HIGH PERFORMANCE ELLIPTIC CURVE CRYPTO-PROCESSOR FOR FPGA PLATFORMS

Chester Rebeiro[1] and Debdeep Mukhopadhyay[2]

Abstract

The paper proposes an efficient implementation of a $GF(2^n)$ Elliptic Curve Processor (ECP) targeted for FPGA platforms. The efficiency is obtained by novel implementations of the underlying finite field primitives required for the ECP. Results are furnished to show that we can realize compact and faster designs compared to existing implementations.

Keywords: Elliptic Curve Cryptography, FPGA, High Performance Implementations

1. Introduction

Elliptic Curve Cryptography (ECC) was invented independently by Koblitz [4] and Miller [6] in 1985. Since then, the security and efficiency of ECC has been accepted, and ECC has been incorporated in several security standards. For a given level of security, the size of the key and the operations involved in ECC computation is shorter than other crypto algorithms. This makes ECC an attractive alternative for today's hand held devices where processing bandwidth, memory resources and power are limited. NIST's standard for Digital Signatures [11] recommends using a prime field or a binary extension field for elliptic curves. Binary extension fields have the advantage that field additions can be performed by XOR operations, therefore no carry is involved. This leads to implementations that require lesser area and have higher performance.

Implementations of Elliptic Curve Cryptosystems follow a layered hierarchical scheme. The performance of the top layers of the hierarchy is greatly influenced by the performance of the underlying layers. It is therefore important to have efficient implementations of finite field operations such as squaring, addition, multiplication and inversion. Among these, finite field multiplication and inversion most critically affect the performance of the Elliptic Curve Cryptosystem.

In the first part of the paper we present our implementations of the important finite field primitives. Our implementation of the finite field multiplier is based on the Karatsuba [2] algorithm. It has the best area time product on an FPGA compared to existing implementations [9] [12]. Our inversion algorithm is a generalization of the Itoh-Tsujii algorithm [1], and computes the inverse in lesser clock cycles compared to the best sequential Itoh-Tsujii implementation [10].

The second part of the paper discusses our implementation of an Elliptic Curve Crypto processor. The implementation is based on the Lopez Dahab projective coordinate system [5] as is done by contemporary works like [8].

1. MS Scholar, Dept. of Computer Science and Engineering, IIT Madras
2. Assistant Professor, Dept. of Computer Science and Engineering, IIT Kharagapur

However our design also includes the final inverse calculation which is not present in [8].

Section 2 reviews Elliptic Curve Cryptography. Section 3 presents our implementation of the Finite Field Primitives used in the Elliptic Curve Processor. The design of the Elliptic Curve processor is discussed in Section 4, while the results are reported in Section 5. Section 6 has the conclusion.

2. Preliminaries

A non super singular Elliptic Curve over the field $GF(2^n)$ is the set of points $(x, y) \in GF(2^n) \times GF(2^n)$ that satisfy the equation

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

where a and $b \in GF(2^n)$. The points on the Elliptic Curve together with the point at infinity (O) form an abelian group under addition. The point O is the identity element of the group. The basic operations that are performed on the group are point addition and point doubling. The equations for the elliptic curve (represented by Equation 1) arithmetic in affine coordinates are shown below. In the table, $P = (x_1, y_1) \neq O$ is a point on the elliptic curve and $Q = (x_2, y_2) \neq O$ is another point on the curve such that $Q \neq P$.

Point Addition $(P+Q)$	$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$ $\lambda = (y_1 + y_2)/(x_1 + x_2)$	Point Doubling $(2P)$	$x_3 = \lambda^2 + \lambda + a$ $y_3 = \lambda(x_1 + x_3) + x_3 + y_1$ $\lambda = x_1 + y_1 / x_1$

The Elliptic Curve Scalar multiplication $(Q = kP)$ is performed by adding P k times over the curve, where P is a point on the curve, called the base point and $k \neq 0$ is a positive integer. The scalar multiplication of the point P is computed using the Algorithm 1.

Algorithm 1: Elliptic Curve Scalar Multiplier

Input : An integer $k \neq 0$ of length l bits and base point P

Output : $Q = kP$

```
1.  begin
2.      Q = O
3.      for i = l − 2 downto 0 do
4.          Q = Double(Q)
5.          if k_i = 1 then
6.              Q = Add(Q, P)
7.          end
8.      end
9.  end
```

The cost of an inversion in affine coordinates is much more expensive than any other field operation. Inversions can be reduced by using a projective coordinate representation. A point P in projective coordinates is represented using three coordinates. In Lopez Dahab (LD) projective coordinates [5] the curve in Equation 1 is transformed to the following.

