High Speed Compact Elliptic Curve Cryptoprocessor for FPGA Platforms

Chester Rebeiro[1] and Debdeep Mukhopadhyay[2]

[1] MS Student, Dept. of Computer Science and Engineering
Indian Institute of Technology Madras, India
rebeiro@cse.iitm.ernet.in
[2] Assistant Professor, Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India
debdeep@cse.iitkgp.ernet.in

Abstract. This paper proposes an efficient high speed implementation of an elliptic curve crypto processor (ECCP) for an FPGA platform. The main optimization goal for the ECCP is efficient implementation of the important underlying finite field primitives namely multiplication and inverse. The techniques proposed maximize the utilization of FPGA resources. Additionally improved scheduling of elliptic curve point arithmetic results in lower number of register files thus reducing the area required and the critical delay of the circuit. Through several comparisons with existing work we demonstrate that the combination of the above techniques helps realize one of the fastest and compact elliptic curve processors.

1 Introduction

Elliptic Curve Cryptography (ECC) provides more security per key bit compared to other security standards. Although fast due to the shorter key size, software implementations of ECC do not meet the high speed required by some networking applications. These applications require ECC to be accelerated by dedicated hardware engines. The most common platform for such hardware accelerators are FPGAs. There are several advantages of using FPGAs for cryptographic applications [21]. Most important is the in-house programmability feature, reconfigurability, low non-recurring costs, simpler design cycles, faster time to market, and greater performance per unit area.

Implementation of ECC follows a layered hierarchical scheme. The performance of the top layers in the hierarchy is greatly influenced by the performance of the underlying layers. It is therefore important to have efficient implementations of finite field arithmetic which form the bottom layer of the hierarchy. Generally for ECC use of prime fields or binary extension fields is recommended. Binary extension fields have the advantage that they have an efficient representation on a computer.

The *elliptic curve crypto processor* (ECCP) proposed in this paper is designed for high speed applications using FPGA as the platform. There are several reported high performance FPGA processors for elliptic curve cryptography

D.R. Chowdhury, V. Rijmen, and A. Das (Eds.): INDOCRYPT 2008, LNCS 5365, pp. 376–388, 2008.

[1][2][3][10]. Various acceleration techniques have been used although most implementations focus on the top layers of ECC. Pipelining and parallelism of the top layers is used to speed up point operations. High speed is also obtained by precomputations, efficient point representations, use of special curves, and efficient instruction scheduling techniques. The ECCP described here achieves high speed by implementing efficient finite field primitives. Additionally the primitives proposed in this paper are optimized for the FPGA platform thus resulting in good area×time product of the processor.

The most important arithmetic primitives for binary finite fields are multiplication and inversion as they occupy the most area and have the longest delay compared to other primitives. Efficient implementation of these primitives would therefore result in an efficient elliptic curve processor. There are several finite field multiplication algorithms that exist in literature. Of these the *Karatsuba* multiplier [7] is one that has sub-quadratic area complexity. It is also shown to be fastest if designed properly [17] [4]. The most common finite field inversion algorithms are based on the *extended Euclidean algorithm* (EEA) and the *Itoh-Tsujii algorithm* (ITA) [6]. Generally the EEA and its variants, the binary EEA and Montgomery algorithms, result in compact hardware implementations, while the ITA is faster. The large area required by the ITA is mainly due to the multiplication unit. ECC requires to have a multiplier present. This multiplier can be reused by the ITA for inverse computations. In this case the multiplier need not be considered in the area required by the ITA. The resulting ITA without the multiplier is as compact as the EEA making it the ideal choice for multiplicative inverse hardware [17].

The elliptic curve processor presented is built with novel multiplication and inversion algorithms. It efficiently implements the elliptic curve operations on the processor so that the scalar multiplication completes in minimum number of clock cycles (given the single finite field multiplier present in the design) and requires the minimum number of register files. The proposed implementation requires two register files, this is lesser compared to [14], which required three register files. The smaller number of register files results in lesser area required. The resulting ECCP is one of the fastest reported and has best area utilization.

The paper is organized as follows: Section 2 has the required background for ECC. Section 3 presents the implementation of the finite field primitives. This section proposes the use of a *hybrid Karatsuba multiplier* and a *quad-Itoh Tsujii inversion algorithm* for FPGA platforms. The 4^{th} section describes the construction of the ECCP, and the 5^{th} section has the comparison of the ECCP with reported works. The final section has the conclusion.

2 Background

The elliptic curve can be represented in *affine coordinates* (2 point system) or *projective coordinates* (3 point system). The projective coordinate representation of an elliptic curve over the field $GF(2^m)$ is given by

$$Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^4 \tag{1}$$

where a and $b \in GF(2^m)$ and $b \neq 0$. The points on the elliptic curve together with the point at infinity ($\mathcal{O}$) form an abelian group under addition.

