ARCHITECTURES FOR ARITHMETIC OPERATIONS IN GF(2^M) USING POLYNOMIAL AND NORMAL BASIS FOR ELLIPTIC CURVE CRYPTOSYSTEMS

Renuka H. Korti[1], Dr. Vijaya C.[2]
[1]Assistant Prof. Dept of E & C, SDMCET., [2]Professor, Dept of E & C, SDMCET, Dharwad, India.
E-mail: rhh_korti@yahoo.com[1], vijayac26@yahoo.com[2]

Abstract
Elliptic Curve Cryptography (ECC) fits well for an efficient and secure encryption scheme. It is efficient than the ubiquitous RSA based schemes because ECC utilizes smaller key sizes for equivalent security. This feature of ECC enables it to be applied to Wireless networks where there are constraints related to memory and computational power. Fast and high-performance computation of finite field arithmetic is crucial for elliptic curve cryptography (ECC) over binary extension fields. Two of the most common basis used in binary fields are polynomial basis and normal basis. The normal basis is especially known to be more efficient than polynomial basis because the inversion can be achieved by performing repeated multiplication and squaring can be executed by performing only one cyclic shift operation. This research paper deals with implementing and evaluating the finite field arithmetic operation algorithms using both polynomial basis (PB) and normal basis (NB) representations. The Normal basis implementation performs better than the polynomial basis representation in terms of area and speed.

Key words: cryptography, elliptic curve cryptography, finite field, polynomial basis, normal basis

I. Introduction

Modern cryptography provides essential techniques for securing information and protecting data. The arithmetic operations in the Galois field GF(2^m) have several applications in coding theory such as BCH codes and Reed Solomon error correction, computer algebra, and cryptography algorithms such as the Rijndael encryption algorithm and Elliptic Curve Crytography. In these applications, time and area efficient algorithms and hardware structures are desired for addition, multiplication, squaring, and inversion operations. The performance of these operations is closely related to the representation of the field elements. Arithmetic in a finite field is different from standard integer arithmetic. There are a limited number of elements in the finite field; all operations performed in the finite field result in an element within that field. Finite fields are used in a variety of applications, including in classical coding theory in linear block codes such as BCH codes and Reed Solomon error correction and in cryptography.

The Elliptic Curve cryptographic system has been proven to be stronger than known algorithms like RSA/DSA. The efficiency of the core Galois field arithmetic improves the performance of elliptic curve based public key cryptosystem implementation.

ECC uses a binary field GF(2^m) or a prime field GF(p). The encryption and decryption speed is an important indicator for evaluating an ECC algorithm. Efficiency of finite field arithmetic operation has great impact on the performance of an ECC, since an ECC computation consists a set of point operations and field multiplication and field inversion are the basic operations involved in the point operation.

The binary field $GF(2^m)$ is widely used in field operations because it is very suitable for VLSI implementation.

The finite field $GF(2^m)$ is a number system containing 2^m elements. Its attractiveness in practical applications stems from the fact that each element can be represented by m binary digits. The practical application of error-correcting codes makes considerable use of computation in $GF(2^m)$. Recent advances in secret communication, such as encryption and decryption of digital messages, also require the use of computation in $GF(2^m)$ [4]. Hence, there is a need for good algorithms for doing arithmetic operations in finite field. The most commonly used basis are polynomial basis(PB) and normal basis (NB)[2][3]. Normal basis [4] is more suitable for hardware implementations than polynomial basis since operations in normal basis representation are mainly comprised of rotation, shifting and exclusive-ORing which can be efficiently implemented in hardware. These operations are implemented on FPGA Spartan3 tool & simulated using verilog on Xilinx 14.5.

II. GF(2^m):

The most commonly used non modular finite field in cryptographic applications is the Galois field 2^m. The efficiency of finite field arithmetic operations in $GF(2^m)$ is deeply relied on how elements are represented. There exist many algorithms for representing elements and computing operations in this field very efficiently. $GF(2^m)$ is called a binary finite field because it can be represented in its multiplication table as {0, 1}m elements. Different algorithms make use of this binary format to manipulate numbers the fastest. For example, addition in this field is nothing more than XORing the array representation of field elements. Another way of representing $GF(2^m)$ is through a polynomial or normal basis.

