An Elliptic Curve Cryptosystem Design Based on FPGA Pipeline Folding

Osama Al-Khaleel, Chris Papachristou, Francis Wolff
Case Western Reserve University
Cleveland, Ohio 44106, USA

Kiamal Pekmestzi
National Technical University
157 80 Athens - Greece

Abstract

In this paper we present an efficient design technique for implementing the Elliptic Curve Cryptographic (ECC) Scheme in FPGAs. Our technique is based on a novel and efficient implementation of modular multiplication which is the core operation of ECC. To implement large bit-length multiplications we used a novel partitioning and pipeline folding scheme to fit at least 256-bit modular multiplications on a single Virtex-4 FPGA. Comparisons to several other schemes are presented.

1 Introduction and Motivation

Elliptic curve cryptography (ECC) is very attractive alternative to other cryptography systems, such as RSA, since it offers smaller and incremental key sizes for the same security level. For example, 160-bit and 192-bit ECC keys are considered equivalent to 1024-bit and 2048-bit RSA keys [10]. ECC systems achieve this higher level of security because of the nature of the elliptic curve discrete logarithm problem (ECDLP), which is known to be extremely hard to be solved [4, 7]. This equivalent security with smaller key sizes results in communication bandwidth savings and processing overhead.

The most time consuming operation in the ECC is the Point Scalar Multiplication, $P = kQ$ over a finite field where k is a scalar integer and P & Q are elliptic points. Unfortunately, this multiply can only be done by sequentially adding the point Q in a loop k times: $P = kQ = Q_1 + Q_2 + \cdots + Q_k$. By exploiting a classical shift-add multiply approach then this multiplication can be done by repeated point addition and point doubling, both of which need finite field arithmetic operations [11] and [2]. Fig. 1 shows the hierarchical relationship of the elliptic curve scalar multiplication and their related algorithms are given in later sections of this paper.

The most expensive finite field operation that is needed by point addition and point doubling is the finite field inversion. However, inversion can be transformed by using projective coordinates into less expensive finite field operations, such as finite field addition and multiplication. Of these transformations, the finite field multiplication is the most expensive. To reduce complexity, the finite field multiplication is basically done as a modular multiplication, which involves normal multiplication. Since the cryptosystems use large bit-length operands, for example the keys, a modular multiplier that can multiply two large numbers is needed. This urges the need for designing a large bit-length normal multiplier to be used in the elliptic system.

In this work we present an elliptic curve cryptosystem design technique using FPGAs. We begin with the algorithmic behavior of the point operations Figure 1. Then, we schedule and map these algorithms into a datapath architecture. Scheduling exploits potential parallelism in point operations to improve performance. For example, using two multipliers in parallel improves the efficiency almost twice than using one. Note the modular multiplication dominates the design of elliptic curve cryptosystem in a reconfigurable FPGA. A key contribution is the tradeoff between performance and area of FPGA by using a partitioning and folding pipeline technique on the modular multiplier. This allows the ECC to reside within a single FPGA avoiding multiple FPGA design which reduces performance due to interchip communication of the external pins.

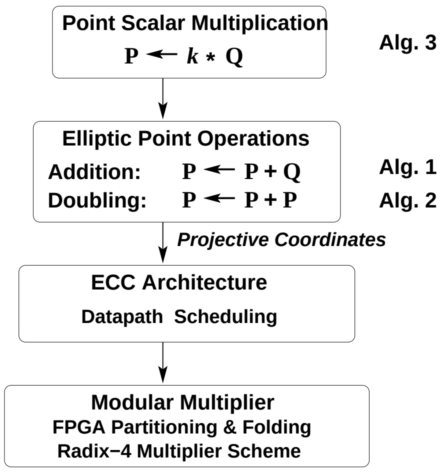

Figure 1. Elliptic curve scalar multiplication hierarchy

Furthermore, we employ a novel radix-4 array multiplication scheme which is the basis for implementing the well-known Montgomery modular multiplier. A key point is that we do not use the builtin multipliers of the Xilinx Virtex-4 FPGA as is commonly done in other works. This turns out to be an advantage when using the folded pipeline technique. Results were compared favorably with respect to the same Xilinx Virtex-II FPGA and a 32-node Beowulf cluster.