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

The LD projective coordinates (X, Y, Z) correspond to the affine coordinates $x = X / Z$ and $y = Y / Z^2$. Doubling the point $P = (X_1, Y_1, Z_1)$ results in the point $2P = (X_3, Y_3, Z_3)$.

$$\begin{aligned} Z_3 &= X_1^2 Z_1^2 \\ X_3 &= X_1^4 + bZ_1^4 \\ Y_3 &= bZ_1^4 Z_3 + X_3(aZ_3 + Y_1^2 + bZ_1^4) \end{aligned} \tag{3}$$

When implemented in hardware, these equations can be parallelized to generate the double in three clock cycles [8] as shown below.

Table 1: Parallel Point Doubling

cycle	C_0	C_1
1	$Z_3 = X_1^2 Z_1^2$	$T_1 = Z_1^4$
2	$T_2 = (X_1^4 + T_1)(Z_3 + Y_1^2 + T_1)$	$X_3 = X_1^4 + T_1$
3	$Y_3 = T_1 Z_3 + T_2$	-

Adding two points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, 1)$ in LD coordinates system results in the point $R = (X_3, Y_3, Z_3)$ as shown below.

$A = Y_2 Z_1^2 + Y_1$ $\quad B = X_2 Z_1 + X_1$ $\quad C = Z_1 B$ $\quad D = B^2(C + aZ_1^2)$

$Z_3 = C^2$ $\quad E = AC$ $\quad X_3 = A^2 + D + E$ $\quad F = X_3 + X_2 X_3$

$G = (X_2 + Y_2)Z_3^2$ $\quad Y_3 = (E + Z_3)F + G$

In hardware, the equations for point addition can be parallelized to complete in eight clock cycles [8].

Table 2: Parallel Point Addition

cycle	C_0	C_1
1	$Y_3 = Y_2 Z_1^2 + Y_1$	
2	$X_3 = X_2 Z_1 + X_1$	
3	$T_1 = X_3 Z_1$	
4	$X_3 = X_3^2(Z_1^2 + T_1)$	$Z_3 = T_1^2$
5	$X_3 = Y_3 T_1 + X_3 + Y_3^2$	$T_1 = Y_3 T_1$
6	$T_1 = X_2 Z_3 + X_3$	
7	$Y_3 = (X_2 + Y_2)Z_3^2$	$T_2 = T_3$
8	$Y_3 = (T_2 + Z_3)T_1 + Y_3$	

3. Implementing Finite Field Primitives on a Xilinx FPGA

Maximizing the performance of the finite field primitives requires the design to be customized for the target hardware. The Xilinx FPGA [13] 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 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.

$$\%UnderUtilizedLUTs = \frac{LUT_2 + LUT_3}{LUT_2 + LUT_3 + LUT_4} * 100 \quad (4)$$

3.1. Finite Field Multiplication

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

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

The basic recursive Karatsuba multiplier cannot be applied directly to ECC because the binary extension fields used in standards such as [11] have a degree which is prime. There have been several published works such as the *Binary Karatsuba Multiplier* [9], the *Recursively Applied Iterative Karatsuba* [7], the *Simple Karatsuba Multiplier* [12] and the *General Karatsuba Multiplier* [12]. The Simple Karatsuba Multiplier is the basic recursive Karatsuba multiplier with a small modification. If an n bit multiplication is needed to be done, n being any integer, it is split into two polynomials as in Equation 5. The A_l and B_l terms have $\lceil n/2 \rceil$ bits and the A_h and B_h terms have $\lfloor n/2 \rfloor$ bits. The Karatsuba multiplication can then be done with two $\lceil n/2 \rceil$ bit multiplications and one $\lfloor n/2 \rfloor$ bit multiplication. In the General Karatsuba Multiplier, the multiplicands are split into more than two terms. For example an n term multiplier is split into n different terms.

3.1.1. The Hybrid Karatsuba Multiplier

Our design for the multiplier is based on observations from Table 3.1.1. The table compares the General and Simple Karatsuba multipliers for gate counts

(two input *XOR* and *AND* gates), LUTs required and percentage of under utilized LUTs on a Xilinx Virtex 4 FPGA.

Table 3: 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	2	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%

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. We therefore conclude that the General Karatsuba multiplier is more efficient for small sizes of multiplicands, while the Simple Karatsuba multiplier is efficient for large multiplicands.

In our proposed 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 results in low LUT requirements. For a 233-bit Hybrid Karatsuba multiplier as shown in Figure 1, the initial four recursions are done using the Simple Karatsuba multiplier, while the final recursion is done with 14-bit and 15-bit General Karatsuba multipliers.