Polynomial Basis Representation:

For the polynomial basis representation, each element of the field represents a polynomial,

a(z) = am-1zm-1 +….+ a2z2 + a1z1+ a0z0

is associated with the binary vector a = (am-1,a2, a1, a0) of length m.

Therefore each operation, such as addition, subtraction, multiplication and inversion are defined using polynomial arithmetic with the coefficients reduced modulo 2. For example, the bit sequence 01100101 would represent the polynomial: $x^6 + x^5 + x^2 + 1$.

Let t = m/ W, and let s = Wt - m. In software, a may be stored in an array of t W-bit words:

Normal Basis Representation:

It is well known that there always exists a normal basis in the finite field $GF(2^m)$ for all positive integers m. For an α € $GF(2^m)$,{ $\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{(m-1)}}$ } is called a normal basis of $GF(2^m)$ over GF(2) if $\alpha, \alpha^2, \alpha^4, \ldots,$ and $\alpha^{2^{(m-1)}}$ are linearly independent. A normal basis always exists in the finite field $GF(2^m)$ for all positive integers m. Every element $A \in GF(2^m)$ can be represented as

$A = a_0\alpha^{2^0} + a_1\alpha^{2^1}, + a_2\alpha^{2^2} + \ldots\ldots\ldots + a_{m-1}\,\alpha^{2^{m-1}}$ ....(1)

Where $a_i \in \{0,1\}$ for i=0,1,2,....,m-1.

If $0 \leq i_1, i_2 \leq m-1$ and $i_1 \neq i_2$,there exists j_1,j_2 such that $A^{2^{i_1}+2^{i_2}} = A^{2^{j_1}+2^{j_2}}$ the basis is called optimal. There are two types of commonly used optimal normal basis (ONB) which can be defined as:

(1) Type-I ONB :$m+1$ is a prime p, and 2 is a primitive modulo p.
(2) Type-II ONB: $2m+1$ is a prime p and either
   (a) 2 is primitive modulo p, or
   (b) $p \equiv 3$ (mod 4) and the multiplicative order of modulo p is m.

Type-1 ONB is used in the proposed work.

III. Polynomial basis arithmetic:

A. Addition

Addition of Galois field elements is performed by bitwise XOR operation, thus requiring only t word operations[1].

Algorithm: Addition in $F2^m$

Input: Binary polynomials a(z) and b(z) of at most m - 1.

Output: $c(z) = a(z) + b(z)$.

1. **for** i from 0 to t – 1 **do**
   1.1 $C[i] \leftarrow A[i] \oplus B[i]$.
2. **return** (c).

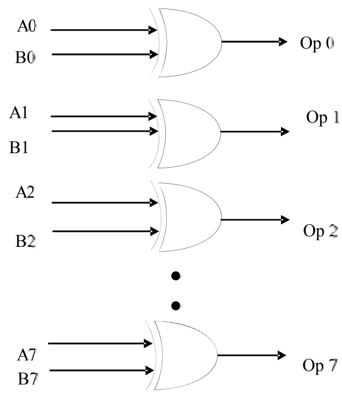


Fig 1 : GF adder circuit F_2^8

B. Multiplication:

Multiplication is the most important arithmetic operation and more time consuming than addition, subtraction and squaring. The multiplier of Finite field based on Karatsuba's divide and conquer algorithm:

The product of a(z) and b(z) is

$a(z)b(z) = (A_1 z^l + A_0)(B_1 z^l + B_0) = A_1B_1z^{2l} + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^l + A_0B_0$ where $l = m/2$ and the coefficients A_0, A_1, B_0, B_1 are binary polynomials in z of degree less than l[1][2].

Algorithm: Binary Karatsuba multiplier for arbitrary m

Input: Two elements A, B GF (2^m) with m an arbitrary number, and where A and B can be expressed as $A = X^{m/2} A^H + A^L$, $B = X^{m/2} B^H + B^L$

Output: A polynomial C = AB with up to 2m-1 coordinates, Where $C = X^m C^H + C^L$.