2 Elliptic curve cryptosystem (ECC)

In this section, we will first briefly introduce the classical Diffie-Hellman public key process followed the elliptic curve background and point operations.

The encryption and decryption setup process in elliptic curve cryptosystems is shown in Figure 2. We will assume Alice is going to send Bob an encrypted plain-text message, M, using an agreed upon elliptic curve E defined over a finite field $GF(q)$ and a point $Q \in E(GF(q))$.

Alice initially generates a random scalar number k_a and computes a public key $K_a = k_a Q$ using the point scalar multiplication. She keeps her private key k_a secret and shares K_a with Bob as a public key. Elsewhere, Bob generates also a random scalar number k_b and computes $K_b = k_b Q$. He keeps his private key k_b secret and shares K_a with Alice as a public key.

To encrypt the message M, which is represented by a point on the elliptic curve, and send it to Bob, Alice computes the encrypted message (ciphertext) C=(Alice's public key K_a, encrypted message E) where the message point added to the point multiplication of Alice's private key by Bobs public key: $E = M + k_a K_b$. Now, Alice sends to Bob the ciphertext $C = (K_a, E)$. After receiving the encrypted message, the ciphertext, Bob can obtain the original message M by computing the following: $E - k_b K_a \Rightarrow M + k_a K_b - k_b K_a \Rightarrow M + k_a k_b Q - k_b k_a Q \Rightarrow M$.

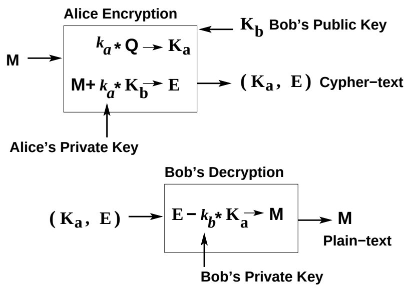

Figure 2. ECC key process

2.1 Background of elliptic curve systems

We now briefly present a mathematical background of cryptosystems based on the elliptic curve theory. For more details, the reader can refer to [4]. The following cubic curve in two variables x and y with odd prime $p > 3$ constitutes an Elliptic curve over $GF(p)$.

$$y^2 = x^3 + ax + b \tag{1}$$

where, $a, b \in GF(p)$ satisfy $4a^3 + 27b^2 \neq 0 \mod p$. The set of solutions, or points $P = (x, y), \forall x, y \in GF(p)$, to equation 1 along with an additional point φ called the point at infinity, forms the elliptic curve $E(GF(p))$ over $GF(p)$. The elliptic curve $E(GF(p))$ with the addition operation$(+)$ form an abelian group. The identity of the addition operation is the point at infinity φ and the inverse of any point is the negative of that point, which is taking the negative of the y-coordinate. For example, the inverse of the point $P(x, y)$ is $(-P(x, y)$ which is $P(x, -y)$.

The addition of two points in $E(GF(p))$ is called $point\ addition$ and is done algebraically as follows:

Assume two points $P = (x_1, y_1) \in E(GF(p))$ and $Q = (x_2, y_2) \in E(GF(p))$, and $P \neq \mp Q$. Then, $R = P + Q \in E(GF(p))$, where $R = (x_3, y_3)$ such that x_3 and y_3 are:

$x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2$, $y_3 = \frac{y_2 - y_1}{x_2 - x_1}(x_1 - x_3) - y_1$

The addition of a point to its self is called $point\ doubling$ and is done algebraically as follows:

Assume a point $P = (x_1, y_1) \in E(GF(p))$ with $y_1 \neq 0$. Then, $R = P + P = 2P \in E(GF(p))$, where $R = (x_3, y_3)$ such that: x_3 and y_3 are given by:

$x_3 = (\frac{3{x_1}^2 + a}{2y_1})^2 - 2x_1$, $y_3 = \frac{3{x_1}^2 + a}{2y_1}(x_1 - x_3) - y_1$

