A Novel methodology for Implementation RSA algorithm using FFT in Galois Field

S.Syamkumar[1] Ch.Umasankar [2]
[1]PG Student (M.Tech), Dept. of ECE, Universal Clg. of Eng. and Tech., Guntur, AP, India
[2]Assistant professor, Dept. of ECE, Universal Clg. of Eng. and Tech., Guntur, AP, India

Abstract— In the past decade, one of the most significant advances in cryptography has been the introduction of the first fully homomorphism encryption scheme (FHE). This advance resolved and opened the door to many new applications. Indeed, using a FHE one may perform an arbitrary number of computations directly on the encrypted data without revealing of the secret key. Thus an untrusted party, such as a remotely hosted server, may perform computations on behalf of the owner on the data without compromising privacy. This property of FHE is precisely what makes it invaluable for the cloud computing platforms today. A fully homomorphic encryption (FHE) scheme is envisioned as being a key cryptographic tool in building a secure and reliable cloud computing environment, as it allows arbitrarily evaluation of a cipher text without revealing the plaintext. However, existing FHE implementations remain impractical due to their very high time and resource costs. Of the proposed schemes that can perform FHE to date, a scheme known as FHE over the integers has the advantage of comparatively simpler theory, as well as the employment of a much shorter public key making its implementation somewhat more practical than other competing schemes. This paper presents the first hardware implementations of encryption primitives for FHE over the integers using FPGA technology. First of all, a super-size hardware multiplier architecture utilizing the Integer-FFT multiplication algorithm is proposed, and a super-size hardware Barrett modular reduction module is designed incorporating the proposed multiplier. Next, two encryption primitives that are used in two schemes of FHE over the integers are designed employing the proposed super-size multiplier and modular reduction modules.

Keywords— Modular multiplication, Rivest–Shamir–Adleman (RSA) cryptosystem, very large scale integration architecture, FHE, GF polynomial.

I. INTRODUCTION

The explosive growth in data communications and internet services has made the cryptography an important research topic to provide the need of confidentiality, authentication, data integrity, and/or non-reputation. The idea of public-key cryptosystem was originally presented by Diffie and Hellman. In 1978, Rivest, Shamir and Adleman introduced the famous RSA public-key cryptosystem, in which the characteristic is carried out by the modular exponentiation and the security lies on our inability to efficiently factor large integers (usually larger than 500 bits). To date, the RSA cryptosystem is still one of the most widely used public key cryptosystems. Moreover, since the size of modulus is at least 512 bits for long-term security, it means that high throughput rate is hard to achieve.

Encryption techniques are used essentially by the network security service to ensure the secret of information. A definition of security is needed to better understand it. According to Katz and Lindell, a security classic definition has two components: a security warranty that no information is leaked and a threat model which describes the adversary's abilities. But it is no need for perfect secrecy in real-world application. A tiny amount of information can be leaked to an adversary with bounded computational power, if it takes too long to decrypt data. This defines the computational security for nowadays cryptographic purposes. Modern cryptography requires a mathematical approach to define security. In this way, a scheme is secure if the success probability of any probabilistic polynomial-time (PPT) attack is negligible. Reliance on definitions and mathematical foundations represents a rigorous approach to cryptography.

The concept of computation on encrypted data without decryption was first introduced by Rivest, Adleman and Dertouzos in 1978. Thirty years later, Gentry proposed a fully homomorphic encryption (FHE) based on ideal lattices. This scheme is far from being practical because of its large computational cost and large ciphertexts. Since then, considerable efforts have been made to devise more efficient schemes.

However, most FHE schemes still have very large ciphertexts (millions of bits for a single ciphertext. This presents a considerable bottleneck in practical deployments.

We consider the following situation: several users upload data encrypted with a public-key FHE, a server carries out computations on the encrypted data and then sends them to an agency who has a decryption key for the FHE. This is common in typical FHE scenarios, such as medical and financial applications. In this situation, one approach to reduce the storage requirement is to use AES encryption to encrypt data, and then perform homomorphic computations on ciphertexts after converting to FHE-ciphertexts. This method has a great advantage in storage and communication, because only small AES-ciphertexts are transmitted from user to server, and these are homomorphically decrypted only when their homomorphic computations are required. In an asymmetric setting, we can still use this approach by adding several public-key FHE cipher texts of a session key. However this approach is not practical when the amount of messages transmitted simultaneously is small compared with the size of on FHE cipher text. Moreover, the conversion of AES-ciphertexts into FHE-ciphertexts requires a levelled FHE with multiplicative depth of at least forty.