1. **Procedure** *BK (C, A, B)*
2. begin
3. $k = [log_2 m]$
4. $d = m - 2^k$;
5. **if** $(d == 0)$ **then**
6. $C = K\, mul2^k (A, B)$
7. **return**;
8. **for** i from 0 to d-1 **do**
9. $M_{Ai} = A_i^L + A_i^H$;
10. $M_{Bi} = B_i^L + B_i^H$;
11. **end for**
12. $mul2^k (C^L, A^L, B^L)$;
13. $mul2^k (C^L, A^L, B^L)$;
14. $BK (C^H, A^H, B^H)$;
15. **for** i from 0 to 2k - 2 **do**
16. $M_i = M_i + C_i^L + C_i^H$;
17. **end for**
18. **for** i from 0 to 2k - 2 **do**
19. $C_{k+i} = C_{k+i} + M_i$;
20. **end for**
21. **for** i from 0 to 2k - 2 **do**
22. $C_{k+i} = C_{k+i} + M_i$;
23. **end for**
24. **end if**
25. **end**

C. Squaring

Squaring a binary polynomial is a linear operation, it is much faster than multiplying two arbitrary polynomials[1][3].

Assume the binary polynomial is $a(x) = \sum_{i=0}^{m-1} a_i x_i$ then the squaring formula can be calculated using equation

$$a(x)^2 = \sum_{i=0}^{2(m-1)} a_i x_i^2 .$$

i.e., if $a(z) = a_{m-1}z^{m-1} + \ldots + a_2z^2 + a_1z^1 + a_0$,
then $a(z)^2 = a_{m-1}z^{2m-1} + \ldots + a_2z^4 + a_1z^2 + a_0$

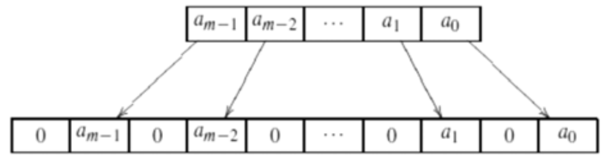


Fig.2: Squaring a binary polynomial $a(z) = a_{m-1}z^{m-1} + \ldots + a_2z^2 + a_1z^1 + a_0$

Algorithm: Polynomial squaring (with word length W = 32)

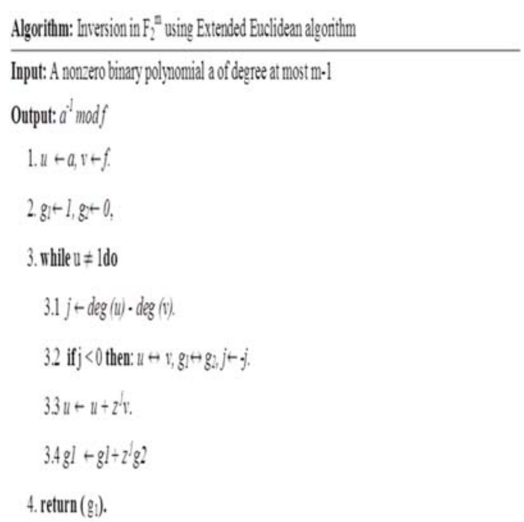
Input: A binary polynomial a(z) of degree at most m-1.

Output: c(z) = a(z)2.

1. Pre computation: For each byte d = (d7,...,d1 ,d0), compute the 16-bit quantity
2. for i from 0 to t – 1 do
   2.1 Let A[i]=(u3 , u2 ,u1 ,u0) where each u j is a byte.
   2.2 C [2i]←(T(u1), T (u0)), C[2i + 1]←(T(u3), T (u2)).
3. return(c).

D. Inversion:

The inverse of a nonzero element a in GF(2^m) is the unique element g ε GF(2^m) such that ag = 1 in GF(2^m), that is, ag = 1 (mod f). This inverse element is denoted as a^{-1}[1][3].



Algorithm: Inversion in F_2^m using Extended Euclidean algorithm

Input: A nonzero binary polynomial a of degree at most m-1

Output: $a^{-1} \bmod f$

1. $u \leftarrow a, v \leftarrow f$.
2. $g_1 \leftarrow 1, g_2 \leftarrow 0$,
3. **while** $u \neq 1$ **do**
   3.1 $j \leftarrow deg(u) - deg(v)$.
   3.2 **if** $j < 0$ **then:** $u \leftrightarrow v$, $g_1 \leftrightarrow g_2$, $j \leftarrow -j$.
   3.3 $u \leftarrow u + z^j v$.
   3.4 $g1 \leftarrow g1 + z^j g2$
4. **return** (g_1).

