A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware [*]

Gerardo Orlando[1] and Christof Paar[2]

[1] General Dynamics Communication Systems
77 A St., Needham MA 02494-2892, USA
`gerardo.orlando@gd-cs.com`
[2] ECE Department,Worcester Polytechnic Institute
100 Institute Road Worcester, MA 01609, USA
`christof@ece.wpi.edu`

Abstract. This work proposes a new elliptic curve processor architecture for the computation of point multiplication for curves defined over fields $GF(p)$. This is a scalable architecture in terms of area and speed specially suited for memory-rich hardware platforms such a field programmable gate arrays (FPGAs). This processor uses a new type of high-radix Montgomery multiplier that relies on the precomputation of frequently used values and on the use of multiple processing engines.

1 Introduction

This work introduces, to the authors' knowledge, the first documented processor architecture for the computation of elliptic curves point multiplications for curves defined over fields $GF(p)$. Hardware implementations have been documented for the computation of point multiplications for curves defined over $GF(2^m)$. The most notable implementations include [1–6].

The architecture presented here is based on the standalone elliptic curve processor architecture introduced in [6]. This architecture is modular, programmable, and suitable for algorithms that rely on precomputations.

Multiplication is typically the most critical operation in the computation of elliptic curves point multiplications. The architecture introduced here uses a Montgomery multiplier. This type of multiplier has been the subject of extensive research, see for example [7–11].

For the elliptic curve processor (ECP) introduced here, this work develops a new multiplier architecture that draws from [9, 12] an approach for high radix multiplication, from [8, 9] the ability to delay quotient resolution, and from [10] the use of precomputation. In particular, this work extends the concept of precomputation. The resulting multiplier architecture is a high-radix, precomputation-based modular multiplier.

2 Mathematical Background

This section provides a brief introduction to elliptic curve point multiplication. Additional information can be found in [13, 14].

The ECP computes elliptic curve point multiplications for arbitrary curves defined over $GF(p)$. Point multiplication is defined as the product $kP = \underbrace{P + P + \ldots P}_{k\ times}$, where k is an integer and P is a point on the elliptic curve. For fields $GF(p)$, the curves of interest are defined by $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \not\equiv 0 \bmod M$ and $M > 3$.

One can visualize the computation of point multiplications as a hierarchy of processing functions. At the top of the hierarchy are the point multiplication functions. These functions compute point multiplications with repeated point additions and point doubles. At the next level of the hierarchy are the point addition and point double functions, which are intimately related to the co-ordinates used to represent the points. At the bottom of the hierarchy are the finite field functions required to perform the point addition and the point double functions. Figure 1 shows how this hierarchy maps into the ECP architecture.

The ECP is best suited for the computation of point multiplications using projective coordinates. When compared against algorithms for affine coordinates, algorithms for projective coordinates trade inversions in the point addition and in the point double operations for a larger number of multiplications and a single inversion at the end of the algorithm. This inversion can be computed with repetitive multiplications: $a^{-1} \bmod M \equiv a^{M-2} \bmod M$, for prime modulus M.

The ECP uses a Montgomery multiplier. The main advantage of this type of multiplier is that it facilitates quotient estimation and facilitates carry propagation in hardware adders. Their main disadvantage is that they compute weighted products: $\mathrm{mult}(A,B) = ABR^{-1} \bmod M$, where R is a constant.

For Montgomery multiplication to be effective, the input operands to the point multiplication algorithm must be transformed into weighted residues of the form $AR \bmod M$. The algorithm is then executed using these residues. At the end of the algorithm, the results are then transformed back to not weighted residues. Note that as described in [15] the addition and subtraction of these residues can be performed using traditional modular addition and subtraction operations. For most cryptographic algorithms, the cost of these transformations is amortized over a large number of operations.

3 Processor Architecture

The elliptic curve processor (ECP), shown in Figure 1, consists of three main components. These components are the main controller (MC), the arithmetic unit controller (AUC), and the arithmetic unit (AU). The MC is the ECP's main controller. It orchestrates the computation of kP and interacts with the host system. The AUC controls the AU. It orchestrates the computation of

point additions/subtractions, point doubles, and coordinate conversions. It also guides the AU in the computation of field inversions. The AU is the hardware that computes field additions/subtractions and multiplications, and performs comparisons.
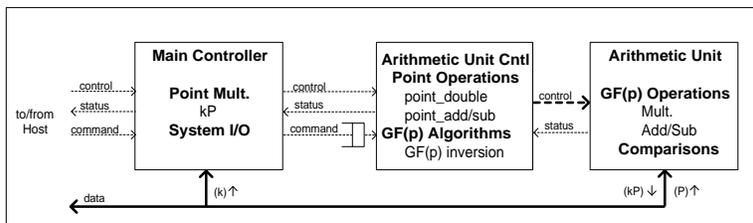

Fig. 1. Elliptic curve processor architecture

The following is a typical sequence of steps for the computation of kP in the ECP using the double-and-add algorithm and the projective coordinates algorithms shown in the appendix.

First, the host loads k into the MC, loads the coordinates of P into the AU, and commands the MC to start processing. The MC does its initialization and then commands the AUC to do its initialization. The AUC initialization includes the conversion of P from affine to projective coordinates and the conversion of these coordinates into weighted residues ($\tilde{X} = XR \bmod M$, $\tilde{Y} = YR \bmod M$, $\tilde{Z} = R \bmod M$). During the computation of kP, the MC scans one bit of k at time starting with the second most significant coefficient and ending with the least significant one. In each of these iterations, the MC commands the AU/AUC to do a point double. If the scanned bit is a 1, it also commands the AU/AUC to do a point addition. For each of these point operations, the AUC generates the control sequence that guides the AU through the computation of the required field and comparison operations. After the least significant bit of k is processed, the MC commands the AU/AUC to convert the result back to affine coordinates. The AU/AUC first converts the result to affine coordinates and then converts the coordinates to not weighted residues (x, y). Then, the MC signals to the host the completion of the kP operation. Finally, the host reads the coordinates of kP from the AU.