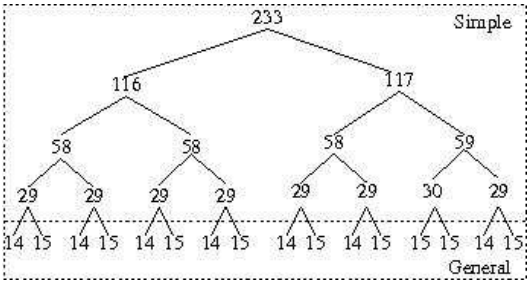

Fig 1: 233 Bit Hybrid Karatsuba Multiplier

3.2. Finite Field Inversion

The *Multiplicative Inverse* of an element $a \in GF(2^n)$ is the element $a^{-1} \in GF(2^n)$ such that $a \cdot a^{-1} \equiv a^{-1} \cdot a \equiv 1 \bmod(n)$. From Fermat's Little Theorem, the multiplicative inverse can be written as $a^{-1} = a^{2^n-2} = (a^{2^{n-1}-1})^2$.

The naive technique of implementing a^{-1} requires $(n-2)$ multiplications and $(n-1)$ squarings. Itoh and Tsujii in [1] reduced the number of multiplications required by an efficient use of addition chains. An *Addition Chain* [3]

for $n \in N$ is a sequence of integers of the form $U = (u_0\ u_1\ u_2 \cdots u_r)$ satisfying the properties $u_0 = 1, u_r = n, u_i = u_j + u_k$ for $k \le j < i$. An addition chain for 232 is given by Equation 7.

$$U = (1\ 2\ 3\ 6\ 7\ 14\ 28\ 58\ 116\ 232) \quad (6)$$

Let $\beta_k(a) = a^{2^k - 1} \in GF(2^n)$ and $\beta_{k+j}(a) = (\beta_k)^{2^j}\ \beta_j$ [10]. If $a \in GF(2^{233})$, then $a^{-1} = (\beta_{232}(a))^2$. Using the addition chain in Equation 7, the inverse of the element *a* can be determined with 232 squarings and 10 multiplications as shown in the Table 4.

Table 4: Inverse of $a \in GF(2^{233})$ using the Itoh-Tsujii Algorithm

	$\beta_{u_i}(a)$	$\beta_{u_j+u_k}(a)$	Exponentiation
1	$\beta_1(a)$		
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	$\beta_{232}(a)$	$\beta_{116+116}(a)$	$(\beta_{116})^{2^{116}}\beta_{116} = a^{2^{232}-1}$

Table 5: Comparison of Squarer and Quad Circuits for $a \in GF(2^{233})$ on Virtex 4

Circuit	Exponentiation	#LUTs	Delay
Squarer	a^2	153	1.483ns
Quad	a^4	230	1.489ns

3.2.1. The Quad-Itoh Tsujii Algorithm

When implemented on an FPGA, there are advantages of using *quad* circuits (i.e. raising *a* to the power of 4) instead of squarers. The Table 5 compares the number of LUTs and the combinational delay for a^2 and a^4. We would expect the area consumed by the *quad* circuit be twice that of the squarer. However this is not the case. The *quad* circuit is about 1.5 times the size of the squarer. Besides this, the combinational delay of the two blocks is the same. This is because the percentage utilization of a LUT for a *quad* circuit is greater than that of a squarer, thus resulting in compact hardware. Based on this observation, we propose a *Quad-Itoh Tsujii* algorithm, which uses quad exponentiation circuits instead of squarers. The *Quad-Itoh Tsujii* algorithm results in lesser number of exponentiations required at a marginal increase in area.

Table 6: Inverse of $a \in GF(2^{233})$ using Quad Itoh Tsujii Algorithm

	$\alpha_{u_i}(a)$	$\alpha_{u_j+u_k}(a)$	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	$\alpha_{116}(a)$	$\alpha_{58+58}(a)$	$(\alpha_{58})^{4^{58}}\alpha_{58} = a^{4^{116}-1}$

Let $\alpha_k(a) = a^{4^k-1}$ and $\alpha_{k+j}(a) = (\alpha_j)^{2^k}\alpha_k = (\alpha_k)^{2^j}\alpha_j$ .Computing the inverse of $a \in GF(2^{233})$ can be done using 116 quad operations and 10 multiplications as shown in Table 6.