In this paper, we explore an alternative method that encrypts messages with a public key encryption (PKE) and converts them into SHE-ciphertexts for homomorphic computations. In this approach, the ciphertext expansion ratio is only two or three regardless of the message size. Moreover, the decryption circuit is very shallow when the SHE allows large integers as messages.

In this paper we present the first hardware implementations of encryption primitives for FHE over the integers using FPGA technology. First of all, a super-size hardware multiplier architecture utilizing the Integer-FFT multiplication algorithm is proposed, and a super-size hardware Barrett modular reduction module is designed incorporating the proposed multiplier. Next, two encryption primitives that are used in two schemes of FHE over the integers are designed employing the proposed super-size multiplier and modular reduction modules.

II. FHE SCHEME BASICS

Fully homomorphic encryption can be considered as ring homomorphism. In mathematics, a ring is a set R equipped with two operations + and * satisfying the following eight axioms, called the ring axioms. R is an abelian group under addition, meaning:

1. (a+b)+c =a+(b+c) for all a,b,c in R(+ is associative)
2. There is an element 0 in R such that a +0 =a and 0 +a =a (0 is the additive identity)
3. For each a in R there exists _a in R such that a+(-a)=(-a)+a=0(-ais the additive inverse of a).
4. A+b=b+a for all a and b in R (C is commutative).

R is a monoid under multiplication, meaning:

5. (a * b) * c = a .(b . c) for all a; b; c in R (. is associative).

6. There is an element 1 in R such that a.1 = a and 1.a = a (1 is the multiplicative identity). Multiplication distributes over addition:

7. a * .(b + c)= (a * b) + .(a * c)for all a; b; c in R (left distributivity).

8. .(b + c) a =.(b * a)+ (c*a) for all a; b; c in R (right distributivity).

A ring homomorphism is a function between two rings which respects the structure. More explicitly, if R and S are two rings, then a ring homomorphism is a function

$$f : R \to S$$ such that

$$f(a+b) = f(a) + f(b)$$
$$f(a \cdot b) = f(a) \cdot f(b)$$

for all a and b in R. Let us see an example of ring homomorphism. Consider the function

$$f : Z_2 \to Z_2$$

given by

$$f(x) = x^2$$

where 2xy = 0 because 2 times anything is 0 in Z2. Next,

$$f(xy) = (xy)^2 = x^2y^2 = f(x)f(y)$$

The second equality follows from the fact that Z2 is commutative. Thus, f is a ring homomorphism. Let (P; C;K;E;D) be a encryption scheme, where P;C are the plaintext and ciphertext spaces, K is the key space, and E,D are the encryption and decryption algorithms. Assume that the plaintexts form a ring (P,*p,*p) and the ciphertexts forma ring (C,*c,*c); then the encryption algorithm E is a map

from the ring P to the ring C, i.e., Ek :p –c , where K is either a secret key (in the secret key cryptosystem) or a public key (in the public-key cryptosystem).

For all a and b in P and k in K, if

$$E_k(a) \oplus_c E_k(b) = E_k(a \oplus_p b)$$
$$E_k(a) \otimes_c E_k(b) = E_k(a \otimes_p b)$$