The operations that are performed on the group are point addition and point doubling. The equation for point addition in *López-Dahab* (LD) projective coordinates [9] for the projective point $P = (X_1, Y_1, Z_1)$ and the affine point $Q = (x_2, y_2)$ is shown in Equation 2. The result is the point on the curve $(P + Q) = (X_3, Y_3, Z_3)$.

$$\begin{gathered} A = y_2 \cdot Z_1^2 + Y_1 \ ; \ B = x_2 \cdot Z_1 + X_1 \ ; \ C = Z_1 \cdot B \\ D = B^2 \cdot (C + a \cdot Z_1^2) \ ; \ Z_3 = C^2 \ ; \ E = A \cdot C \ ; \ X_3 = A^2 + D + E \\ F = X_3 + x_2 \cdot Z_3 \ ; \ G = (x_2 + y_2) \cdot Z_3^2 \ ; \ Y_3 = (E + Z_3) \cdot F + G \end{gathered} \tag{2}$$

The LD projective equation of doubling a point $P = (X_1, Y_1, Z_1)$ is given in Equation 3. The result is the point on the curve $2P = (X_3, Y_3, Z_3)$.

$$\begin{gathered} Z_3 = X_1^2 \cdot Z_1^2 \ ; X_3 = X_1^4 + b \cdot Z_1^4 \\ Y_3 = b \cdot Z_1^4 \cdot Z_3 + X_3 \cdot (a \cdot Z_3 + Y_1^2 + b \cdot Z_1^4) \end{gathered} \tag{3}$$

The cryptographic operation in the ECCP is *scalar multiplication* : Given a basepoint P on the curve and a scalar k, the scalar multiplication determines the product kP. This computation is done using a *double and add algorithm* which traverses the bits of the scalar k and does a point doubling for every bit in k. For every bit set to 1 the point doubling is followed by a point addition.

3 Implementing Finite Field Primitives on an FPGA

Maximizing the performance of the finite field primitives requires the design to be customized for the target hardware. The Xilinx FPGA [22] is made up of *configurable logic blocks* (CLBs). Each CLB on a Xilinx Virtex 4 FPGA contains two slices. Each slice contains two *lookup tables* (LUTs). The LUT is the smallest programmable element in the FPGA. A LUT has four inputs and can be configured for any logic function having a maximum of four inputs. The LUT can also be used to implement logic functions having less than four inputs, two for example. In this case only half the LUT is utilized the remaining part is not utilized. Such a LUT having less than four inputs is an *under utilized LUT*. Most compact implementations are obtained when the utilization of each LUT is maximized. The percentage of under utilized LUTs in a design is determined using Equation 4. LUT_k signifies that k ($1 \leq k \leq 4$) inputs out of 4 are used by the design block realized by the LUT. So, LUT_2 and LUT_3 are under utilized LUTs, while LUT_4 is fully utilized.

$$\%UnderUtilizedLUTs = \frac{LUT_2 + LUT_3}{LUT_2 + LUT_3 + LUT_4} * 100 \tag{4}$$

3.1 Finite Field Multiplication

Finite field multiplication of two elements in the field $GF(2^m)$ is defined as $C(x) = A(x)B(x)modP(x)$, where $A(x)$, $B(x)$, and $C(x) \in GF(2^m)$ and $P(x)$ is the irreducible polynomial of degree m which generates the field $GF(2^m)$. Implementing the multiplication requires two steps. First, the polynomial product $C'(x) = A(x)B(x)$ is determined, then the modular operation is done on $C'(x)$. The Karatsuba algorithm is used for the polynomial multiplication. The Karatsuba algorithm achieves its efficiency by splitting the m bit multiplicands into two 2-term polynomials : $A(x) = A_h x^{m/2} + A_l$ and $B(x) = B_h x^{m/2} + B_l$. The multiplication is then done using three $m/2$ bit multiplications as shown in Equation 5. The three $m/2$ bit multiplications are implemented recursively using the Karatsuba algorithm.

$$\begin{aligned} C'(x) &= (A_h x^{m/2} + A_l)(B_h x^{m/2} + B_l) \\ &= A_h B_h x^m + (A_h B_l + A_l B_h) x^{m/2} + A_l B_l \\ &= A_h B_h x^m + ((A_h + A_l)(B_h + B_l) + A_h B_h + A_l B_l) x^{m/2} + A_l B_l \end{aligned} \tag{5}$$

The basic recursive Karatsuba multiplier cannot be applied directly to ECC because the binary extension fields used in standards such as [19] have a degree which is prime. There have been several published variants of the Karatsuba multiplier for ECC such as the *binary Karatsuba multiplier*[15], the *simple Karatsuba multiplier*[20], and the *general Karatsuba multiplier*[20]. The simple Karatsuba multiplier is the basic recursive Karatsuba multiplier with a small modification. If an m bit multiplication is needed to be done, m being any integer, it is split into two polynomials as in Equation 5. The A_l and B_l terms have $\lceil m/2 \rceil$ bits and the A_h and B_h terms have $\lfloor m/2 \rfloor$ bits. The Karatsuba multiplication can then be done with two $\lceil m/2 \rceil$ bit multiplications and one $\lfloor m/2 \rfloor$ bit multiplication. In the general Karatsuba multiplier, the multiplicands are split into more than two terms. For example an m term multiplier is split into m different terms.