The ECP uses two loosely coupled controllers, the MC and the AUC, that execute their respective operations concurrently. These are programmable processors that execute one instruction per clock cycle.

The AU incorporates a multiplier, an adder (or adders), and a register file, all of which can operate in parallel on different data. The AU's large register set supports algorithms that rely on precomputations. An example of a precomutation-based algorithm is an adaptation of a fixed base exponentiation method introduced in [16] for operations involving a known point. This algorithm requires on average $\lfloor m/w \rfloor + 2^w$ point additions, the storage of $\lceil m/w \rceil$ points, and no point

doubles. In the previous expressions, w is the window size, which is a measure of the number of bits processed in parallel. To illustrate the benefits of precomputation, consider a fixed point multiplication for an arbitrary curve defined over $GF(2^{192} - 2^{64} - 1)$, which is one of the fields recommended in [17]. Compared to the traditional double-and-add algorithm, the fixed point algorithm is over four times faster (assuming the use of the projective coordinates in [18] with $Z = 1$ and $w = 4$).

4 Arithmetic Unit

The Arithmetic Unit (AU) is the ECP's main processing unit. As Figure 2 shows, it consists of a register file, an adder (or adders), and a multiplier. The multiplier is the AU's most critical component, and, consequently, it is the component that drives the AU's architecture. The AU's architecture is defined at a high level by the multiplication algorithm it implements and at a low level by the number representation it uses.
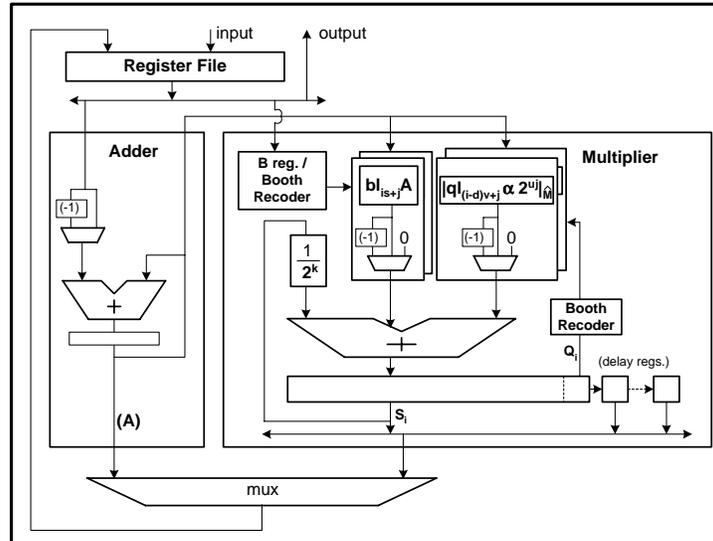

Fig. 2. Arithmetic Unit

The most popular cryptographic algorithms in use today require arithmetic with large operands ($160 \ldots 1024^+$ bits). To achieve a high rate of computation, most hardware implementations resort to iterated multiplication methods that approximate the desired result rather than computing exact ones. The approximated results are then refined to exact results in post-processing operations. The tradeoff is accuracy for speed. The ECP's multiplier is an example. It im-

plements an iterated multiplication algorithm that approximates the multiplication of $AR \bmod M$ and $BR \bmod M$ as $ABR \bmod M + \epsilon M$, where ϵM is a measure of the accuracy of the multiplication. Note that for the basic forms of Montgomery multiplication $\epsilon = 1$.

Number representation is an important element of an arithmetic architecture. It defines how the numbers are represented and consequently how arithmetic is conducted. The selection of a number representation is influenced by the design methodology, the target architecture, and the area-time (or cost-speed) goals. The ECP architecture is independent of number representations. To validate the ECP architecture a prototype that uses redundant number representation was developed. The implementation results are discussed in section 8.

5   Modular Multiplication Algorithm

Algorithm 1 shows the ECP's main multiplication algorithm. This algorithm is a generalized version of the Montgomery multiplication algorithm with quotient pipelining introduced in [9]. This generalized version supports positive and negative operands and incorporates Booth recoding and precomputation. Positive and negative numbers arise naturally in Booth recoding and they are often used in elliptic curve algorithms.

Booth recoding is a technique that allows the representation of a two's complement number $B = \sum_{i=0}^{s-1} B_i 2^{ui}$, where $s = n/u$, $B_{i<s-1} \in [0, 2^u)$ and $B_{s-1} \in [-2^{u-1}, 2^{u-1})$, as $B = \sum_{i=0}^{s-1} B'_i 2^{ui}$ where $B'_i \in [-2^{u-1}, 2^{u-1}]$. Here we assume that B is represented by an integer number of digits of radix 2^u and also that its most significant bit represents the sign.

This work uses the Modified Booth Algorithm, which is a window based method [19, 20]. This method uses s windows, where each window$_i$ groups the set of bits $(b_{iu+(u-1)}b_{iu+(u-2)}..b_{iu-1})_2$ for $i = 0..s - 1$, and where $b_{-1} = 0$ ($B_i = \sum_{j=0}^{u-1} b_{iu+j} 2^j$). The set of bits enclosed by window$_i$, is encoded as $B'_i = -b_{iu+(u-1)}2^{u-1} + (\sum_{j=0}^{u-2} b_{iu+j} 2^j) + b_{iu-1}$. Note that in Algorithm 1 the recoding is done on a digit-by-digit basis. For this algorithm r, s, u, and v divide k. The variables qh_i and bh_i are respectively the most significant bits of Q_i and B_i.