Overview of FHE

Craig Gentry, using lattice-based cryptography, showed the first fully homomorphic encryption scheme as announced by IBM on 25 June 2009. His scheme supports evaluations of arbitrary depth circuits. His construction starts from a somewhat homomorphic encryption scheme using ideal lattices that is limited to evaluating low-degree polynomials over encrypted data. It is limited because each ciphertext is noisy in some sense, and this noise grows as one adds and multiplies ciphertexts, until ultimately the noise makes the resulting ciphertext indecipherable. He then shows how to modify this scheme to make it bootstrappable—in particular, he shows that by modifying the somewhat homomorphic scheme slightly, it can actually evaluate its own decryption circuit, a self-referential property. Finally, he shows that any bootstrappable somewhat homomorphic encryption scheme can be converted into a fully homomorphic encryption through a recursive self-embedding. In the particular case of Gentry's ideal-lattice-based somewhat homomorphic scheme, this bootstrapping procedure effectively "refreshes" the ciphertext by reducing its associated noise so that it can be used thereafter in more additions and multiplications without resulting in an indecipherable ciphertext. Gentry based the security of his scheme on the assumed hardness of two problems: certain worst-case problems over ideal lattices and the sparse (or low-weight) subset sum problem. Regarding performance, ciphertexts in Gentry's scheme remain compact insofar as their lengths do not depend at all on the complexity of the function that is evaluated over the encrypted data. The computational time only depends linearly on the number of operations performed. However, the scheme is impractical for many applications, because ciphertext size and computation time increase sharply as one increases the security level. To obtain 2k security against known attacks, the computation time and ciphertext size are high-degree polynomials in k. Stehle and Steinfeld reduced the dependence on k substantially. They presented optimizations that permit the computation to be only quasi-k3:5 per Boolean gate of the function being evaluated. In 2009, Marten van Dijk, Craig Gentry, ShaiHalevi, and VinodVaikuntanathan presented a second fully homomorphic encryption scheme, which uses many of the tools of Gentry's construction, but which does not require ideal lattices. Instead, they show that the somewhat homomorphic component of Gentry's ideal latticebased scheme can be replaced with a very simple somewhat homomorphic scheme 50 3 Fully Homomorphic Encryption that uses integers. The scheme is therefore conceptually simpler than Gentry's ideal lattice scheme, but has similar properties with regard to homomorphic operations and efficiency. In 2010, Nigel P. Smart and FrederikVercauteren presented a fully homomorphic encryption scheme with smaller key and ciphertext sizes. The Smart– Vercauteren scheme follows the fully homomorphic construction based on ideal lattices given by Gentry. It also produces a fully homomorphic scheme from a somewhat homomorphic scheme. For somewhat homomorphic scheme, the public and the private keys consist of two large integers (one of which shared by both the public and the private keys), and the ciphertext consists of one large integer. The Smart–Vercauteren scheme has smaller ciphertext and reduced key size than Gentry's scheme based on ideal lattices. Moreover, the scheme also allows efficient fully homomorphic encryption over any field of characteristic two. However, the major problem with this scheme is that the key generation method is very slow. This scheme is still not fully practical. At the rump session of Eurocrypt 2011, Craig Gentry and ShaiHalevi presented a working implementation of fully homomorphic encryption (i.e., the entire bootstrapping procedure) together with performance numbers. Recently, Coron, Naccache, and Tibouchi proposed a technique allowing to reduce the public-key size of the van Dijk et al. scheme to 600 KB. In April 2013 the HElib was released, via GitHub, to the open source community which implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme, along with many optimizations to make homomorphic evaluation runs faster.

III. PROPOSED SYSTEM

The arithmetic operations in the Galois field have several applications in coding theory, Computer algebra and cryptography. Galois field is the set of all positive integers from 0, 1,... (P-1) where P is a prime

number. It is denoted by GF(P^m) where m is any positive value.

Many devices that perform functions such as error-control encoding, error detection, and error correction, operate by performing Galois field arithmetic over In practice, most implementations take p=2 and use binary digits (bits) to represent elements from the field. Performing Galois field arithmetic operations over GF(P^M) requires addition and multiplication. With, addition and multiplication modulo 2 become the exclusive-OR and logical-AND function, respectively. For this reason, and the ease with which a symbol of size 2^m may be handled in a binary system [e.g., a single byte may be represented as an element from GF(2^m). Galois fields of size 2^m are widely used.

The modular multiplication architecture is different from the interleaved version of Montgomery multiplication traditionally used in RSA design. By selecting different bases of 16 or 24 bits, it can perform 8,192-bit or 12,288-bit modular multiplication. A new RSA modular exponentiation algorithm using FFT multiplication is proposed to reduce one third of the calculation time of the large-number multiplication in modular multiplication. The design was implemented on the Altera's Stratix-V FPGA and 90-nm application-specified integrated circuit technologies.