In order to avoid the inversion operation in $GF(q)$, which is modular inversion and usually expensive, different co-ordinates systems are used to replace the inversion with other faster finite field operations. The conversion between the *affine* co-ordinates and the *projective* co-ordinates is defined as follows [3]:

1) From *affine* to *projective*: $(x \rightarrow X, y \rightarrow Y, 1 \rightarrow Z)$

2) From *projective* to *affine*: $(x = \frac{X}{Z^2}, y = \frac{Y}{Z^3})$

2.2 Point operations using projective coordinates

• **Point Addition**. If $P_0 = (X_0, Y_0, Z_0)$ and $P_1 = (X_1, Y_1, Z_1)$ are two different points represented using the projective co-ordinates, then the addition of the two points is a new point, $P_2 = P_0 + P_1$, with new projective coordinates X_2, Y_2, Z_2. The projective coordinates of the new point can be calculated in Algorithm 1. We observe that point addition using projective coordinates in $GF(p)$ requires 16 (modular (finite field) multiplications and 7 modular additions. However, the execution of the algorithm can be parallelized by using two modular multipliers and two add/sub units. This will speed up the point addition by almost 50%. The data flow graph (DFG), assuming two modular multipliers and two modular add/sub units, of the point addition using the projective coordinates in $GF(p)$ is shown in Figure 3.

Algorithm 1: Point addition using projective coordinates

```
Input  : $P_0, P_1$
Output: $P_2 = P_0 + P_1$
$U_0 \leftarrow X_0 Z_1^2$;
$S_0 \leftarrow Y_0 Z_1^3$;
$U_1 \leftarrow X_1 Z_0^2$;
$S_1 \leftarrow Y_1 Z_0^3$;
$W \leftarrow U_0 - U_1$;
$R \leftarrow S_0 - S_1$;
$T \leftarrow U_0 + U_1$;
$M \leftarrow S_0 + S_1$;
$Z_2 \leftarrow Z_0 Z_1 W$;
$X_2 \leftarrow R^2 - TW^2$;
$V \leftarrow TW^2 - 2X_2$;
$2Y_2 \leftarrow VR - MW^3$;
return $X_2, Y_2, Z_2$
```

• **Point doubling**. If $P_1 = (X_1, Y_1, Z_1)$ is a point represented using the projective co-ordinates, then the doubling of the point is a new point, $P_2 = P_1 + P_1$, with new projective coordinates X_2, Y_2, Z_2. The projective coordinates of the new point can be calculated as in Algorithm 2. From Algorithm 2, point addition using projective coordinates in $GF(p)$ requires 10 modular (finite field) multiplications and 4 modular additions. However, the execution of the algorithm can be parallelize by using two modular multipliers and two add/sub units. This will speed up the point doubling by almost 50%. The data flow graph (DFG), assuming two modular multipli-

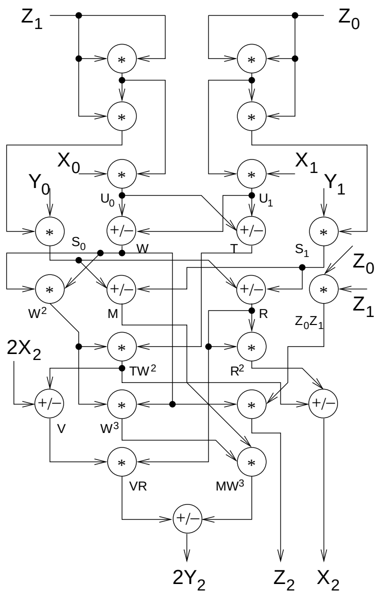

Figure 3. Scheduled Data flow graph of point addition using two multiplier and two add/sub units in $GF(p)$