IV. Normal Bases Arithmetic:

A. Addition:

Addition is the bitwise exclusive-or (XOR) of all bits of the two addends. This operation is identical to polynomial basis addition.

B. Squaring:

Squaring is a bitwise cyclic rotation . Addition and squaring are the two simplest and fastest operations in normal basis.

Thus, if A=(a0,a1,a2,…,am-1) then A^2 =(am-1,a0,a1,…,am-2)

Hence, squaring in GF(2^m) can be realized physically by logic circuitry which accomplishes cyclic shifts in a binary register. Such squaring circuitry is illustrated in block form in Fig. 1.

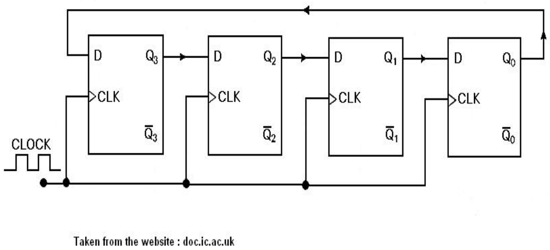
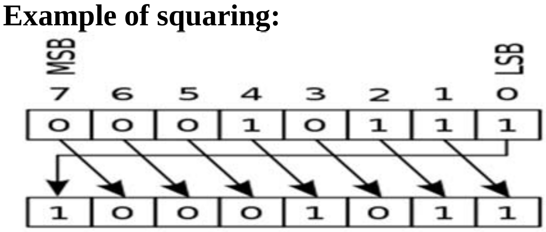


Fig.3: 4-bit Squaring circuitry:

Example of squaring:



C. Multiplication:

Let x and y be the two elements in GF(2^m) which can be expressed as

X=$x_0\alpha^{2^0}+x\alpha^{2^1},+x_2\alpha^{2^2}+\ldots\ldots\ldots+x_{m-1}\,\alpha^{2^{m-1}}$ ……………………………………………………………(2)

Y=$y_0\alpha^{2^0}+y_1\alpha^{2^1},+y_2\alpha^{2^2}+\ldots\ldots+y_{m-1}\,\alpha^{2^{m-1}}$ ……………………………………………………………(3)

The product of both X and Y can be defined as

Z= X * Y ………………………………….(4)

Squaring in the normal basis can be performed by cyclically shifting the elements in GF(2^m) but the multiplication is complex when compared to other basis. Hence in order to perform the multiplication in the normal basis conversion of the basis is needed. The normal basis N can be expressed as

N= $\alpha^{2^0}+\alpha^{2^1},+\alpha^{2^2}+\ldots+\alpha^{2^{m-1}}$……………..(5)

Let us consider the generating polynomial G(X), where G(X) is an irreducible All-One-Polynomial of degree m and m+1 represents relative prime 2. G(X) can be expressed as

G(x)=1+X^1+X^2+…….+X^{m-1}……………………(6)

α denotes the root of G(X) it satisfies the property α^{m+1}= 1. If α^{m+1}= 1, then the normal basis N can simply be transformed to the following shifted standard basis N':

N'={α^1,α^2,α^3,…….,α^m}……………………………(7)

Defining the conversion of permutation P from the basis N to N'. The Permutation P is also performed for X and Y and it can be expressed as

X= $x_0\alpha^{2^0}+x_1\alpha^{2^1},+x_2\alpha^{2^2}+\ldots+x_{m-1}\,\alpha^{2^{m-1}}$…. ……………………………………………(8)

=$x'_1\alpha^1+x'_2\alpha^2+x'_3\alpha^3+\ldots+x'_m\alpha^m$ ……….(9)

Y=$y_0\alpha^{2^0}+y_1\alpha^{2^1},+y_2\alpha^{2^2}+\ldots+y_{m-1}\,\alpha^{2^{m-1}}$.. ……………………………………………(10)

=$y'_1\alpha^1+y'_2\alpha^2+y'_3\alpha^3+\ldots+y'_m\alpha^m$…………(11)

Where

For i=0,1,2,…….,m-1 and j=1,2,3,…..,m,

$x'_j = x_i$,

$y'_j = y_i$,

And j= 2^i mod(m+1).