Hybrid Karatsuba Multiplier : The design of the proposed hybrid Karatsuba multiplier is based on observations from Table 1. The table compares the general and simple Karatsuba multipliers for gate counts, LUTs, and percentage

Table 1. Multiplication Comparison on Xilinx Virtex 4 FPGA

n	General			Simple		
	Gates	LUTs	LUTs Under Utilized	Gates	LUTs	LUTs Under Utilized
2	7	3	66.6%	7	3	66.6%
4	37	11	45.5%	33	16	68.7%
8	169	53	20.7%	127	63	66.6%
16	721	188	17.0%	441	220	65.0%
29	2437	670	10.7%	1339	669	65.4%
32	2977	799	11.3%	1447	723	63.9%

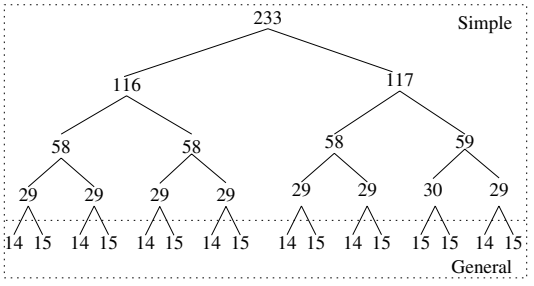

Fig. 1. 233 Bit Hybrid Karatsuba Multiplier

of under utilized LUTs on a Xilinx Virtex 4 FPGA. For the simple Karatsuba multiplier, the percentage of under utilized LUTs is high resulting in bloated area requirements. In the case of the general Karatsuba multiplier, the percentage of under utilized LUTs is low therefore there is better LUT utilization even though the gate count is higher. For $n > 29$, the number of gates in the general Karatsuba multiplier exceeds the benefits obtained by fully utilizing the LUTs resulting in bigger area requirements.

In the hybrid Karatsuba multiplier, all recursions are done using the simple Karatsuba multiplier except the final recursion. The final recursion is done using a general Karatsuba multiplier when the multiplicands have a size less than 29 bits. The initial recursions using the simple Karatsuba multiplier result in low gate count, while the final recursion using the general Karatsuba multiplier result in low LUT requirements. For a 233-bit hybrid Karatsuba multiplier shown in Figure 1, the four initial recursions are done using the simple Karatsuba multiplier, while the final recursion is done with 14-bit and 15-bit general Karatsuba multipliers.

3.2 Finite Field Inversion

The *multiplicative inverse* of an element $a \in GF(2^m)$ is the element $a^{-1} \in GF(2^m)$ such that $a^{-1} \cdot a \equiv 1 \bmod(m)$. From Fermat's little theorem, the multiplicative inverse can be written as $a^{-1} = a^{2^m-2} = (a^{2^{m-1}-1})^2$. The naive technique of computing a^{-1} requires $(m-2)$ multiplications and $(m-1)$ squarings. Itoh and Tsujii in [6] reduced the number of multiplications required by an efficient use of addition chains. An *addition chain* for $n \in \mathbb{N}$ is a sequence of integers of the form $U = (\ u_0\ \ u_1\ \ u_2\ \ \cdots u_r\ )$ satisfying the properties $u_0 = 1$, $u_r = n$ and $u_i = u_j + u_k$, for some $k \leq j < i$. An addition chain for 232 is $U = (\ 1\ \ 2\ \ 3\ \ 6\ \ 7\ \ 14\ \ 28\ \ 29\ \ 58\ \ 116\ \ 232\ )$.

Let $\beta_k(a) = a^{2^k-1} \in GF(2^m)$ and $\beta_{k+j}(a) = (\beta_j)^{2^k}\beta_k = (\beta_k)^{2^j}\beta_j$ [16], then $a^{-1} = (\beta_{m-1}(a))^2$. Using the addition chain shown above, the inverse of an element $a \in GF(2^{233})$ can be determined with 232 squarings and 10 multiplications as shown in Table 2.