Algorithm 2: Point doubling with projective coordinates

```
Input  : $P_1, a$
Output: $P_2 = P_1 + P_1$
$M \leftarrow 3X_1^2 + aZ_1^4$;
$Z_2 \leftarrow 2Y_1Z_1$;
$S \leftarrow 4X_1Y_1^2$;
$X_2 \leftarrow M^2 - 2S$;
$T \leftarrow 8Y_1^4$;
$Y_2 \leftarrow M(S - X_2) - T$;
return $X_2, Y_2, Z_2$
```

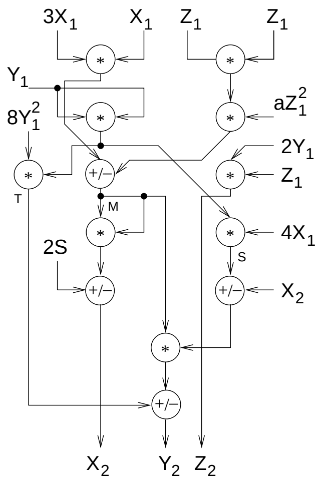

Figure 4. Scheduled Data flow graph of point doubling using two multiplier and two add/sub units in $GF(p)$

Algorithm 3: Point scalar multiplication

```
Input  : A point $Q$, $l$-bit integer
         $k = \sum_{i=0}^{l-1} k_i 2^i, k_i \in \{0, 1\}$
Output: $P = kQ$
$P \leftarrow Q$;
for $i \leftarrow (l-1)$ to $zero$ do
    $P \leftarrow P + P$;
    if $k_i = 1$ then
        $P \leftarrow P + Q$;
return $P$;
```

ers and two modular add/sub units, of the point doubling using the projective coordinates in $GF(p)$ is shown in Figure 4.

• **Point scalar multiplication**. The actual point multiplication $P = kQ$ is done by repeated point addition and point doubling. One efficient method to compute the point multiplication is based on the binary expansion of the integer k. This method is presented in Algorithm 3.

3 Implementations

The basis of our ECC implementation are the previous Algorithms 1, 2, 3. The point scalar multiplication Algorithm 3 performs the actual ECC process (Figs 1, 2) whereas Algorithms 1, 2 perform the detailed operations based on modular addition/subtraction and modular multiplication. Note both Algorithms 1, 2 are very computation intensive mainly because of the modular multiplication. The dataflow behavior of these algorithms can be extracted and scheduled in a number of ways to exploit parallelism, subject to resource constraints. For example, two schedules for these algorithms are shown in Figs. 3, 4 assuming a limitation of 2 modular adders/subtracters and two modular multipliers being used simultaneously. However, maximum parallelism involving upto four multipliers is possible. A baseline datapath architecture that can accommodate the potential of these algorithms for parallelism is illustrated in Fig. 5.

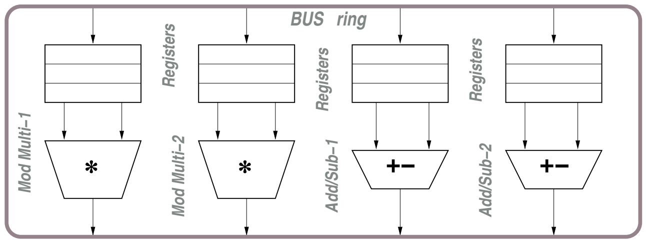

Figure 5. Datapath architecture of the Elliptic cryptosystem

This datapath includes four modular operation units, a two-port register files for each unit, all connected through a bus ring structure. A finite state controller implements the control flow of each point operation algorithm. For faster performance, the bus ring may target the register files individually, incurring additional hardware cost.

We remark that the modular multiplier is by far the most dominant component in Fig. 5 both in delay time and area. Therefore, we focused our work on designing an efficient array multiplication scheme to implement the modular multiplier.