The validity of Algorithm 1 can be proven using an induction argument similar to the one used in [9] to prove the validity of the Montgomery multiplication algorithm in which this algorithm is based. One can verify with induction on i that Equation (1) defines an invariant of the loop. For this verification note that $\lfloor Si/2^k \rfloor$ defines a truncated division equivalent to $(S_i - Q_i)/2^k$.

Notation: The symbol $|x|_{\hat{M}}$ is used to express an approximate modulo reduction that satisfies the following relation: $|x|_{\hat{M}} = x \bmod M + \epsilon M \equiv x \bmod M$. $|x|_M$ is used to express least residue; that is, $|\ |x|_M\ | < M$, where the symbol $|y|$ represents the absolute value of y.

Using the loop invariant in Equation (1), one can verify that when $i = n+d+1$ the output of the algorithm satisfies Equation (2). This equation establishes that the multiplication output is $S_{n+d+2} \equiv \left| ABR^{-1} \right|_{\hat{M}}$ (note that $QM \equiv 0 \bmod M$). This equation also defines the range, or accuracy, of the multiplication result in

terms of the maximum values of A and B (note $B_{i \geq n} = 0$); the maximum value for the reduction terms, $\mathcal{QM}$, which is defined in Equations (4-5)(note $Q_0 = 0$); the value of the multiplication constant $R = 2^{kn}$; and the quotient resolution delay, d.

Algorithm 1: Modular Multiplication with Precomputation

Inputs:

$A \in (-\mathcal{A}, \mathcal{A})$, $\mathcal{A} > 0$

$B = \sum_{i=0}^{n+d} B_i 2^{ki} \in (-\mathcal{B}, \mathcal{B})$, $\mathcal{B} > 0$, $B_{i<t-1} \in [0, 2^k)$, $B_{i=t-1} \in [-2^{k-1}, 2^{k-1})$,
$\quad B_{i \geq t} = 0$, $t \leq n$

$\alpha \equiv \left| 2^{-\overline{k}(d+1)} \right|_M$, $gcd(M, 2) = 1$, $R = 2^{kn}$, d – quotient resolution delay

Output:

$S_{n+d+2} \equiv |AB/R|_{\hat{M}} \in (-(\mathcal{AB} + \mathcal{QM})/R, (\mathcal{AB} + \mathcal{QM})/R)$

/* Pre-processing */

1. $\quad S_0 = 0$, $Q_{i<0} = 0$

2. $\quad$ for $i = 0$ to 2^{r-1} do

2.1. $\quad A[i] = iA$

$\quad$ end for

3. $\quad$ for $i = 0$ to 2^{u-1} do

3.1. $\quad$ for $j = 0$ to $v - 1$ do

3.1.1. $\quad \alpha[i, j] = \left| i\alpha 2^{uj} \right|_{\hat{M}}$

$\quad$ end for

$\quad$ end for

/* Processing */

4. $\quad$ for $i = 0$ to $n + d$ do

$\quad$ /* Quotient Determination */

4.1. $\quad Q_i = |S_i|_{2^k}$

$\quad$ /* Recoding: $ql_j \in [-2^{u-1}, 2^{u-1}]$; $bl_j \in [-2^{r-1}, 2^{r-1}]$; $qh_j, bh_j \in [0, 1]$ */

4.2. $\quad qh_i 2^k + \sum_{j=0}^{v-1} ql_{iv+j} 2^{uj} = Q_i + qh_{i-1}$ $\quad$ /* $k = uv$ */

4.3. $\quad bh_i 2^k + \sum_{j=0}^{s-1} bl_{is+j} 2^{rj} = B_i + bh_{i-1}$ $\quad$ /* $k = rs$ */

$\quad$ /* Reduction */

4.4. $\quad \widetilde{Q\alpha}_i = \sum_{j=0}^{v-1} \alpha[\ |ql_{iv+j}|\ , j\ ](sign(ql_{iv+j}))$

4.5. $\quad \widetilde{AB}_i = \sum_{j=0}^{s-1} A[\ |bl_{is+j}|\ ](sign(bl_{is+j}))2^{rj}$

4.6. $\quad S_{i+1} = \lfloor S_i/2^k \rfloor + \widetilde{Q\alpha}_{i-d} + \widetilde{AB}_i$

$\quad$ end for

/* Post-processing */

5. $\quad S_{n+d+2} = 2^{kd} S_{n+d+1} + \sum_{i=0}^{d-1} Q_{n+1+i} 2^{ki} + qh_n$

Loop Invariant:

$$2^{ki} S_i + 2^{k(i-d)} \sum_{j=0}^{d-1} Q_{i+j-d} 2^{kj} + qh_{i-d-1} 2^{k(i-d)} = \tag{1}$$

$$2^k A((\sum_{j=0}^{i-1} B_j 2^{kj}) - bh_{i-1} 2^{ki}) + 2^k \sum_{j=0}^{i-d-2} (\widetilde{Q\alpha}_{j+1} 2^{k(d+1)} - \widetilde{Q}_{j+1}) 2^{kj}$$

Result after n+d+1 loop iterations:

$$2^{kd}S_{n+d+1} + \sum_{j=0}^{d-1} Q_{n+1+j}2^{kj} + qh_n = \frac{AB + \sum_{j=0}^{n-1}(\widetilde{Q\alpha}_{j+1}2^{k(d+1)} - \widetilde{Q}_{j+1})2^{kj}}{R}$$

(2)

$$= (AB + QM)/R$$
$$\in (-(\mathcal{AB} + \mathcal{QM})/R, (\mathcal{AB} + \mathcal{QM})/R))$$

QM:

$$\widetilde{Q}_i = \sum_{j=0}^{v-1} ql_{iv+j}2^{uj}$$

(3)

$$QM = \sum_{j=0}^{n-1}(\widetilde{Q\alpha}_{j+1}2^{k(d+1)} - \widetilde{Q}_{j+1})2^{kj}$$

(4)

$$\in (-\mathcal{QM}, \mathcal{QM})$$
$$\mathcal{QM} > max(\ |QM|\ )$$

(5)

Note that implementations can take advantage of the parallelism defined in steps 4.2-4.6 of Algorithm 1 without using Booth recoding. These implementations can set $qh_i = bh_i = 0$ for all i, and use digits $ql_{iv+j} \in [0, 2^u)$ and $bl_{is+j} \in [0, 2^r)$.