Table 2. Inverse of $a \in GF(2^{233})$ using generic ITA

	$\beta_{\mathbf{u_i}}(\mathbf{a})$	$\beta_{\mathbf{u_j}+\mathbf{u_k}}(\mathbf{a})$	Exponentiation
1	$\beta_1(a)$		a
2	$\beta_2(a)$	$\beta_{1+1}(a)$	$(\beta_1)^{2^1}\beta_1 = a^{2^2-1}$
3	$\beta_3(a)$	$\beta_{2+1}(a)$	$(\beta_2)^{2^1}\beta_1 = a^{2^3-1}$
4	$\beta_6(a)$	$\beta_{3+3}(a)$	$(\beta_3)^{2^3}\beta_3 = a^{2^6-1}$
5	$\beta_7(a)$	$\beta_{6+1}(a)$	$(\beta_6)^{2^1}\beta_1 = a^{2^7-1}$
6	$\beta_{14}(a)$	$\beta_{7+7}(a)$	$(\beta_7)^{2^7}\beta_7 = a^{2^{14}-1}$
7	$\beta_{28}(a)$	$\beta_{14+14}(a)$	$(\beta_{14})^{2^{14}}\beta_{14} = a^{2^{28}-1}$
8	$\beta_{29}(a)$	$\beta_{28+1}(a)$	$(\beta_{28})^{2^1}\beta_1 = a^{2^{29}-1}$
9	$\beta_{58}(a)$	$\beta_{29+29}(a)$	$(\beta_{29})^{2^{29}}\beta_{29} = a^{2^{58}-1}$
10	$\beta_{116}(a)$	$\beta_{58+58}(a)$	$(\beta_{58})^{2^{58}}\beta_{58} = a^{2^{116}-1}$
11	$\beta_{232}(a)$	$\beta_{116+116}(a)$	$(\beta_{116})^{2^{116}}\beta_{116} = a^{2^{232}-1}$

Generalizing the Itoh-Tsujii Algorithm: The equation for the square of an element $a \in GF(2^m)$ is given by $a(x)^2 = \sum_{i=0}^{m-1} a_i x^{2i} \bmod p(x)$, where $p(x)$ is the irreducible polynomial. This is a linear equation and hence can be represented in the form of a matrix (T) as shown : $a^2 = T \cdot a$. The matrix depends on the finite field $GF(2^m)$ and the irreducible polynomial of the field. The exponentiation in the ITA is done with squarer circuits. We extend the ITA so that the exponentiation can be done with any 2^n circuit and not just squarers. Raising a to the power of 2^n is also linear and can be represented in the form of a matrix: $a^{2^n} = T^n(a) = T'a$.

For any $a \in GF(2^m)$ and $k \in \mathbb{N}$, define $\alpha_k(a) = a^{2^{nk}-1}$. Using the theorems shown below, we can conclude that any 2^n circuit can be used to implement the Itoh-Tsujii algorithm thus generalizing the algorithm.

Theorem 1. *If* $a \in GF(2^m)$, $\alpha_{k_1}(a) = a^{2^{nk_1}-1}$ *and* $\alpha_{k_2}(a) = a^{2^{nk_2}-1}$ *then* $\alpha_{k_1+k_2}(a) = (\alpha_{k_1}(a))^{2^{nk_2}}\alpha_{k_2}(a)$, *where* k_1, k_2 *and* $n \in \mathbb{N}$.

Theorem 2. *The inverse of an element* $a \in GF(2^m)$ *is given by* $a^{-1} = \left[\alpha_{\frac{m-1}{n}}(a)\right]^2$ *when* $n \mid (m-1)$ *and* $a^{-1} = \left[(\alpha_q(a))^{2^r}\beta_r(a)\right]^2$ *when* $n \nmid (m-1)$, *where* $nq + r = m - 1$ *and* n, q *and* $r \in \mathbb{N}$.

Quad Itoh Tsujii Inversion Algorithm: Consider the case when $n = 2$ such that $\alpha_k(a) = a^{4^k-1}$. To implement this requires quad circuits instead of the conventional squarers. On FPGA platforms, using quad circuits has advantages over squarers. An example of this advantage is shown in Table 3. The table shows the equation of each output bit for an element $b \in GF(2^9)$ for a squarer and a quad circuit and the number of LUTs each circuit takes.

We would expect the LUTs required by the quad circuit be twice that of the squarer. However this is not the case. The quad circuit's LUT requirement is only 1.5 times that of the squarer. This is because the quad circuit has a lower percentage of *under utilized LUTs* (Equation 4).For example, from Table 3 we note

Table 3. Comparison of LUTs required for a Squarer and Quad circuit for $GF(2^9)$

Output bit	Squarer Circuit		Quad Circuit	
	$b(x)^2$	#LUTs	$b(x)^4$	#LUTs
0	b_0	0	b_0	0
1	b_5	0	b_7	0
2	$b_1 + b_5$	1	$b_5 + b_7$	1
3	b_6	0	$b_3 + b_7$	1
4	$b_2 + b_6$	1	$b_1 + b_3 + b_5 + b_7$	1
5	b_7	0	b_8	0
6	$b_3 + b_8$	1	$b_6 + b_8$	1
7	b_8	0	$b_4 + b_8$	1
8	$b_4 + b_8$	1	$b_2 + b_4 + b_6 + b_8$	1
Total LUTs		4		6