Today, many embedded processors have AES or RSA cores included. This paper is aimed at taking a similar approach and designing a specific hardware or IP blocks for accelerating the core computations in FHE. Since the most computationally intensive operations in the FHE primitives are large-number modular multiplications, our initial attempt is to tackle the design of a large-number multiplier that can handle 768 000 bits, in support of the 2048-dimension FHE scheme demonstrated by Gentry and Halevi. In addition to FHE, large-number arithmetic also has other important applications in science, engineering, and mathematics. Specifically, when we need exact results or the results that exceed the range of floating point standards, we usually turn to multiprecision arithmetic [9]. An example application is in robust geometric algorithms. Replacing exact arithmetic with fixed-precision arithmetic introduces numerical errors that lead to nonrobust geometric computations. High precision arithmetic is a primary means of addressing the nonrobustness problem in such geometric algorithms. One of the holy grails of modern cryptography is FHE, which allows arbitrary computation on encrypted data. Given a need to perform a binary operation on the plaintext, FHE enables that to be accomplished via manipulation of the ciphertext without the knowledge of the encryption key.

$$E(x_1)+E(x_2) = E(x_1+x_2) \text{ and } E(x_1)\star E(x_2) = E(x_1 \otimes x_2)$$

For example, The first FHE was proposed by Gentry and was seen as a major breakthrough in cryptography. However, its preliminary implementation is too inefficient to be used in any practical applications. A number of optimizations were used in the Gentry–Halevi FHE variant, and the results of a reference implementation were presented . Due to limited space, here we only provide a high-level overview of the primitives.

Encryption:

To encrypt a bit b ∈ {0, 1} with a public key (d, r), encryption first generates a random "noise vector $u = \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with the probability p and as ±1 with probability (1 − p)/2 each. Gentry showed that u can contain a large number of zeros without impacting the security level, i.e., p could be very large. A message bit b is then encrypted by computing

$$c = [u(r)]_d = \left[b + 2\sum_{i=1}^{n-1} u_i r^i \right]_d \quad (1)$$

where d and r are parts of the public key. For the small setting with a lattice dimension of 2048, d and r have a size of about 785 000 bits.

When encrypted, arithmetic operations can be performed directly on the ciphertext with the corresponding modular operations. Suppose c1 = Encrypt(m1) and c2 = Encrypt(m2); then we have

$$\text{Encrypt}(b_1 + b_2) = (c_1 + c_2) \bmod d \quad (2)$$

$$\text{Encrypt}(b_1 * b_2) = (c_1 * c_2) \bmod d. \quad (3)$$

Decryption: The source bit b can be recovered by computing

$$b = [c \cdot w]_d \bmod 2 \quad (4)$$

where w is the private key. The size of the w is the same as that of d and r .

Decryption: Briefly, the recyption process is simply the homomorphic decryption of the ciphertext. The actual procedure of decryption is very complicated, so we choose not to explain it here. But from the brief description above, we can see that the fundamental operations for FHE are large-number addition and

multiplication. Addition has far less computing complexity than multiplication, so we focus on the hardware architecture of the multiplication using VLSI design.

Multiplication Algorithms Large-integer multiplication is by far the most time consuming operation in the FHE scheme. Therefore, we have selected it as the first block for hardware acceleration. A review of the literature shows that there is a hierarchy of multiplication algorithms. The simplest algorithm is the naive $O(N^2)$ algorithm (often called the grade school algorithm). The first improvement to the grade school algorithm was due to Karatsuba in 1962. It is a recursive divide-and-conquer algorithm, solving an N bit multiplication with three N/2 bit multiplications, giving rise to an asymptotic complexity of $O(N^{\log_2 3})$. Toom and Cook generalized Karatsuba's approach, using polynomials to break each N bit number into three or more pieces. Once the subproblems have been solved, the Toom–Cook method uses polynomial interpolation to construct the desired result of the N bit multiplication. The asymptotic complexity of the Toom–Cook algorithm depends on k (the number of pieces) and is $O(N^{\log(2k-1)/\log(k)})$.

