	13
		MUL	9	9	14
		MUL	15	14	13
		MUL	10	10	15
Montgomery to Normal Conversion (M2N)		MUL	9	16	9
		MUL	10	16	10

TABLE VII: Local Memory and Register Map

Location in LMEM	Variable	Description
	Local Memory Map	
0	Xp:Xp*R	Xp:Xp in Montgomery Domain
1	Yp:Yp*R	Yp:Yp in Montgomery Domain
2	Zp:Zp*R = 1:R	Zp:Zp in Montgomery Domain
3	Tp:Tp*R = a:a*R	Tp:Tp in Montgomery Domain
4	M	Modulus
5	Mwave	Mwave for Orup Multiplier
6	Rsq	R^2 mod M
7	2:M-2	M-2 for MMI Implementation
8	k	scalar value
9	Xq*R:Xq	Xq in Montgomery Domain:Xq
10	Yq*R:Yq	Yq in Montgomery Domain:Yq
11	Zq*R	Zq in Montgomery Domain
12	Tq*R	Tq in Montgomery Domain
13	Zq*R:Zq^{-1}*R	Used in P2A
14	Zq^{-2}*R	Used in P2A
15	Zq^{-3}*R	Used in P2A
16	1	
17-22		Temporary Memory Locations
	Register Map	
R0	i	ceil($\log_2$k)-2
R1	j	ceil($\log_2$(M-2))-2

operand size. For 256-bit operands, M_{comp} is 60 and AS_{comp} is 5; for 512-bit operands, M_{comp} is 110 and AS_{comp} is 5.

Table VIII shows the clock cycle counts for each individual stage of the entire ECC operation. CC_{N2M} requires four independent multiplications. In the architecture with one multiplier, all multiplications will be executed one by one. However, for the other two architectures with more than one multiplier, these multiplications can be executed in the form of batches to reduce the overall clock cycle count. N_i is the number of iterations required to perform a scalar multiplication. In our case it is set to $\lceil(\log_2 k)\rceil$-1 = 255. We assume that Elliptic Curve Point Addition (ECPA) will be executed 50% of the times.

TABLE VIII: Execution time in terms of clock cycles for all stages of ECC. Note: N_i is the number of iterations of the main loop required, Num_{BB} - Number of batches

Clock Cycles per stage	**Formula**	**1 Mult**	**2 Mult**	**4 Mult**
$\mathbf{CC_{N2M}}$	Num_{BB} * CC_{Nm}	248	128	68
$\mathbf{CC_{ECSM}}$	CC_{PDBL} * N_i + CC_{PADD}*$\lfloor(N_i/2)\rfloor$	300,821	165,941	141,776
$\mathbf{CC_{P2A}}$	CC_{Nm}* N_i + CC_{Nm}*$\lfloor(N_i/2)\rfloor$ + Num_{BB} * CC_{Nm}	23,932	23,812	23,812
$\mathbf{CC_{M2N}}$	Num_{BB} * CC_{Nm}	124	64	64
$\mathbf{CC_{TOTAL}}$	CC_{N2M} + CC_{ECSM} + CC_{P2A} + CC_{M2N}	325,125	189,945	165,720

TABLE IX: Comparison with Previous Implementations of Scalar Multiplication for 128-bit security

Reference	Family	Area Utilization	Clock Cycles	Freq MHz	Time ms
This Work	Zynq-7000	1011 Slices/1677 LUTs, 9 Block RAMs, 18 DSP Units	325,125	227	1.43
		1636 Slices/2712 LUTs, 9 Block RAMs, 36 DSP Units	189,945		0.83
		2841 Slices/4698 LUTs, 9 Block RAMs, 72 DSP Units	165,720		0.73
Guillermin et al. [8]	Stratix-II	9177 ALMs, 96 DSP Units	106,896	157	0.68
Lai et al. [16]	Virtex-5	3657 Slices, 10 DSP Units	225,720	263	0.86
Amiet et al. [2]	Virtex-7	6816 LUTs, 20 DSP Units	335,360	225	1.49
Sakiyama et al. [21]	Virtex-II	10,847 Slices	269,886	100	2.70
McIvor et al. [17]	Virtex-II	15,755 Slices, 256 18 x 18 Multipliers	151,360	40	3.86
Ananyi et al. [3]	Virtex-4	20,793 Slices, 1 Block RAM, 32 DSP Units	366,000	60	6.10
Ghosh et al. [7]	Virtex-4	20,123 Slices/33,700 LUTs	129,300	43	7.70
Hamilton et al. [10]	Virtex-5	2025 Slices	970,000	100	9.70
Vliegen at al. [26]	Virtex-II	1947 Slices, 9 Block RAMs, 7 18 x 18 Multipliers	1074,625	68	15.76

Table IX shows our results for scalar multiplication with one, two and four multipliers respectively. All reported results from literature perform ECSM over a 256-bit field. The architecture in [21] is a dual-field design. The number of clock cycles in [17] is reduced due to the use of a full-word 256*256-bit Montgomery multiplier which takes only 32 clock cycles to perform one multiplication. At the same time, it results in large area consumption as the multiplier itself requires 11,992 slices. In [8] as the design in based on Residue Number System (RNS), a very large number of multipliers is required. The parameter size can not be changed without configuration. Although most of these designs try to achieve flexibility by allowing arbitrary parameter sizes or elliptic curves, the issue of manual scheduling of operations which needs to be very specific to their designs is not addressed. Even small changes in the design would require the designer to perform the entire scheduling of operations from scratch. Our results show that after adding the capability of automated scheduling, we still achieve comparable results in terms of the latency and area utilization. At the same time, it facilitates the use of same architecture for any cryptosystem composed of basic prime field arithmetic. The results also indicate that for more complicated cryptosystems that require larger number of independent multiplications at one time, the clock cycle count can be reduced even further resulting in substantial reduction in the overall time.