6 Analysis of Modular Multiplication Algorithm

In order to realize an area efficient multiplier, the ECP implements Algorithm 1 using precomputation. Precomputation reduces the complexity of the multiple input adder needed to add all the terms in step 4.6 of Algorithm 1 at the expense of a set of additions at the beginning of the algorithm (steps 2 and 3) and storage. The issues associated with the implementation of Algorithm 1 using precomputations are studied in the next sections.

6.1 Accuracy

The accuracy of the modular multiplication result is influenced by the range of the input operands, the method employed to compute reduction terms, the multiplication constant R, and the quotient resolution delay d.

In [12] two methods are defined for the computation of the reduction terms $|x\alpha|_{\hat{M}}$. These methods are referred to as multiplication-based and lookup-based reduction methods. The multiplication-based approach computes $|x\alpha|_{\hat{M}} = x\,|\alpha|_M$. The lookup-based method computes $|x\alpha|_{\hat{M}} = |x\alpha|_M$. The accuracy of one multiplication-based and two lookup-based reduction methods are summarized in Table 1. (Note that the reduction method affects the value of $\widetilde{Q\alpha}_i$.)

R is a design parameter that influences the reduction accuracy of the multiplier. As Table 1 shows, the accuracy of the result is bounded by the magnitude of $\mathcal{QM}$, which grows proportionally with R. For applications requiring iterated multiplications, such as modular exponentiations, R is often chosen so that the accuracy of a multiplication result falls in the range $(-2\mathcal{QM}/R, 2\mathcal{QM}/R)$, or $[0, 2\mathcal{QM}/R)$ when handling only positive numbers. Examples for this last application can be found in [9], for which $A, B \in [0, 2(2^{k(d+1)}M))$, $R > 4(2^{k(d+1)}M)$ and $\mathcal{QM} \in [0, 2^{k(d+1)}MR)$.

The results in Table 1 correspond to the worse case values for $\mathcal{QM}$, where the value of a modulo reductions is approximated as M. Parameter selection can greatly improve the accuracy and speed of a multiplication; for example, [17] specifies modulus of the form $M = \sum_{i=t}^{m-1} m_i 2^i \, {}^+/_- \, 1$, for which $\left|2^{-kx}\right|_M = (M \, {}^-/_+ \, 1)/2^{kx}$ for $t \geq kx$. For $t \geq k(d+1)$, $\mathcal{QM} < MR$ for all the reduction methods listed in Table 1.

Table 1. Accuracy of multiplication- and lookup-based reduction methods

Red. method	$\widetilde{Q\alpha_i}$		$\mathcal{QM}$ (worst case)
Multiplication	$\sum_{j=0}^{v-1} ql_{iv+j} \left\lfloor 2^{-k(d+1)} \right\rfloor_M$	2^{uj}	$2^{k(d+1)} M \left(\frac{2^{kn}-1}{2^u-1}\right)\left(\frac{2^u}{2}\right) < 2^{k(d+1)} MR$
Lookup 1	$\sum_{j=0}^{v-1} \left\lfloor ql_{iv+j} 2^{-k(d+1)} \right\rfloor_M$	2^{uj}	$2^{k(d+1)} M \left(\frac{2^{kn}-1}{2^u-1}\right) < 2^{k(d+1)} MR\left(\frac{2}{2^u}\right)$
Lookup 2	$\sum_{j=0}^{v-1} \left\lfloor ql_{iv+j} 2^{-k(d+1)} 2^{uj} \right\rfloor_M$		$2^{k(d+1)} M \left(\frac{2^{kn}-1}{2^k-1}\right) v < 2^{kd} MR(2v)$

6.2 Processing Time

Equation (6) provides a processing time approximation for Algorithm 1. This equation assumes that a single precomputation engine performs all the precomputations and transmits them to the respective processing units. In this equation Tb and Tq represent the processing time for the computation of $\widetilde{AB_i}$ and $\widetilde{Q\alpha_i}$ for $i = 0..n + d$. The processing time is the sum of the precomputation time, which is identified with the p subscript, and the processing time, which is identified with the m subscript. (Note that the processing time Tb_m include n processing operations because $B_{i \geq n} = 0$. This condition does not apply to Tq_m.)

The expression in Equation (6) is normalized with respect to a reference unit of time. The processing cost of a precomputation operation is weighted by a factor a and the processing cost of a processing operation is weighted by the factor b. The factor c defines the number of multiplications over which the precomputation cost is amortized. (Note that it is common in many cryptographic algorithms to perform a large number of consecutive operations using the same modulus.)