The next set of algorithms in the hierarchy are based on using FFTs to compute convolutions. According to Knuth , Strassen came up with the idea of using FFTs for multiplcation in 1968, and worked with Schönhage to generalize the approach, resulting in the famous Schönhage–Strassen algorithm, with an asymptotic complexity of $O(N \cdot \log N \cdot \log\log N)$.

All the operations in FHE are modular operations. Usually, two different approaches are used to address the modular multiplication. The first is to do multiplication first, followed by modular reduction. The other approach, proposed, interleaves the multiplication with modular reduction. This is an efficient grade-school approach, performing the equivalent of two $O(N^2)$ multiplications. The interleaved Montgomery approach is quite commonly used for modular multiplication in the RSA algorithm, To understand the arithmetic cost of different multiplication algorithms, we implement three different modular multiplication algorithms in carefully tuned MIPS 64 assembly and count the number of ALU operations for each. The first algorithm uses the interleaved version of Montgomery multiplication proposed. This is an efficient grades chool approach, performing the equivalent of two O(N2) multiplications. The second algorithm uses the non interleaved three-multiplication Montgomery reduction implemented with Karatsuba multiplication (it uses the Karatsuba method if the arguments are larger than three words, and switches to grade-school multiplication to handle the base case when the arguments are small). The third algorithm adopted in this paper is based on FFT multiplication and is described in detail in the next section. This algorithm also uses a traditional three multiplication Montgomery reduction. Comparing the Karatsuba and FFT multipliers, both of which compute the product and then reduce the result modulo N, we can see that FFT multiplication is faster, requiring only one-third of the number of instructions as the Karatsuba multiplier. Comparing the FFT multiplier with interleaved Montgomery approach which is widely used in RSA for modular multiplication, we see that the FFT multiplier uses only 1/20th of the number of instructions. The interleaved version of Montgomery multiplication is popular and efficient in RSA, but it is no longer efficient for the modular multiplication in FHE. In all, the approach we adopt for modular multiplication is the most efficient algorithm. From above, we can see that large-number multiplication is the most crucial part for the modular multiplication. Therefore, we take the first step to design a fast multiplier for hardware implementation. For further reading, there are a number of papers that cover hardware implementation of large-number multiplication Yazaki and Abe implement a 1024-bit Karatsuba multiplier, and they investigate a hardware implementation of FFT multiplication. Kalach investigates a hardware implementation of finite field FFT multiplication. However, that paper does not present any information about the hardware resources and performance.

FFT Multiplication

FFT multiplication is based on convolutions. For example, to compute the product A times B, we express the numbers A and B as sequences of digits (in some base b) and then compute the convolution of the two sequences using FFTs.

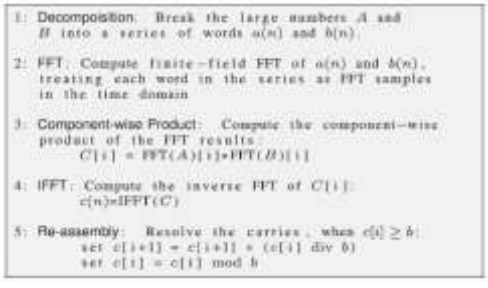


Fig. 1. FFT multiplication.

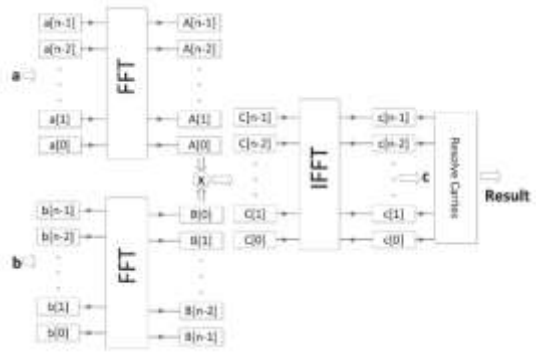


Fig. 2. FFT-based multiplication algorithm.