• **Radix-4 multiplication scheme**. We briefly introduce the array multiplier that we used to build the Montgomery modular multiplier. For two k-bit numbers X and Y, by processing 2 bits at a time, their product is

$P = XY = \{2^{k-1}x_{k-1} + 2^{k-2}x_{k-2} + X_{k-3,k-4}\} \times \{2^{k-1}y_{k-1} + 2^{k-2}y_{k-2} + Y_{k-3,k-4}\}$

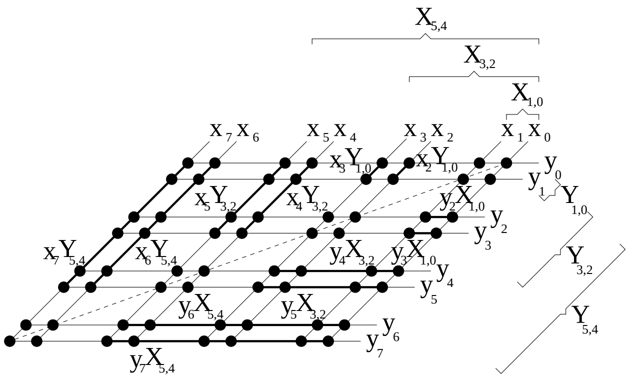

Figure 6. Schematic illustration of the multiplication algorithm

By expanding the above equation we can obtain the mathematical formulas of our array multiplication scheme. This scheme has significant advantages in terms of performance over other array multipliers. The array multiplier is schematically illustrated in Figure 6. The solid lines connect the partial product bits to distinguish the partial product bits group. If we fold the array in Figure 6 along the diagonal, the array multiplication algorithm can be derived.

The detailed derivation of this scheme, including the multiplier cells, is in our earlier work [1]. The basic cells of the array multiplier are shown in Figure 7 and an 8-bit example is built using these cells in Figure 8. We used carry select adders in the bottom stage of the multiplier to add the final partial products.

We used the array multiplier to build a modular multiplier based on the well-known montgomery modular multiplication algorithm [5]. The modular multiplier is embedded within the datapath structure of Fig. 5.

• **Partitioning and folded pipeline**. Partitioning is needed whenever the size of the circuit of the multiplier is large to fit in the FPGA device. The partitioning process is done in a way to have a main partition and other secondary partitions, Fig. 9.

The main partition is reused to implement any secondary partition by reconfiguring the main partition through control signals. The number of partitions depends on the size of the FPGA device being used and the size of the circuit. The size of the main partition should fit within the FPGA resources. The number of partitions are adjusted until the circuit fits in the FPGA device. Reusing the main partition can be done by buffering and feeding back the intermediate outputs to the inputs of the main partition, as shown in Figure 9. Multiplexing circuit should be used to choose between the startup inputs and the intermediate outputs. Also, multiplexing is used, dur-