VII. CONCLUSIONS

The entire setup can be used as a framework to simplify and ease the development cycle of cryptosystems composed of prime field arithmetic at the lowest level of hierarchy. We have developed a generalized execution unit which is capable to be configured for different operand sizes and process schedules for multiple sets of parameters and still yield comparable results to other hardware implementations based on ECC. We also utilize specialized embedded resources available in FPGAs to implement the arithmetic units to achieve performance goals. Our methodology and tools bridge the gap between cryptographers and hardware engineers. Cryptographers can finish their algorithmic investigations with the multiple variants of the three-operand code, similar to that available at [4]. The hardware designers can take over from that point on, and easily and quickly investigate multiple algorithmic alternatives, using our automated scheduler. The scheduler can easily handle very large sequences of instructions to automate a very cumbersome task performed manually by hardware designers for each cryptosystem.

VIII. FUTURE WORK

We plan to extend the current version of the execution unit with a pipelined adder/subtractor unit that can process multiple additions and subtractions concurrently, while still taking into account all data dependencies. We also intend to extend the entire setup to all commonly used pairing based algorithms

like Tate pairing and Optimal Ate pairing for multiple operand sizes and different parameter sets.

REFERENCES

[1] phpSyntaxTree. http://ironcreek.net/phpsyntaxtree/.

[2] D. Amiet, A. Curiger, and P. Zbinden. Flexible FPGA-based architectures for curve point multiplication over GF(p). *2016 Euromicro Conference on Digital System Design (DSD)*, 00:107–114, 2016.

[3] K. Ananyi, H. Alrimeih, and D. Rakhmatov. Flexible hardware processor for elliptic curve cryptography over NIST prime fields. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17:1099–1112, 2009.

[4] D. J. Bernstein and T. Lange. Explicit-formulas database. http://www.hyperelliptic.org/EFD.

[5] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 213–229, London, UK, UK, 2001. Springer-Verlag.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[7] S. Ghosh, M. Alam, D. R. Chowdhury, and I. S. Gupta. Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks. *Comput. Electr. Eng.*, 35(2):329–338, March 2009.

[8] N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over Fp. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 48–64, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] X. Guo, J. Fan, P. Schaumont, and I. Verbauwhede. Programmable and parallel ecc coprocessor architecture: Tradeoffs between area, speed and security. In *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2009*, pages 289–303. Springer-Verlag, September 2009.

[10] M. Hamilton and W.P. Marnane. FPGA implementation of an elliptic curve processor using the GLV method. *2009 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2009)*, 00:249–254, 2009.

[11] D. Harris and K. Kelley. Parallelized very high radix scalable Montgomery multipliers. In *Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 306–311, 2005.

[12] D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu. An improved unified scalable radix-2 Montgomery multiplier. In *Computer Arithmetic*, 2005.

[13] M. Huang, K. Gaj, and T. El. Ghazawi. New hardware architectures for Montgomery modular multiplication algorithm. *Transactions on Computers*, 2010.

[14] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[15] Ç.K. Koç, T. Açar, and B.S. Jr. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[16] J-Y. Lai, Y. Wang, and C. Huang. High-performance architecture for elliptic curve cryptography over prime fields on FPGAs. *Interdisciplinary Information Sciences*, 18(2):167–173, 2012.

[17] C.J. McIvor, M. McLoone, and J.V. McCanny. Hardware elliptic curve cryptographic processor over GF(p). *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(9):1946–1957, 9 2006. Copyright 2008 Elsevier B.V., All rights reserved.

[18] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193–199, Jul 1995.

[19] M Rogawski, E. Homsirikamol, and K. Gaj. A novel modular adder for one thousand bits and more using fast carry chains of modern FPGAs. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8, 2014.

[20] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2006*, page 415–429. Springer-Verlag, October 2006.

[21] K. Sakiyama, E. D. Mulder, B. Preneel, and I. Verbauwhede. A parallel processing hardware architecture for elliptic curve cryptosystems. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing, ICASSP 2006, Toulouse, France, May 14-19, 2006*, pages 904–907, 2006.

[22] D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Workshop on Cryptographic Hardware and Embedded Systems—CHES 2007*. Springer-Verlag, 2007.

[23] D. Suzuki and T Matsumoto. How to maximize the potential of FPGA-based DSPs for modular exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011.

[24] A. F. Tenca and Ç. K. Koç. A scalable architecture for modular multiplication based on Montgomery's algorithm. *IEEE Trans. Computers*, 52(9):1215–1221, 2003.

[25] A. F. Tenca, G. Todorov, and Ç. K. Koç. *High-Radix Design of a Scalable Modular Multiplier*, pages 185–201. Springer Berlin Heidelberg, 2001.

[26] J. Vliegen, N. Mentens, J. Genoe, A. Braeken, S. Kubera, A. Touhafi, and I. Verbauwhede. A compact FPGA-based architecture for elliptic curve cryptography over prime fields. In *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010, Rennes, France, 7-9 July 2010*, pages 313–316, 2010.