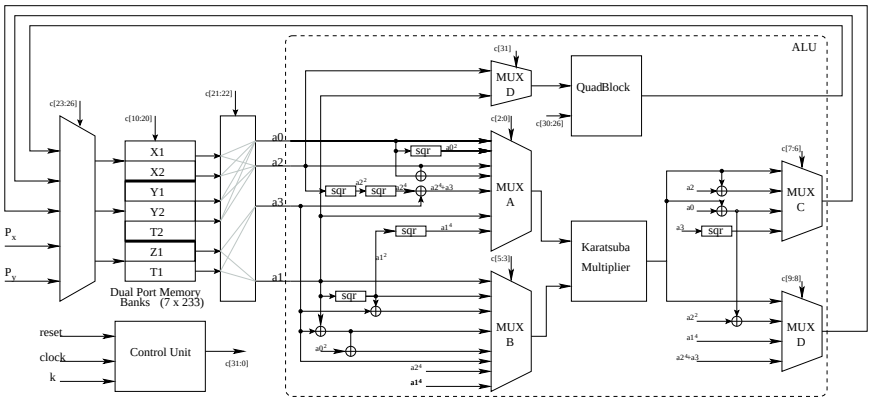

Fig 3: Elliptic Curve Crypto Processor

4. The Elliptic Curve Crypto Processor

This section presents the implementation of the Elliptic Curve Crypto Processor (Figure 2) based on the primitives discussed earlier. The processor consists of three main modules: the ALU, the register bank and the control unit. The inputs to the processor are the base point $P = (P_x, P_y)$ and the key *k*. The output is the scalar product *kP* whose coordinates are stored in the registers X_1 and Y_1 at the end of the computation. The scalar multiplication is computed using Algorithm 1, and the Tables 1and 2. Each equation from the table is evaluated using the ALU and the intermediate results are stored in the register bank.

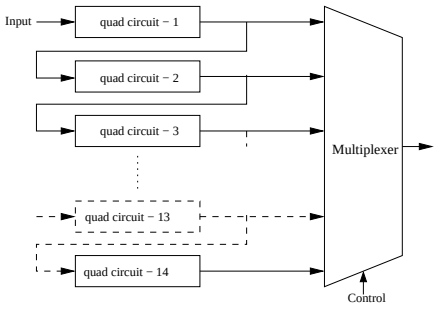

Fig 3: Quad Block: Raises the Input to the Power of 4^k

The ALU : The main part of the ALU is the Quadblock and the Multiplier. The multiplier is based on the Hybrid Karatsuba algorithm and is used in the scalar multiplication as well as during the inversion. The Quadblock is used only during inversion. It consists of cascaded *quad* circuits. There are 14 cascaded circuits as shown in the Figure 3. If the number of quad operations required is less than 14, a multiplexer is used to tap out the interim outputs. In this case the output is obtained in one clock cycle. If the number of quad operations required is greater than 14, the output is recycled in the QuadBlock. Squarer and adder circuits are replicated several times in the ALU to increase throughput. This is possible at minimum resource overhead because of the simplicity of these circuits. The ALU is capable of producing two outputs (C_0, C_1) per clock cycle. Of the two outputs, only one can be from a multiplication due to the single multiplier present. The latency of the entire ALU is mainly due to the multiplier.

The Register Bank: There are seven 233 bit dual port registers configured as three banks. The FPGA's distributed RAM is used for the purpose. The input to the registers is either the base point or the outputs of the ALU or QuadBlock. The outputs of the register are fed to the inputs of the ALU.

The Control Unit: There are 32 control signals (c_0 to c_{32}) that are generated by the control unit at every clock cycle. These signals switch data to : the ALU (c_{21},c_{22}), the inputs to the multiplier (c_0 to c_5), the outputs of the ALU (c_6 to c_9), the control for the Quadblock (c_{26} to c_{31}), the inputs and outputs of the register bank (c_{10} to c_{20} and c_{23} to c_{26}) . The Finite State Machine (Figure 4) has 34 states. The point doubling is implemented with equations from Table 1 and requires three states (*D*1 to *D*3). The point addition is implemented with equations from Table 2 and requires eight states (*A*1 to *A*8). The inverse is calculated using the Quad-Itoh Tsujii (Table 6), and requires twenty one states (*I*1 to *I*21) for completion. The Quad-Itoh Tsujii states entered when the *complete* signal is asserted. The *complete* signal is issued when all key bits in the key k are considered. If the least significant bit (LSB) of k is a zero, then the complete signal is asserted during the D3 state. If the LSB of k is one, the state *A*8 asserts the *complete* signal. The *init* states are required to load the initial values into registers at the cost of 2 clock cycles. The clock cycles for the computation of the scalar multiplication is related to the hamming weight *h* of the scalar *k* and *l*, which denotes the length of the binary string representing *k*.