X and Y can be represented by the shifted standard basis, the product Z of X and Y can be expressed as

Z = X * Y

=($x'_1\alpha^1+x'_2\alpha^2+x'_3\alpha^3+\ldots+x'_m\alpha^m$)*B…..(12)

= $x'_1\alpha^1$ B+$x'_2\alpha^2$B+$x'_3\alpha^3$B+…….+$x'_m\alpha^m$ B

Each term in the above equation (12) can be expanded and each term of the product Z can be calculated using

z'_0=($x'_0y'_0+x'_1y'_m+x'_2y'_{m-1}+x'_2y'_{m-2}+\ldots+x'_my'_1$) mod 2

$z'_1=(x'_0y'_1+x'_1y'_0+x'_2y'_m+x'_2y'_{m-1}+......+x'_my'_2)$ mod 2
$z'_2=(x'_0y'_2+x'_1y'_1+x'_2y'_0+x'_2y'_m+......+x'_my'_3)$ mod 2
....
$z'_{m-1}=(x'_0y'_{m-1}+x'_1y'_{m-2}+x'_2y'_{m-3}+.......+x'_{m-1}y'_0+x'_my'_{m-1})$ mod 2
$z'_m=(x'_0y'_1+x'_1y'_{m-1}+x'_2y'_{m-2}+.......+x'_{m-1}y'_1+x'_my'_0)$ mod 2...........................(13)

Based on the generating polynomial G(X) expressed in the equation (6) we have

$1+\alpha+\alpha^1+\alpha^2+.......+\alpha^m=0$,
$1=\alpha+\alpha^1+\alpha^2+.......+\alpha^m$………....................(14)

If z'_1=1,then the polynomial can be summed in to Z and is expressed as
For i = 1,2,.....,m,

$$z'_i=\begin{cases} z'_i+1 \bmod 2 & if\ z'_0=1 \\ z'_i & if\ z'_0=0 \end{cases}$$

At the end inverse permutation p^{-1} is performed in order to transform the shifted standard basis N' into original normal basis N as follows

$z_i=z'_i$ and
$i=2^j$ mod (m+1) for i = 1,2,....,m, and
j = 0,1,2,.....,m-1.

The final result Z can be calculated as

$Z=z_0\alpha^{2^0}+z_1\alpha^{2^1},+z_2\alpha^{2^2}+.....+z_{m-1}\alpha^{2^{m-1}}$......
....(15)

Fig.4 illustrates the hardware implementation of the proposed algorithm. Permutations *P*1 and *P*2 belong to permutation *P*, and permutation *P*3 belongs to the inverse permutation *P*-1. The functions of *P*1, *P*2 and *P*3, each with m inputs and m outputs are defined by
Permutations p1 and p2 with inputs I_j and outputs O_i
$O_i=I_j$
$I=2^j$ mod(m+1) for i = 1,2,3......,m and j = 0,1,2,...... ,m-1
Inputs for p1 are given as
$I_i=b_i$ where,$0\le i\le m-1$
Outputs for *P*1 are given by
$b_{i'}=O_i$ where, $1\le i\le m$
apply $b_{0'}=0$ and b_0 directly to flip flop D_0
Inputs for *P*2 are given as
$I_i=a_i$ where, $0\le i\le m-1$

Outputs from *P*2 are given as
$a_i'=O_i$ where, $1\le i\le m-1$
apply $a_0=0$ and a_0 directly in to s_0

Inverse permutation p3 with inputs I_i and outputs O_j
$O_j=I_i$
$j=2^i$ mod(m+1)

The final result *C* is obtained through permutation *P*3. The proposed normal basis multiplier needs m+1 2-input AND gates, 2m+1 2-input XOR gates and 3m+3 1-bit flip-flops. The proposed sequential normal basis multiplier is regular and expandable, and is therefore naturally suited to VLSI implementation[4][6][7].

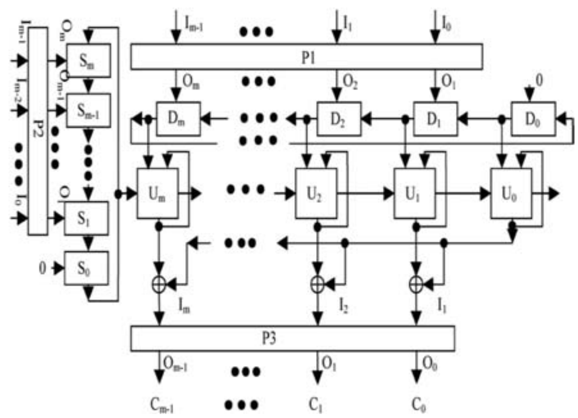


Fig. 4. The proposed normal basis multiplier in GF(2^m).