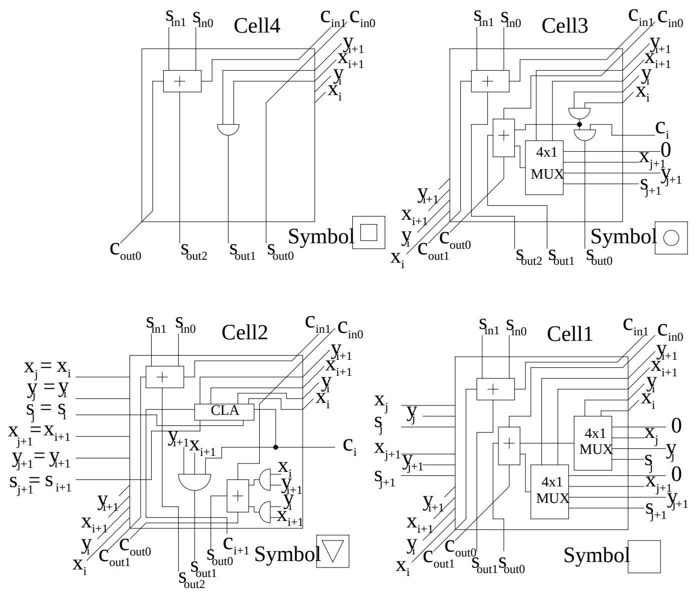

Figure 7. The four basic cells of the array multiplier

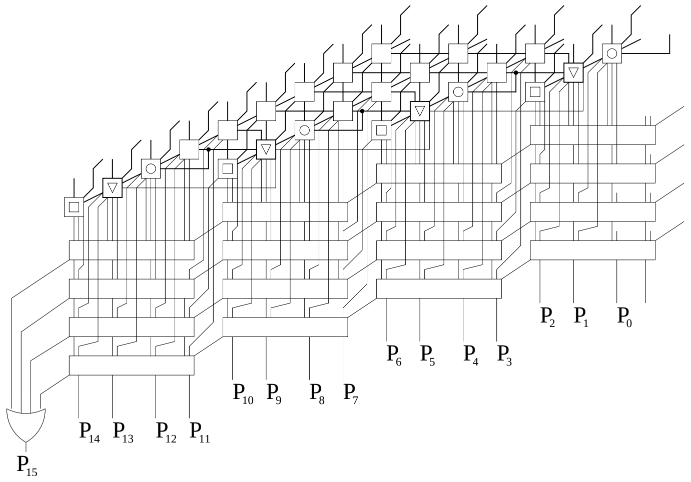
P_{15}
P_{14} P_{13} P_{12} P_{11}
P_{10} P_9 P_8 P_7
P_6 P_5 P_4 P_3
P_2 P_1 P_0

Figure 8. 8-bit array multiplier

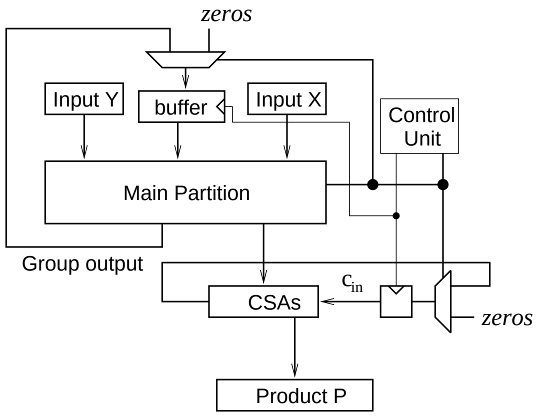

Figure 9. The overall folded pipeline circuit

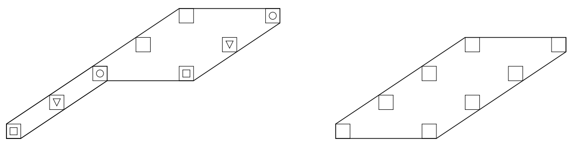
(a) Group A (b) Group B

Figure 10. The grouping

ing the reuse of the main partition to implement a secondary partition, to isolate any unused component. The multiplexing circuits are controlled by configuration signals that are generated by a control unit. The intermediate feedback outputs are registered by clocked registers.

In this work, the targeted FPGA was Virtex-4 FPGA device. We could map the multiplier for up to 128-bit without any need to partition the circuit. However, we had to partition the circuit for higher bit-length. To do this, we first group the basic four cells into two different groups (group A and group B) as shown in Figure 10(a) and 10(b).

We begin with a 32-bit unpartitioned example shown in Figure 11 where the groups, A and B, are represented by symbols. Group A is represented by an empty circle and group B is represented by a full circle. The CSAs and the connections between the groups are not shown for clarity. To demonstrate the partitioning process in this example, the multiplier was partitioned into four partitions. It turned out that the multiplier can easily be partitioned into different partitions, where the secondary partitions are subsets of the main partition. By having the secondary (i.e. second, third and fourth in Figure 12) partitions being subsets of the main partition, the multiplier has the advantage that the main partition can be reused to implement any of the secondary partitions. The control unit sends configuration signals to control the multiplexing circuits. The four partitions, for this 32-bit example, are shown in Figure 12.