$$Clockcycles = 1 + 3((l-1)-(h-1)) + 11(h-1) + 21$$
$$= 3(l-h) + 11h + 11$$

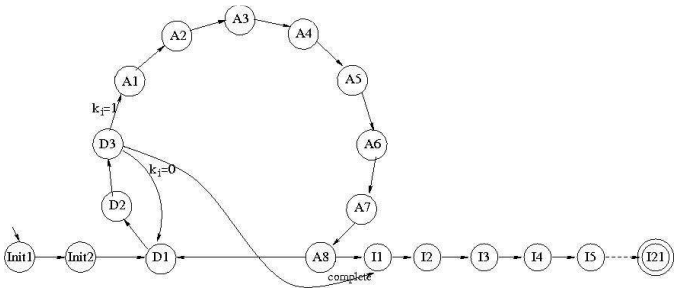

Fig 4: The Finite State Machine for the Elliptic Curve Processor

5. Experimental Results

In this section we present the performance results of our implementation. Our test platform is a Xilinx Virtex 4 FPGA. There are several papers which publish similar works on Elliptic Curve Crypto Processors and primitives. In order to have a uniform platform, we have reimplemented some of the important published works [9] [10] and plugged them into our EC processor to compare results. The Table 7 compares the results of EC processors containing different multiplications and inverse algorithms. The Clock Cycles in the table is the cycles required for one doubling, addition and an inverse. When k=3, we require to perform both doubling and addition. In this case, h=2 and l=2, hence the design for the Hybrid Karatsuba multiplier with Quad-Itoh Tsujii requires 33 clock cycles. The computation for the remaining architectures are similar, keeping in mind that employing the QuadBlock requires 21 clock cycles and the squarer 31 clock cycles for completion.

From the table, the Hybrid Karatsuba based implementations result in the smallest and fastest processors. The Hybrid Karatsuba multiplier saves about 2500 LUTs. The processors with the Quad Itoh Tsujii inversion require the least clock cycles (10 less than a squarer based implementation). The Performance metric η_1 considers the clock cycles. Results show that the processor with the Hybrid Karatsuba and the Quad Itoh Tsujii has the best performance. The Performance metric η_2 does not consider clock cycles. This shows that the combination of a Hybrid Karatsuba multiplier and a Squarer based Itoh-Tsujii has best results.

6. Conclusion

This paper presents an implementation of an Elliptic Curve processor. Novel

Table 7: Comparison of various primitives plugged into our $GF(2^{233})$ ECP

	LUTs	Frequency (f) in MHz	Clock Cycles (CC)	Performance (η_1) $f/LUTs*CC$	Performance (η_2) $f/LUTs$
Hybrid Karatsuba, Quad Itoh Tusjii	34394	37.611	33	33.137	1093
Binary Karatsuba [9] Quad Itoh Tsujii	36970	35.433	33	29.043	958
Hybrid Karatsuba Squarer Itoh Tsujii [10]	33326	37.853	43	26.414	1135
Binary Karatsuba [9] Squarer Itoh Tsujii [10]	35805	35.669	43	23.167	996

techniques for implementing a Karatsuba multiplier and an Itoh Tsujii Inversion algorithm result in efficient implementations of the processor on FPGA platforms. The Hybrid Karatsuba multiplier can be used in Elliptic Curves to minimize the LUTs required and increase the operating frequency. The Quad Itoh Tsujii algorithm can be used to obtain the output with minimum computation time.

References

[1] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF (2^m) using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[2] Anatoly A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[3] Donald E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[4] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[5] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) Without Precomputation. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 316–327, London, UK, 1999. Springer-Verlag.

[6] Victor Miller. Uses of Elliptic Curves in Cryptography. *Advances in Cryptology, Crypto'85*, 218:417–426, 1986.

[7] Steffen Peter and Peter Langendörfer. An Efficient Polynomial Multiplier in $GF(2^m)$ and its application to ecc designs. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1253–1258, San Jose, CA, USA, 2007. EDA Consortium.

[8] Sabel Mercurio Henríquez Rodríguez et. al. 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, 2005. IEEE Computer Society.

[9] Francisco Rodríguez-Henríquez and Çetin Kaya Koç. On Fully Parallel Karatsuba Multipliers for GF(2^m). In *Proc. of the International Conference on Computer Science and Technology (CST)*, pages 405–410.

[10] Francisco Rodríguez-Henríquez et.al, Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials. *Des. Codes Cryptography*, 45(1):19–37, 2007.

[11] U.S. Department of Commerce,National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2000.

[12] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224, 2006.

[13] Xilinx. *Virtex-4 User Guide*, 2007.