The factor e represents the number of precomputation sets to be computed. As written, Algorithm 1 requires one set for the scalar products $\widetilde{AB_i}$ and up

to v sets for the scalar products $\widetilde{Q}\alpha_i$. Note that for the Lookup 2 reduction method v sets need to be computed in step 3.1. For the multiplication and the Lookup 1 reduction methods, the precomputation engine can broadcast a single precomputation set to the relevant processing engines. For the precomputation of a single set, eliminate the loop in step 3.1, compute $\alpha[i] = |i\alpha|_{\hat{M}}$ in step 3.1.1, and compute $\widetilde{Q}\alpha_i = \sum_{j=0}^{v-1} \alpha[\ |ql_{iv+j}|\ ](sign(ql_{iv+j}))2^{uj}$ in step 4.4.

$$T_{MM} = Tb + Tq = (Tb_p + Tb_m) + (Tq_p + Tq_m) \tag{6}$$
$$= \left(\frac{a_b e_b}{c_b}2^{r-1} + b_b n)\right) + \left(\frac{a_q e_q}{c_q}2^{u-1} + b_q(n + d)\right)$$

To determine the optimum number of precomputations it is best to express Equation (6) in terms of $m = \lceil log_2 M \rceil$. Equation (7) provides an approximation, where $n = \lceil m/k \rceil + d + f$, f is a constant and $R = 2^{m+k(d+f)}$. According to Equation (2) and the possible cases in Table 1, $f \in [0, 2]$ when $\mathcal{A} = \mathcal{B} = 2^{k/2}\mathcal{QM}/R$ and the target multiplication accuracy is $S_{n+d+2} \in (-2(\mathcal{QM}/R), 2\mathcal{QM}/R)$. These parameters are of interest here because they define a small number of iterations for Algorithm 1 that generate results suitable for repeated multiplications and they also allow a number of additions to be performed between multiplications without the need for reduction. Unless otherwise specified, this document will assume the use of the aforementioned parameters for general multiplications.

$$T_{MM} = \left(\frac{a_b e_b}{c_b}2^{r-1} + b_b\lceil m/rs \rceil + b_b(d + f)\right) \tag{7}$$
$$+ \left(\frac{a_q e_q}{c_q}2^{u-1} + b_q\lceil m/uv \rceil + b_q(2d + f)\right)$$

6.3 Operations of Interest for Scalar Point Multiplication

Table 2 list some of the operations of interest in the computation of point multiplications. This table assumes that $\mathcal{A} = \mathcal{B} = 2^{k/2}\mathcal{QM}/R$. Entries 1 and 2 in this table are used in the projective coordinate algorithms defined in [18]. Entry 1 corresponds to the classical multiplication operation. Entry 2 defines a division by 2 requiring just $d + 2$ iterations of the loop in Algorithm 1. Entry 3 defines a multiplication of a special form which is used here to reduce the magnitude of a value presumed to be $|0|_M$ before comparing it to zero. Note that for Entry 3, $\mathcal{QM}$ is defined with respect to x $(n = x)$ as shown in Table 1, and this value may be different from the value of $\mathcal{QM}$ used to define $\mathcal{A}$.

Some of the elliptic curve algorithms defined in the open literature, such as the ones in [18], use comparisons in time critical functions, such as point addition and point double. Comparisons are used, among others, to identify the point at infinity during point add and point double operations. These comparisons involve

field elements, therefore numbers A and B are considered equal if $A - B \equiv |0|_M$, which implies that their difference is a multiple of M.

The accuracy of Algorithm 1 is of the order $\mathcal{QM}/R$, where $\mathcal{QM}$ is defined in Table 1. Rather than adding specialized circuitry to perform this function, here we recommend an approach that multiplies a value presumed to be zero by a constant. The idea is to perform this multiplication with high accuracy in a short amount of time. To achieve high accuracy, we recommend the use of Algorithm 1 with low quotient resolution delay ($d \approx 0$) and possibly by using a more exact version of Algorithm 1 (see Table 1). To achieve a short processing time, we recommend multiplication by $\left| 2^{-kx} \right|_{\hat{M}}$ according to Table 2, where the parameter x is adjusted so that the value of the multiplication result is close to the value of M.

The recommended algorithm for the comparison of two field elements A and B works as follows. First compute $A - B$. The result of this operation is a multiple of M if $A \equiv |B|_M$, and, if that is the case, $\left| (A - B)/2^{kx} \right|_{\hat{M}}$ will also be a multiple of M. Then, compute $\left| (A - B)/2^{kx} \right|_{\hat{M}}$ according to Table 2. Finally, refine the result to a value in the range $(-M, M)$ and compare it against 0.

For the comparison of A with zero, assume that $\mathcal{A} = 2^{k/2 + k(d_a + 1)} M$, which could correspond to the Multiplication-based reduction method shown in Table 1, and that the operation $\left| A/2^{kx} \right|_{\hat{M}}$ is done using the Lookup 2 reduction method with $d = 0$ and $x = d_a + 2$, where d_a corresponds to the quotient resolution delay associated with A. For this example, the result of $\left| A/2^{k(d_a + 2)} \right|_{\hat{M}}$ falls in the range $(-(2v+1)M, (2v+1)M)$. This result can be computed with $d_a + 3$ iterations of the loop in Algorithm 1, but because these iterations are computed without quotient resolution delay ($d = 0$) each one can take up to d_a clock cycles. In other words, a multiplier can compute multiplications with and without quotient resolution delay, but when performing operations involving no quotient resolution delay, the multiplier must wait for the quotient resolution. The quotient resolution is assumed to take up to d_a clock cycles.

Note that the algorithm just described is useful for a large set of applications. If additional accuracy is needed for the reduction operations, one can implement in the ECP a more accurate version of the multiplication algorithm, one such algorithm is presented in [9].