Table 4. Comparison of Squarer and Quad Circuits on Xilinx Virtex 4 FPGA

Field	Squarer Circuit		Quad Circuit		Size ratio
	$\#LUT_s$	Delay (ns)	$\#LUT_q$	Delay (ns)	$\frac{\#LUT_q}{2(\#LUT_s)}$
$GF(2^{193})$	96	1.48	145	1.48	0.75
$GF(2^{233})$	153	1.48	230	1.48	0.75

Table 5. Inverse of $a \in GF(2^{233})$ using Quad-ITA

	$\alpha_{\mathbf{u_i}}(\mathbf{a})$	$\alpha_{\mathbf{u_j}+\mathbf{u_k}}(\mathbf{a})$	Exponentiation
1	$\alpha_1(a)$		a^3
2	$\alpha_2(a)$	$\alpha_{1+1}(a)$	$(\alpha_1)^{4^1}\alpha_1 = a^{4^2-1}$
3	$\alpha_3(a)$	$\alpha_{2+1}(a)$	$(\alpha_2)^{4^1}\alpha_1 = a^{4^3-1}$
4	$\alpha_6(a)$	$\alpha_{3+3}(a)$	$(\alpha_3)^{4^3}\alpha_3 = a^{4^6-1}$
5	$\alpha_7(a)$	$\alpha_{6+1}(a)$	$(\alpha_6)^{4^1}\alpha_1 = a^{4^7-1}$
6	$\alpha_{14}(a)$	$\alpha_{7+7}(a)$	$(\alpha_7)^{4^7}\alpha_7 = a^{4^{14}-1}$
7	$\alpha_{28}(a)$	$\alpha_{14+14}(a)$	$(\alpha_{14})^{4^{14}}\alpha_{14} = a^{4^{28}-1}$
8	$\alpha_{29}(a)$	$\alpha_{28+1}(a)$	$(\alpha_{28})^{4^1}\alpha_1 = a^{4^{29}-1}$
9	$\alpha_{58}(a)$	$\alpha_{29+29}(a)$	$(\alpha_{29})^{4^{29}}\alpha_{29} = a^{4^{58}-1}$
10	$\alpha_{116}(a)$	$\alpha_{58+58}(a)$	$(\alpha_{58})^{4^{58}}\alpha_{58} = a^{4^{116}-1}$

that output bit 4 requires three XOR gates in the quad circuit and only one in the squarer. However both circuits require only 1 LUT. These observations are scalable to large fields as is shown in Table 4.

Based on these observations we propose a *quad-ITA* which uses quad exponentiation circuits instead of squarers. The steps involved in obtaining the inverse of an element $a \in GF(2^{233})$ (Table 5) now requires 10 multiplications and 115 quads (compared to 232 quads when squarers are used).

4 Elliptic Curve Crypto Processor

The elliptic curve crypto processor (ECCP) (Figure 2) takes as input a scalar k and produces the product kP, where $P = (P_x, P_y)$ is the basepoint of the curve. The basepoint along with the curve constant is stored in the ROM and

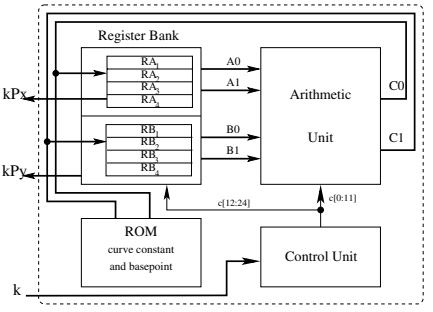

Fig. 2. Block Diagram of the Elliptic Curve Crypto Processor

loaded into registers during initialization. The ECCP implements the scalar multiplication algorithm using the elliptic curve double and add formulae. To be implemented, these formulae require arithmetic operations such as additions, squarings, and multiplications (Figure 3). Of these the hardware for multiplication is the biggest therefore the ECCP can afford to have only one finite field multiplier. Several adders and squarers can be used as they contribute marginally to the latency and area of the processor.

The equation for point addition (Equation 2) has 8 multiplications (assuming a=1) therefore with one multiplier (which is capable of doing one multiplication per clock cycle) it would require a minimum of 8 clock cycles. Similarly point doubling (Equation 3) would require a minimum of 4 clock cycles. The design of the arithmetic unit is optimized so that the addition and doubling takes the minimum required clock cycles.