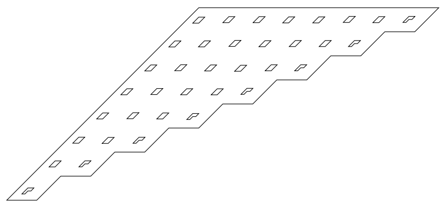
Figure 11. 32-bit example using cell groups

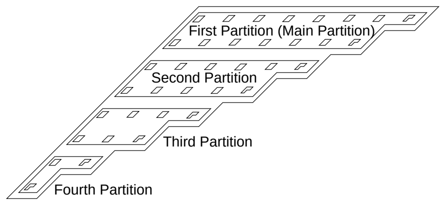

Figure 12. The four partitions of the 32-bit example

The main partition as well as the secondary partitions are implemented in a folded pipeline scheme within the FPGA device. That is, each partition is instantiated by reconfiguring the main partition by means of signals that control the multiplexing circuits. The pipeline folding circuit is in Fig. 9.

n	Folded Part.	Grp. A	Grp. B	FPGA LUTs	Clock MHz
128	1	32	496	31977	16.4
192	8	6	267	15739	83.3
256	4	16	888	65163	32.0

Table 1. Folded pipeline grouping and partition results in Virtex-4

	Single multiplier		Two multipliers	
n	Point Addition	Point Doubling	Point Addition	Point Doubling
128	3.170μs	1.954μs	1.758μs	0.977μs
192	4.720μs	2.880μs	2.610μs	1.705μs
256	6.403μs	4.002μs	3.602μs	2.401μs

Table 2. Virtex-4 time delays using our modular multiplier

4 Results

We implemented all our designs in Xilinx Virtex-II and Virtex-4 devices. We used Synopsys VHDL/Verilog simulators and the Xilinx ISE toolset running on Sun Blade 1000.

Table 1 presents our results on the folded pipeline implementation of the point multiplier design in Virtex-4 FPGA. Column two shows the number of partitions that where folded using the technique discussed in the previous section in order to avoid a multichip FPGA implementation. Columns three and four show the the number of groups related to the radix-4 multiplier. Column five shows the number of FPGA LUTs of the main partition. Column six shows the clock of the resulting folded pipeline. No partitioning was required for 128-bit radix-4 multiplier comfortably fitting within the FPGA. For 192 and 256-bit, it was not possible to fit within a single FPGA. Interestingly, the 192-bit required more folding than the 256-bit multiplier but the faster 192-bit pipeline clock compensated for this due to smaller number of groups than the 256-bit.

We implemented the point doubling operations in Virtex-4 devices. The time delays are shown in Table 2. The table compares the time delays of single versus two modular multiplier implementations. We note for the two modular multiplier implementation case, we mapped the scheduling diagrams of Figs. 3 and 4, into the datapath to parallelize the execution which results in around 50% improvement.

Table 3 shows the point scalar multiplication time delays built upon the elliptic point operations, Algorithm 3. The Virtex-4 results are for both single and double modular multiplier implementions, Figs. 3 and 4. Note that the number of parallel mulipliers has a major impact on performance.

Comparison between time delay of point addition and point doubling, in $GF(p)$, in our case with those in [5] are give in Table 4. Point addition and point doubling are done faster by using the proposed modular multiplier. The comparison was

	Modular Multiplier in [5]			**Folded Modular Multiplier**			**Speedup %**	
n	XC2V FPGA	Point Addition	Point Doubling	XC2V FPGA	Point Addition	Point Doubling	Point Addition	Point Doubling
128	P50-7-ff1517	5.59μs	3.49μs	P50-7-ff1517	4.45μs	2.78μs	25.62%	25.44%
256	P125-7-ff1696	11.53μs	7.20μs	P100-6-ff1704	10.34μs	6.45μs	11.51%	11.63%

Table 4. Comparison of Virtex-II time delays between [5] and our own modular multiplier

n	128	192	256
Single multiplier	0.463ms	1.078ms	1.998ms
Double multiplier	0.253ms	0.624ms	1.161ms

Table 3. Virtex-4 delay of scalar point multiplication

based on Virtex-II Pro FPGA.