Table 2. Multiplications of interest

#	Mult.	B	$\mathcal{AB}$	R	n	$\lvert S_{n+d+2} \rvert$
1	$\left\lvert ABR^{-1} \right\rvert_{\hat{M}}$	B	$< 2^k(\mathcal{QM}/R)^2$	$> (2^k \mathcal{QM})^{1/2}$	$log_{2^k} R$	$< 2\mathcal{QM}/R$
2	$\left\lvert A/2 \right\rvert_{\hat{M}}$	2^{k-1}	$2^{k-1}\mathcal{A}$	2^k	1	$< \mathcal{A}$
3	$\left\lvert A/2^{kx} \right\rvert_{\hat{M}}$	1	$\mathcal{A}$	2^{kx}	x	$< (\mathcal{A} + \mathcal{QM})/2^{kx}$

6.4 Area and Storage

The most complex operation of Algorithm 1 is the computation of the two scalar multiplications $\widetilde{AB}_i$ and $\widetilde{Q\alpha}_i$ – the multiplication in step 5 is just a shift operation. These scalar multiplications can be computed using scalar multipliers. For the computation of a scalar multiplication, a scalar multiplier would add up to $k/2$ numbers per clock cycle when employing Booth recoding and k copies when using no recoding. Assuming that all the operands in Algorithm 1 are of the same size, the concurrent computation of step 4.6 would require the addition of $k+1$ operands when using recoding or $2k+1$ when using no recoding. On the other hand, when using precomputation the concurrent computation of step 4.6 requires the addition of $s+v+1$ operands.

A limiting factor in the practical implementation of multiplication with precomputation is the size of the memory required to store the precomputed values. The use of Booth recoding in Algorithm 1, reduces the memory requirements by half when no storage is provided for values known to have zero value (e.g., $0*A$). Assuming that each precomputed product used in the computation of $\widetilde{AB}_i$ requires $m+k(d+1)+r$-bits of storage, that each precomputed scalar product used in the computation of $\widetilde{Q\alpha}_i$ requires $m+u$-bits of storage, and that each processing unit stores its own set of precomputed values, Algorithm 1 requires $s2^{r-1}(m+k(d+1)+r)+v2^{u-1}(m+u)$ -bits of storage. Note that if multiple reduction methods are used concurrently, such as the use of a reduction method with $d \neq 0$ and one with $d = 0$, more than one copy of reduction coefficients needs to be stored.

Note that the relationship between r and s, and, u and v, allow designers to control the memory size at the expense of processing elements; for example, to achieve a given k, a designer could fix r and then derive s, which defines the required number of processing elements. This approach is particularly attractive for architectures that employ fixed size memory elements, such as field programmable gate arrays (FPGAs).

6.5 Effect of Quotient Pipelining

Quotient pipelining is the technique that allows fast rate of computations by allowing the use of delayed reduction terms ($d \neq 0$). The delay is reflected in steps 4.4 and 4.6 of Algorithm 1. The computation in step 4.4 occurs in the background and takes d iterations to complete. To avoid stalling, the results from step 4.4 are consumed as they become available in step 4.6.

The cost of this technique is reduced accuracy, increased processing time and increased area. The impact of this technique can be reduced by eliminating processing functions associated with the quotients, such as recoding, and by hiding quotient operations behind other functions. For example, the scalar products in step 4.6 of Algorithm 1 could be computed serially with all the processing engines dedicated to the computation of a scalar multiplication, instead of having two sets, each working on a different scalar product.

6.6 Number Representation

The previous discussion considered the upper layers of the ECP architecture, which are independent of the number representation. This section considers the specific example of stored-carry representation.

Stored-carry representation is attractive for the implementation of an ECP, among others, because of its support for fast addition using carry-save addition, natural interaction with non-redundant number representation, and its ability to support two's complement arithmetic.

The main drawbacks of stored-carry representation stem from its representation of a number with two numbers; for example, $A = C + S$, where A, B, and C are numbers of almost equal size. This representation doubles the storage requirements for an operand with respect to non-redundant representation and makes comparisons difficult. A comparison can be carried out by performing a subtraction, converting the result to non-redundant representation and then comparing the result against zero.

The use of Booth recoding in Algorithm 1 alleviates the storage requirements imposed by stored-carry representation. In addition, the ability to amortize precomputations over a large number of operations can be used to reduce memory requirements by storing precomputed values in non-redundant representation. The ECP's multiplier architecture, shown in Figure 2, also makes provisions for the conversion of numbers to non-redundant representation; for example, the conversion of B can be done in a digit-by-digit basis before recoding. In addition, the system could employ a carry propagate adder for the conversion of numbers to non-redundant representation before storing them in the register file.

7 Multiplier Architecture

The AU's architecture is shown in Figure 2. The multiplier and adder together implement Algorithm 1. The adder, which is optimized for accumulation ($A = A + B$), feeds precomputation values to the multiplier. Both the adder and the multiplier receive one of their inputs from the register file. They also output results to the register file.

To accomplish a high rate of computation, the architecture shown in Figure 2 can be implemented using stored-carry representation. To balance storage and processing speed requirements one can choose to represent some numbers in stored-carry representation and others in non-redundant representation.

The reduction terms $\left| i\alpha 2^{uj} \right|_{\hat{M}}$ and some of the temporary results can be converted to non-redundant representation before storage. Operand B of the multiplication can be loaded to the multiplier in stored-carry form and then converted to non-redundant representation one digit at a time as the loop in Algorithm 1 progresses. The reduction terms Q_i can also be converted to non-redundant representation before applying Booth recoding.

To support stored-carry representation, the architecture in Figure 2 must be enhanced with a carry propagate adder and with an efficient way to store

numbers represented in stored-carry representation. For the ECP prototype described in the next section, we implemented a carry propagate adder with a digit-serial adder placed at point **(A)** in Figure 2. For the storage of numbers represented in stored-carry representation, we recommend that the output multiplexer in Figure 2 be able to independently forward to the register file each of the numbers used to represent a number in stored-carry representation; that is, for $A = C + S$, this multiplexer can send either C or S to the register file.