The arithmetic unit in the ECCP has two outputs. At every clock cycle, at least one of the outputs is derived from the multiplier. This ensures that the multiplier is used in every clock cycle. In order to generate two outputs the

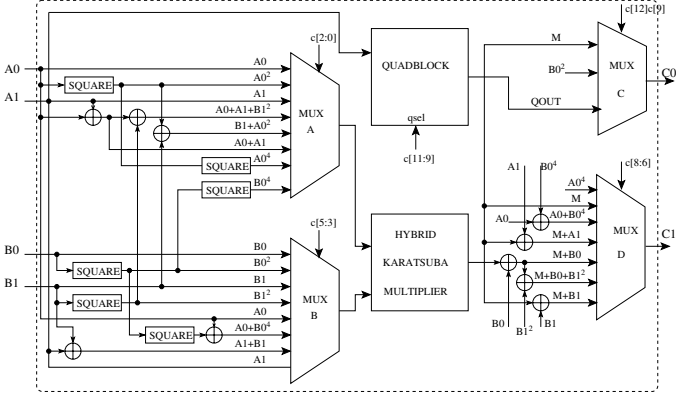

Fig. 3. Arithmetic Unit

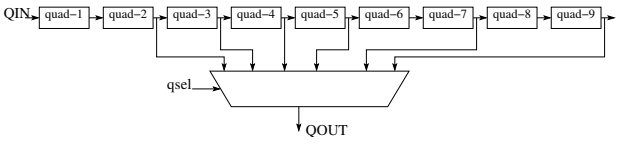

Fig. 4. Quadblock: Raises the Input to the Power of 4^q

arithmetic unit requires at least four input lines. The data on the four input lines is read from registers. The registers are implemented using the FPGA's *dual ported distributed RAM*. Each dual ported RAM has two address lines, two output data lines, and one input data line, therefore to feed the four input arithmetic unit at least two dual ported RAMs (RA and RB) are required. Each dual ported RAM, called a *register file*, implements four 233 bit registers and the two register files are collectively known as a *register bank*.

The *control unit* generates a 25 bit control word every clock cycle. This control word selects the four registers whose data would be read, selects the inputs to the multiplier using multiplexers $MUXA$ and $MUXB$ and selects the output of the arithmetic unit through $MUXC$ and $MUXD$. The control word also determines the register where the result gets written into.

Using LD projective coordinates has the overhead that the result has to be converted from projective to affine coordinates. This requires an inverse to be computed followed by two multiplications. The inverse is computed using the proposed quad-ITA. Obtaining the inverse requires steps in Table 5 to be computed. Each step has a computation of a power followed by a multiplication. The computation of the power has the form α^{4^q}, where q is as large as 58. Computing $\alpha^{4^{58}}$ would require 58 cascaded quad circuits. However this would result in a large latency. An alternate solution is to have $s(< 58)$ cascaded quad circuits. Computing the power α^{4^s} can be done in one clock cycle. Computing α^{4^q} with $q < s$ is done by tapping out the interim result using a multiplexer (Figure 4). Computing α^{4^q} with $q > s$ is done by recycling the result in the quadblock. This would require $\lceil q/s \rceil$ clock cycles. The number of cascades s is the largest number of quads such that the overall delay of the quadblock is less than the longest

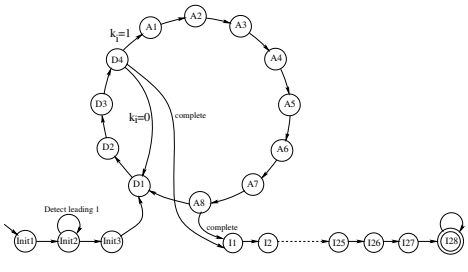

Fig. 5. Finite State Machine

Table 6. Scheduling Point Doubling on the ECCP

Cycle	Operation	AU Inputs				AU Output	
		$A0$	$A1$	$B0$	$B1$	$C0$	$C1$
1	$RA_1 = RB_4^2 \cdot RA_1^2$; $RB_3 = RA_1^4$	RA_1	-	-	RB_4	RA_1	RB_3
2	$RB_3 = RA_2 = RB_3 \cdot RA_3$	-	RA_3	RB_3	-	RA_2	RB_3
3	$RA_3 = (RB_4^4 + RA_2)(RA_1 + RB_1^2 + RA_2)$	RA_2	RA_1	RB_4	RB_1	RA_2	RB_4
	$RB_4 = (RB_4^4 + RA_2)$						
4	$RB_1 = RB_3 \cdot RA_1 + RA_2$	RA_1	RA_2	-	RB_3	-	RB_1

delay in the ECCP, which is through the multiplier. This ensures that the quad-block does not alter the maximum frequency of operation. For $GF(2^{233})$, nine cascades of quad produce the required result.

The finite state machine (FSM) for the ECCP is shown in Figure 5. There are three initialization states ($Init1$ to $Init3$), four states ($D1$ to $D4$) are required for the point doubling, eight states ($A1$ to $A8$) are required for the point addition. At the state $D4$ a decision is made depending on the value of the key bit k_i. If $k_i = 1$, the addition states are entered. If $k_i = 0$ the doubling corresponding to the next bit in k is considered. The final conversion of the result from projective to affine requires 28 states ($I1$ to $I28$). The number of clock cycles required is given by $\#ClockCycles = 3 + 12(h-1) + 4(l-h) + 28$, where l is the length of the scalar k and h its hamming weight.