We compared our 256-bit point addition results to a multi-processor implementation based on a 32-node Beowulf cluster [9]. The results shown in Table 5 also include their Virtex-II implementation which uses 144 18-bit embedded multipliers. Recall that our implementation does not use any Virtex embedded multipliers. For a fair comparison, we also implemented the 256-bit point addition in the same Virtex-II technology. From Table 5 we can see that the speedup in our case with respect to the Beowulf implementation is 11.2 and the speedup with respect to the FPGA-embedded multiplier of [9] is 1.4. Note also that they would require a two-chip Virtex-II implementation, for point addition and point doubling, respectively, whereas our pipeline folding results in one FPGA chip design.

32-node Beowulf Cluster [9]	Embedded Multiplier [9]	Folded Pipeline in our case
196.72μs	24.56μs	17.512μs

Table 5. Comparison of ECC adder implementation between a 32-node Beowulf cluster & Virtex-II FPGAs

Other FPGA designs of the elliptic cryptosystem architectures are in [6, 8]. Comparison with [6] is difficult because they use the embedded multipliers in a Spartan FPGA, whereas we use the older Virtex-II and do not use any embedded multipliers. However, in terms of performance, we have a speedup of around 27.9 for the 160-bit point scalar multiplication. In terms of resources, we use 23% less slices without embedded multipliers. Reference [8] also does not use embedded multipliers in Virtex-II using instead shift-and-add multiplication. We have a speedup of around 5.3 in comparison to their 192-bit implementation of point scalar multiplication in Virtex-II, however, no complete resource details are included.

We remark that most of the published works use many 18-bit embedded multipliers to construct a single large bit multiplier. The advantage of our pipeline folding scheme, however, is that we can configure several parallel multipliers in the FPGA fabric to increase the performance.

5 Conclusions

We have presented an efficient technique for implementing the Elliptic Curve Cryptographic Scheme in Virtex-II and Virtex-4 FPGAs. Our technique is based on a novel and efficient implementation of modular multiplication which is the core operation of ECC. The main advantages of our technique are: a) Using a radix-4 modular multiplier wich allows more efficient bit processing than radix-2. b) Scheduling and mapping of the elliptic curve algorithms exploiting potential parallelism. c) Using a partitioning and folded pipeline technique on the modular multiplier to tradeoff performance and area. d) Avoiding fixed-width embedded multipliers to allow bit-length flexibility by configuring our multipliers within the FPGA fabric. These advantages when compared to other works resulted in better performance.

References

[1] A large scale multiplier ... *Omitted for Blind Review*, 2006.

[2] S. Devarkal and D. A. Buell. Elliptic curve arithmetic. *Proceedings, MAPLD*, 2003.

[3] IEEE. Standard for public-key cryptography. Nov 1999.

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

[5] C. McIvor, M. McLoone, and J. V. McCanny. FPGA montgomery modular multiplication architectures suitable for ECCs over $GF(p)$. *ISCAS-06*, 3, May 2004.

[6] N. Mentens, K. Sakiyama, L. Batina, I. Verbauwhede, and B. Preneel. Fpga-oriented secure data path design: Implementation of a public key coprocessor. *IEEE Field Programmable Logic and Applications (FPL-06)*, pages 133–138, 2006.

[7] V. Miller. Use of elliptic curves in cryptography. *In H. C. Williams,editor, Advances in Cryptology – CRYPTO '85, Lecture Notes in Computer Science*, pages 417–426, 1986.

[8] M. Morales-Sandoval and C. Feregrino-Uribe. On the hardware design of an elliptic curve cryptosystem. In *5th IEEE Mex. Intern. Conf. in Computer Sceince (ENC'04)*, 2004.

[9] G. Quan, J. P. Davis, S. Devarkal, and D. A. Buell. High-level synthesis for large bit-width multipliers on fpgas: A case study. *CODES+ISSS'05*, pages 213–218, September 2005.

[10] M. Shirase and Y. Hibino. An architecture for elliptic curve cryptograph computation. *SIGARCH Computer Architecture News*, 33(1):124–133, October 2005.

[11] H. Thapliyal and M. B. Srinivas. A high speed and efficient method of elliptic curve encryption using ancient indian vedic mathematics. *MAPLD Intern. Conf.*, September 2005.