Note that the two numbers used to represent a number in stored-carry representation can be treated as two numbers represented in non-redundant representation. Therefore, for the terms represented in stored-carry representation, such as $A[\ |bl_{is+j}|\ ](sign(bl_{is+j})$, one can use two processing units per term. Where each processing unit handles numbers represented in non-redundant representation. This design approach allows the use of a common processing unit architecture for stored-carry and non-redundant number representations.

8 Prototype Implementation

The validity of the ECP architecture was verified with a prototype that implemented the double-and-add algorithm using the projective coordinates algorithms defined in [18] for point addition and point double operations (the algorithms are shown in the appendix). This prototype was programmed to support the field $GF(2^{192} - 2^{64} - 1)$, which is one of the fields specified in [17].

To verify the ECP's architectural scalability to larger fields, a modular multiplier for fields as large as $GF(2^{521} - 1)$ was also prototyped. This field is the biggest one recommended in [17] for elliptic curves defined over $GF(p)$. This prototype exhibits the same area scalability and frequency of operation as does the multiplier of the ECP prototype. The following discussion focuses exclusively on the ECP prototype.

The ECP prototype used a 16-bit MC processor with 256 words of program memory, a 32-bit AUC processor with 2048 words of program memory, and a dual set of 128 registers, each of which is $m + k(d + 2)$ bits wide. The dual set of registers permits the storage of numbers in stored-carry representation. (Note that a single register set capable of storing stored-carry numbers could have been used instead.) The prototype provided a 32-bit I/O interface to the host system. The ECP multiplier exhibits the following attributes: $s = v = 2$, $r = u = 4$, $k = 8$, $m = 192$, and $d = 4$.

The ECP prototype for $GF(2^{192}-2^{64}-1)$ uses 11,416 LUTs, 5,735 Flip-Flops, and 35 BlockRAMS. LUTs are lookup-tables that are used in the prototype as 16x1-bit RAMs or as 4-input gates. The BlockRAMs are dual-ported 4k-bit blocks of RAM, which are used in the register file, in the MC and AUC as program memory, and in the multiplier as Booth recoders. The frequency of operation of the prototype was 40 MHz. (The frequency of operation of the 521-bit multiplier was 37.3 MHz.)

The validity of the prototype was verified with non-optimized code. Assuming that the ECP is coded in a form that extracts 100% throughput from its

multiplier, it will compute a point multiplication for an arbitrary point on a curve defined over $GF(2^{192} - 2^{64} - 1)$ in approximately 3 msecs ($n = 192/8 + 1$, $d = 4$, $k = rs = uv = 8$) using the algorithms shown in the appendix. This estimate ignores the processing cost of additions and overhead operations and it assumes the computation of $17m$ multiplications per point multiplication: $15.5m$ for the point double and the point add operations and $1.5m$ for the inverse required in the conversion to affine coordinates. For the modular multiplications, this estimate assumes negligible precomputation cost for the reduction terms, $\widetilde{Q\alpha_i}$, and assumes the precomputation of $2^3 - 2$ values for the terms $\widetilde{AB_i}$ (no computation required for $0A$ or $1A$).

The prototypes were implemented using the Xilinix's XCV1000E-8-BG680 (Virtex E) FPGA. The prototypes were coded in VHDL. They were synthesized with Synopsis' FPGA Compiler 3.5.0 and Xilinx's Design Manager M3.1i.

8.1 Comparisons with other Implementations

Table 3 summarizes the features of the multiplier used in the ECP prototype and the features of one of the multiplier architectures introduced in [10] which also relies on precomputation. Both of these multipliers exhibit comparable area requirements (#LUTs), when one assumes $s = v = 1$ and $r = u = 4$. Note that the multiplier in [10] uses a fixed value of k, where this value is highly dependent on the underlying FPGA architecture.

It should be pointed out that the multiplier architecture introduced in [10] can be enhanced with some of the techniques introduced here. For example, to overcome the radix limitation, currently fixed at 2^4, this multiplier could employ multiple processing engines per cell ($s, v \neq 1$), and to reduce memory requirements it could use Booth recoding.

Table 3. ECP multiplier vs. Design 2 multiplier [10]

Characteristic	ECP	Design 2 ($k = 4$)[10]
Type	semi-systolic	systolic
Main Application	Elliptic Curves	Exponentiation
Basic Operation	$\left.\left\|ABR^{-1}\right\|\right._{\hat{M}}$	$\left.\left\|ABR^{-1}\right\|\right._{\hat{M}}$ & $\left.\left\|ACR^{-1}\right\|\right._{\hat{M}}$
Throughput(mult./#clks)	$1/(\lceil m/k \rceil + 2d)$	$2/(2\lceil m/k \rceil)$
Latency (#clks)	$< \lceil m/k \rceil + 2d$	$2\lceil m/k \rceil$
Accuracy	$\leq 2(2^{k(d+1)}M)$	$2(2^k)M$
Max. Radix	2^{rs}	2^k
#LUT	$(2 + 4(2s + v))(m + k(d+1))$	$12m$
# Flip-Flops	$(2 + 2s + v)(m + k(d+1))$	$12m$
Frequency (MHz)	40	48
FPGA	XCV1000E-8-BG680	XC4000

9 Conclusions

This work proposed a new elliptic curve processor architecture for the computation of point multiplication for curves defined over fields $GF(p)$. This processor uses a new type of high-radix Montgomery multiplier that relies on the precomputation of frequently used values and on the use of multiple processing engines.

The ECP's architectural scalability was verified with prototype implementations suitable for the implementation of secure elliptic curve cryptosystems (192- and 521-bits). Our estimates reflect that if were possible to extract 100% throughput from our multiplier, the computation of a point multiplication in a curve defined over $GF(2^{192} - 2^{64} - 1)$ could be computed in about 3 msecs using the double-and-add algorithm and the projective coordinates algorithms defined in [18].