4.1 Point Equations on the ECCP

The projective double and add equations on the ECCP (Equations 3 and 2) require to be scheduled efficiently in order to minimize the number of clock cycles required. The scheduling ensures that every clock cycle has a multiplication. Table 6 and Table 7 shows the scheduling for point doubling and point addition. They show the operations that are performed at every clock cycle, the data that is driven into the arithmetic unit, and the registers used to store the output of the arithmetic unit. The constant b is assumed to be present in register RA_3. The point $P = (X_1, Y_1, Z_1)$ is present in registers (RB_4, RB_1, RA_1) and another affine point $Q = (x_2, y_2)$ is present in registers (RA_4, RB_2). The point doubling operation computes $P = 2P$ while the point addition operation computes $P = P + Q$. Point doubling taking 4 clock cycles while addition takes 8 clock cycles.

Table 7. Scheduling Point Addition on the ECCP

Cycle	Operation	AU Inputs				AU Output	
		$A0$	$A1$	$B0$	$B1$	$C0$	$C1$
1	$RB_1 = RB_2 \cdot RA_1^2 + RB_1$	RA_1	-	RB_1	RB_2	-	RB_1
2	$RB_4 = RA_4 \cdot RA_1 + RB_4$	RA_1	RA_4	RB_4	-	-	RB_4
3	$RA_2 = RB_3 = RA_1 \cdot RB_4$	-	RA_1	-	RB_4	RA_2	RB_3
4	$RB_4 = RB_4^2(RB_3 + RA_1^2)$	RA_1	-	RB_4	RB_3	-	RB_4
5	$RA_2 = RB_1 \cdot RB_2$	RA_2	-	RB_4	RB_1	RA_2	RB_4
	$RB_4 = RB_1^2 + RB_4 + RB_1 \cdot RA_2$						
6	$RA_1 = RB_3^2$; $RB_3 = RB_4 + RA_4 \cdot RB_3^2$	-	RA_4	RB_3	RB_4	RA_1	RB_3
7	$RB_1 = (RA_4 + RB_2)RA_1^2$	RA_1	RA_4	-	RB_2	-	RB_1
8	$RB_1 = (RA_1 + RA_2)RB_3 + RB_1$	RA_1	RA_2	RB_1	RB_3	-	RB_1

5 Performance Evaluation

In this section we compare our work with reported $GF(2^m)$ elliptic curve crypto processors implemented on FPGA platforms (Table 8). Our ECCP was synthesized using *Xilinx's ISE* for *Virtex 4* and *Virtex E* platforms. Since the published works are done on different field sizes, we use the measure *latency/bit* for evaluation. Here latency is the time required to compute kP. Latency is computed by assuming the scalar k has half the number of bits 1. The only faster implementations are [18] and [3]. However [18] does not perform the final inverse computation required for converting from LD to affine coordinates. Also, as shown in Table 9, our implementation has better area time product compared to [3], while the latency is almost equal. To compare the two designs we scaled the area of [3] by a factor of $(233/m)^2$ since area of the elliptic curve processors is mostly influenced by the multiplier, which has an area of $O(m^2)$. The time is scaled by a factor $(233/m)$ since it is linear.

Table 8. Comparison of the Proposed $GF(2^m)$ ECCP with FPGA based Published Results

Work	Platform	Field m	Slices	LUTs	Gate Count	Freq (MHz)	Latency (ms)	Latency /bit (ns)
Orlando [12]	XCV400E	163	-	3002	-	76.7	0.21	1288
Bednara [2]	XCV1000	191	-	48300	-	36	0.27	1413
Kerins [8]	XCV2000	239	-	-	74103	30	12.8	53556
Gura [5]	XCV2000E	163	-	19508	-	66.5	0.14	858
Mentens [11]	XCV800	160	-	-	150678	47	3.810	23812
Lutz [10]	XCV2000E	163	-	10017	-	66	0.075	460
Saqib [18]	XCV3200	191	18314	-	-	10	0.056	293
Pu [13]	XC2V1000	193	-	3601	-	115	0.167	865
Ansari [1]	XC2V2000	163	-	8300	-	100	0.042	257
Chelton [3]	XCV2600E	163	15368	26390	238145	91	0.033	202
	XC4V200	163	16209	26364	264197	153.9	0.019	116
This Work	XCV3200E	233	18705	36802	306516	28.91	0.065	279
	XC4V140	233	19674	37073	314818	60.05	0.031	124

Table 9. Comparing Area×Time Requirements with [3]

Work	Field (m)	Platform	Slices (S)	Scaled Slices $SS = S(\frac{233}{m})^2$	Latency (ms) (T)	Scaled Latency (ms) $TS = T(\frac{233}{m})$	Area ×Time $(SS \times TS)$
Chelton [3]	163	XC4V200	16209	33120	0.019	0.027	894
This Work	233	XC4V140	19674	19674	0.031	0.031	609

6 Conclusion

In this paper we proposed an elliptic curve crypto processor for binary finite fields. It is compact and one of the fastest implementations reported. High speed of operation is obtained by having a combinational finite field multiplier using the proposed quad Itoh Tsujii algorithm to find the inverse, duplicating hardware units, and efficient implementation of point arithmetic. The ECCP is also compact and has better area×time product compared to the fastest ECCP. Compactness is achieved by the proposed hybrid Karatsuba multiplier and by minimizing the register file requirements in the ECCP.