D. Inversion:

Multiplicative inversion is highly complex and most studied finite field arithmetic operation. A novel multiplicative inversion is developed based on the proposed normal basis multiplier.
From Fermat's theorem, for every B € GF(2^m), B^{2^m} = B yielding

$$\begin{aligned} &B^{-1} \\ &=B^{2^m-2} \\ &=B^{2+2^2+2^3+...+2^{m-1}} \\ &=B^2B^{2^2}B^{2^3}...B^{2^{m-1}} \\ &=B^2(B^2)^2((B^2)^2)^2...\overbrace{((...((B)^2)^2)^2...)^2}^{m-1} \end{aligned}$$

..*(16)*
Fig. 5 shows the hardware implementation based on Eq. (16). The shift register T, which comprises m flip-flops, responds to the squaring computation of B^2, $(B^2)^2$, …, and $\overbrace{(...(B^2)^2...)^2}^{m-1}$.

Permutations P1 and P2 belong to permutation P and P^{-1}, respectively. The proposed algorithm for multiplicative inverse is described below.

Algorithm:

/*computing B^{-1} */

Step 1: Initialization

(1) Reset all 1-bit latches in cells U_i for $0 \leq i \leq m$ to 0s.

(2) Load operand B into shift register T.

Step 2: Deriving B^2

(1) Shift T to left by one bit.

(2) $D_0 = 0$, load D with T through permutation P1.

(3) Do not shift D.

(4) $S_0 = 1$

(5) Load final B^2 into shift register S.

(6) $S_0 = 0$

Step 3: Squaring and multiplication

(1) Shift T to left by one bit.

(2) $D_0 = 0$; load D with T through permutation P1.

(3) Shift D and S one bit for each clock cycle. After m+1 clock cycles, obtain D*S and store it in S.

Step 4: Repeat Step 3 m-3 times. Determine the final result of B^{-1} from the output of permutation P2.

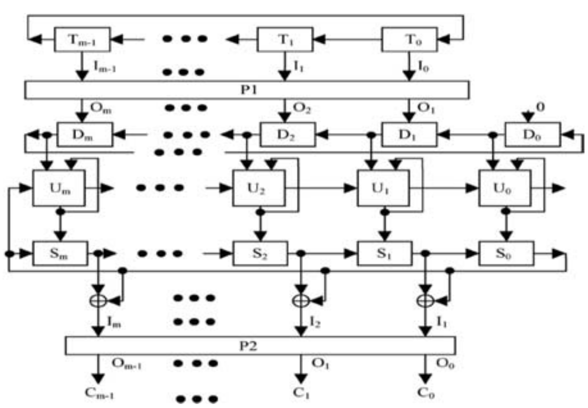


Fig. 5:The proposed normal basis multiplicative inverter in GF(2^m).

The proposed inverter is regular and modular, making it very attractive for VLSI implementation. The proposed inverter provides better time-area complexity for the larger value of m[5][8].

V. Results

Results of Polynomial basis arithmetic

a) Simulation done for two inputs (1100) 12 & (1000) 8.

Addition (00) = (00000100)

Multiplication (01) = (01100000)

Squaring (10) = (01010000)

Inversion (11) = (10110000)

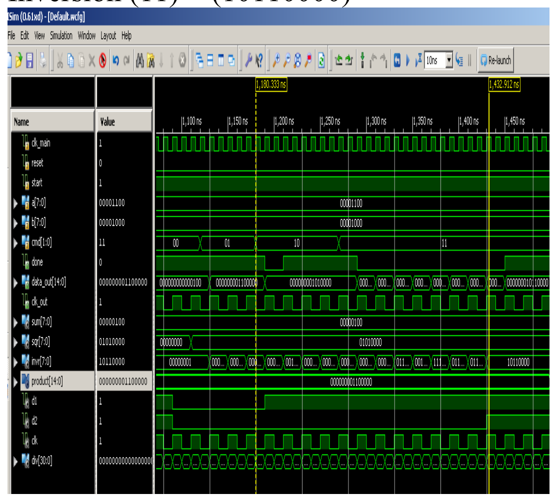


Fig.6: Simulation done for two inputs 12 & 8

Results of Normal basis arithmetic:

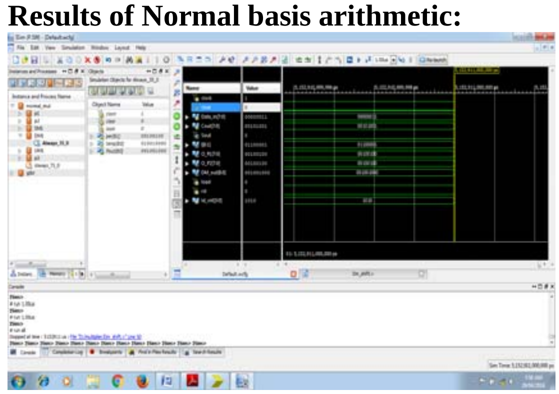


Fig.7: Waveform of normal basis multiplier.

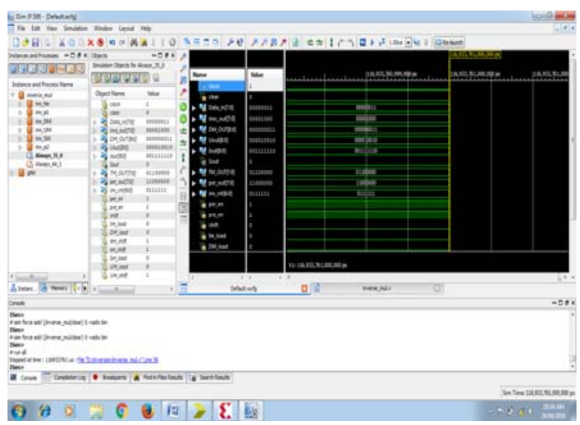


Fig. 8: Waveform of normal basis squaring and multiplicative inverse.

VI. Conclusion

We have implemented finite field arithmetic operations, i.e. addition, squaring, multiplication and inverse algorithms, on both polynomial basis and normal basis representations over GF(2^m). PB multipliers own the major features of simplicity, regularity, and modularity. The normal basis is especially known to be more efficient than polynomial basis because the inversion can be achieved by performing repeated multiplication and squaring can be executed by performing only one cyclic shift operation. Thus, are Very suitable for VLSI implementation.

Acknowledgement
I am highly obliged to the Management, Principal and HOD, Department of Electronics and Communication Engineering, SDM College of Engineering and Technology for providing me with the facilities being required .

References

[1] Darrel Hankerson, Scott Vanstone, Alfred J. Menezes"Guide to Elliptic Curve Cryptography"

[2] Sameh M. Shohdy, Ashraf B. El-Sisi, and Nabil Ismail "Hardware Implementation of Efficient Modified Karatsuba Multiplier Used in Elliptic Curves" International Journal of Network Security, Vol.11, No.3, PP.155–162, Nov. 2010

[3] Vijaylaxmi Hiremath, Renuka Korti "Implementation of finite field arithmetic unit for cryptographic applications", Proceedings of AECE-IRAJ International Conference, Tirupati, India, pp.70-74, 14th July 2013

[4] Reyhani-Masoleh, A. and Hasan, M. A., "A New Construction of Massey-Omura Parallel Multiplier over GF(2^m)," *IEEE Trans. Computers*, Vol. 51, pp. 511-520 (2002).

[5] Wang, C. C., Truong, T. K., Shao, H. M., Deutsch, L.J., Omura, J. K. and Reed, I. S., "VLSI Architectures for Computing Multiplications and Inverses in GF(2m)," *IEEE Trans.Computers*,Vol.C-34, pp. 709-717 (1985).

[6] Reyhani-Masoleh, A. and Hasan, M. A., "A New Construction of Massey-Omura Parallel Multiplier over GF(2m)," *IEEE Trans. Computers*, Vol. 51, pp. 511-520 (2002).

[7] Hasan, M. A.,Wang, M. and Bhargava, V. K., "Modular Construction of Low Complexity Parallel Multipliers for a Class of Finite Fields GF(2m)," *IEEE Trans.Computers*, Vol. 41, pp. 962 - 971 (1992).

[8] Jyoti N. D., Renuka Korti "Performance analysis of multiplication and inversion algorithm over GF(2^m) for coding and cryptographic applications, IJAREST, Volume 03 Issue 04, April-2016