References

1. G. Agnew, R. Mullin, and S. Vanstone, "An implementation of elliptic curve cryptosystems over $F_{2^{155}}$," *IEEE Journal on Selected areas in Communications*, vol. 11, pp. 804–813, June 1993.
2. M. Rosner, "Elliptic curve cryptosystems on reconfigurable hardware," Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, USA, May 1998.
3. L. Gao, S. Shrivastava, and G. Sobelman, "Elliptic curve scalar multiplier design using FPGAs," in *Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)* (C. Koc and C. Paar, eds.), vol. LNCS 1717, Springer-Verlag, August 1999.
4. S. Sutikno, R. Effendi, and A. Surya, "Design and implementation of arithmetic processor $F_{2^{155}}$ for elliptic curve cryptosystems," in *The 1998 IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 647–650, November 1998.
5. K. Leung, K. Ma, W. Wong, and P. Leong, "FPGA implementation of a microcoded elliptic curve cryptographic processor," in *Eight Annual IEEE Symposuium on Field-Programmable Custom Computing Machines, FCCM '00*, (Napa Valley, California, USA), 2000.
6. G. Orlando and C. Paar, "A high performance elliptic curve processor for $GF(2^m)$," in *Workshop on Cryptographic Hardware and Embedded Systems - CHES 2000*, vol. LNCS 1965, (Worcester, Massachusetts, USA), Springer-Verlag, August 2000.
7. P. Kornerup, "A systolic, linear-array multiplier for a class of right-shift algorithms," *IEEE Transactions on Computers*, vol. 43, pp. 892–898, August 1994.
8. M. Shand and J. Vuillemin, "Fast implementations of RSA cryptography," in *Proceedings 11th Symposium on Computer Arithmetic*, pp. 252–259, 1993.
9. H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proceedings 12th Symposium on Computer Arithmetic*, pp. 193–199, 1995.
10. T. Blum, "Modular exponentiation on reconfigurable hardware," Master's thesis, Dept. of ECE, Worcester Polytechnic Institute, Worcester, U.S.A., May 1999.
11. S. E. Eldridge and C. D. Walter, "Hardware implementation of Montgomery's modular multiplication algorithm," *IEEE Transactions on Computers*, vol. 42, pp. 693–699, July 1993.
12. W. Freking and K. Parhi, "A unified method for iterative computation of modular multiplications and reduction operations," in *International Conference on Computer Design (ICCD '99)*, pp. 80–87, 1999.

13. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.
14. I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*. Cambridge, UK: Cambridge University Press, first ed., 1999.
15. P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, April 1985.
16. E. Brickell, D. Gordon, K. McCurley, and D. Wilson, "Fast exponentiation with precomputation," in *Lecture Notes in Computer Science 658: Advances in Cryptology — EUROCRYPT '92*, pp. 200 – 207, Springer-Verlag, Berlin, 1993.
17. F. I. P. S. Publication, "FIPS 186-2: Digital Signature Standard (DSS)," January 2000.
18. P1363, *Standard Specifications for Public-key Cryptography (Draft Version 8)*. IEEE, October 1998.
19. B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*. New York: Oxford University Press, Inc., 1999.
20. I. Koren, *Computer Arithmetic Architectures*. Prentice-Hall, 1993.

A   Elliptic Curve Point Multiplication

Algorithm 2: Double-and-add point multiplication using the projective coordinates algorithms defined in [18]

```
double_and_add(x, y, k)
  (X, Y, Z) = conv_projective(x,y)
  (X_0, Y_0, Z_0) = (X, Y, Z) /* P_0 = P */
  for i = l − 2 down to 0 do
    (X, Y, Z) = double(X, Y, Z) /* P = 2P */
    if k_i = 1 then /* P = P + P_0 */
      (X, Y, Z) = add(X_0, Y_0, Z_0, X, Y, Z)
    end if
  end for
  (x, y) = conv_affine(X, Y, Z)
return (x, y)
```

```
double(X_1, Y_1, Z_1)
  /* if P = O then return O */
  if (X_1, Y_1, Z_1) = O then return(O)
  else /* P ≠ O return 2P */
    M = 3X_1^2 + aZ_1^4
    Z_2 = 2Y_1 Z_1
    S = 4X_1 Y_1^2
    X_2 = M^2 − 2S
    T = 8Y_1^4
    Y_2 = M(S − X_2) − T
  endif
return(X_2, Y_2, Z_2)
```

```
conv_projective(x, y)
return (X = x, Y = y, Z = 1)
```

```
conv_affine(X, Y, Z)
return(x = X/Z^2, y = Y/Z^3)
```

```
add(X_0, Y_0, Z_0, X_1, Y_1, Z_1)
  /* if P_1 = O then return P_0 */
  if (X_1, Y_1, Z_1) = O then
    return(X_0, Y_0, Z_0)
  /* else if P_0 = −P_1 then return O */
  else if (X_0, Y_0, Z_0) = −(X_1, Y_1, Z_1) then
    return(O)
  /* else if P_0 = P_1 then return 2P_0 */
  else if (X_0, Y_0, Z_0) = (X_1, Y_1, Z_1) then
    (X_2, Y_2, Z_2) = double(X_0, Y_0, Z_0)
  else /* return P_2 = P_0 + P_1 */
    U_0 = X_0 Z_1^2
    S_0 = Y_0 Z_1^3
    U_1 = X_1 Z_0^2
    S_1 = Y_1 Z_0^3
    W = U_0 − U_1
    R = S_0 − S_1
    T = U_0 + U_1
    M = S_0 + S_1
    Z_2 = Z_0 Z_1 W
    X_2 = R^2 − TW^2
    V = TW^2 − 2X_2
    2Y_2 = VR − MW^3
  endif
return(X_2, Y_2, Z_2)
```