References

1. Ansari, B., Hasan, M.A.: High Performance Architecture of Elliptic Curve Scalar Multiplication. Technical report, Department of Electrical and Computer Engineering, University of Waterloo (2006)
2. Bednara, M., Daldrup, M., von zur Gathen, J., Shokrollahi, J., Teich, J.: Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. In: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, pp. 157–164 (2002)
3. Chelton, W.N., Benaissa, M.: Fast Elliptic Curve Cryptography on FPGA. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16(2), 198–205 (2008)
4. Grabbe, C., Bednara, M., Shokrollahi, J., Teich, J., von zur Gathen, J.: FPGA Designs of Parallel High Performance $GF(2^{233})$ Multipliers. In: Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS 2003), Bangkok, Thailand, vol. II, pp. 268–271 (May 2003)
5. Gura, N., Shantz, S.C., Eberle, H., Gupta, S., Gupta, V., Finchelstein, D., Goupy, E., Stebila, D.: An End-to-End Systems Approach to Elliptic Curve Cryptography. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 349–365. Springer, Heidelberg (2003)
6. Itoh, T., Tsujii, S.: A Fast Algorithm For Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. Inf. Comput. 78(3), 171–177 (1988)
7. Karatsuba, A.A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Physics Doklady 7, 595–596 (1963)
8. Kerins, T., Popovici, E., Marnane, W.P., Fitzpatrick, P.: Fully Parameterizable Elliptic Curve Cryptography Processor over $GF(2)$. In: Glesner, M., Zipf, P., Renovell, M. (eds.) FPL 2002. LNCS, vol. 2438, pp. 750–759. Springer, Heidelberg (2002)
9. López, J., Dahab, R.: Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 201–212. Springer, Heidelberg (1999)
10. Lutz, J., Hasan, A.: High Performance FPGA based Elliptic Curve Cryptographic Co-Processor. In: ITCC 2004: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004), Washington, DC, USA, vol. 2, p. 486. IEEE Computer Society, Los Alamitos (2004)
11. Mentens, N., Ors, S.B., Preneel, B.: An FPGA Implementation of an Elliptic Curve Processor $GF(2^m)$. In: GLSVLSI 2004: Proceedings of the 14th ACM Great Lakes symposium on VLSI, pp. 454–457. ACM, New York (2004)
12. Orlando, G., Paar, C.: A High Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. In: Paar, C., Koç, Ç.K. (eds.) CHES 2000. LNCS, vol. 1965, pp. 41–56. Springer, Heidelberg (2000)
13. Pu, Q., Huang, J.: A Microcoded Elliptic Curve Processor for $GF(2^m)$ Using FPGA Technology. In: Proceedings of 2006 International Conference on Communications, Circuits and Systems, vol. 4, pp. 2771–2775 (June 2006)
14. Rodríguez, S.M.H., Rodríguez-Henríquez, F.: An FPGA Arithmetic Logic Unit for Computing Scalar Multiplication using the Half-and-Add Method. In: ReConFig 2005: International Conference on Reconfigurable Computing and FPGAs, Washington, DC, USA. IEEE Computer Society, Los Alamitos (2005)
15. Rodríguez-Henríquez, F., Koç, Ç.K.: On Fully Parallel Karatsuba Multipliers for $GF(2^m)$. In: Proc. of the International Conference on Computer Science and Technology (CST), pp. 405–410

16. Rodríguez-Henríquez, F., Morales-Luna, G., Saqib, N.A., Cruz-Cortés, N.: Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials. Des. Codes Cryptography 45(1), 19–37 (2007)
17. Rodríguez-Henríquez, F., Saqib, N.A., Díaz-Pèrez, A., Ķoc, Ç.K.: Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology). Springer-Verlag New York, Inc., Secaucus (2006)
18. Saqib, N.A., Rodríiguez-Henríquez, F., Diaz-Perez, A.: A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication Over $GF(2^m)$. In: Proceedings of 18th International Parallel and Distributed Processing Symposium (April 2004)
19. U.S. Department of Commerce, National Institute of Standards and Technology. Digital signature standard (DSS) (2000)
20. Weimerskirch, A., Paar, C.: Generalizations of the Karatsuba Algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224 (2006)
21. Wollinger, T., Guajardo, J., Paar, C.: Security on FPGAs: State-of-the-art Implementations and Attacks. Trans. on Embedded Computing Sys. 3(3), 534–574 (2004)
22. Xilinx. Virtex-4 User Guide (2007)