Once Once we have the convolution of the digits, the product A times B can be found by resolving the carries between digits. The FFT multiplication algorithm is presented in Fig. 1 and as a diagram in Fig. 2. The FFT computations can done either in the domain of complex numbers or in a finite field or ring. In the complex number domain, it is trivial to construct the roots of unity required for the FFT, but the computations must be done with floating point arithmetic and the round-off error analysis is quite involved. In the finite field/ring case, all the computations are done with integer arithmetic and are exact. However, the existence and the calculation of the required root of unity will depend heavily on the structure of the chosen finite field/ring. For our FFT multiplier, we follow the steps of our previous work and implement the FFT in the finite field Z/pZ, where p is the prime $2^{64}-2^{32}+1$. This prime is from a special class of numbers called Solinas primes (see [22]). As we shall see, this choice of p has three compelling advantages for FFTs.

1) We can do very large FFTs in Z/pZ. Since 232 divides p − 1, we can do any power-of-2-sized FFT up to 232.

2) There exists a very fast procedure for computing x modulo p for any x.

3) For small FFTs (up to size 64), the roots of unity are all powers of 2. This means that small FFTs can be done entirely with shifting and addition, rather than requiring expensive 64-bit multiplications.

FFTs in the Finite Field Z/pZ

To perform FFTs in a finite field, we need three operators: addition, subtraction, and multiplication, all modulo p, where $2^{64}-2^{32}+1$. Addition and subtraction are straightforward (if the result is larger than p then subtract p, and if the result is negative, then add p). For multiplication, if X and Y are in Z/pZ, then X ∗ Y will be a 128-bit number, which we can represent as $X * Y = 2^{96}a + 2^{64}b + 2^{32}c + d$ (where a, b, c, and d are each 32-bit values). Next, using two identities of p, namely, $2^{96} \bmod p = -1$ and $2^{64} \bmod p = 2^{32} - 1$, we can rewrite the product of X ∗ Y as

$$\begin{aligned} X * Y &\equiv 2^{96}a + 2^{64}b + 2^{32}c + d \pmod{p} \\ &\equiv -1(a) + (2^{32} - 1)b + (2^{32})c + d \\ &\equiv (2^{32})(b + c) - a - b + d. \end{aligned}$$

This means that a 128-bit number can be reduced modulo p to just a few 32-bit additions and subtractions. Further, note that $2^{192} \bmod p = 1$, $2^{96} \bmod p = -1$, $2^{384} \bmod p = 1$, etc. This leads to a fast method to reduce any sized value modulo p. Break the value up into 96-bit chunks and compute the alternating sum of the chunks. Then reduce the result as above. This means that a 128-bit number can be reduced modulo p to just a few 32-bit additions and subtractions. Further, note that 2192 mod p = 1, 2^{96} mod p = −1, 2^{384} mod p = 1, etc. This leads to a fast method to reduce any sized value modulo p. Break the value up into 96-bit chunks and compute the alternating sum of the chunks. Then reduce the result as above. In addition to the arithmetic operator, there are three other criteria in order to perform multiplication with finite field FFTs. First, to compute an FFT of size k, a primitive root of unity r_k must exist. In a finite field, the process for doing an FFT is analogous to FFTs in the complex domain; thus

$$X_i = \sum_{j=0}^{k-1} x_j (r_k)^{ij} \pmod{p} \quad (5)$$

The inverse FFT (IFFT) is just

$$x_i = k^{-1} \sum_{j=0}^{k-1} X_j (r_k)^{-ij} \pmod{p} \quad (6)$$

for all the usual methods for decomposing FFTs, such as Cooley–Tukey, except $(r_k)^j$ takes the place of $e^{j2\pi i/k}$. With large FFTs, the primitive roots almost always look like random 64-bit numbers; e.g., the $r_{65\,536}$ that we use is 0xE9653C8DEFA860A9.However, for FFTs of size 64 or less, the roots of unity will always be powers of 2. As noted above, 2^{192} mod p = 1, which means $(2^3)^{64}$ mod p = 1 and therefore $r_{64} = 2^3 = 0 \times 08$. Likewise, $r_{16} = 2^{12}$. For our hardware implementation, we will choose k = 65 536 and b = 2^{24}. These values meet the criteria above and allow us to multiply two numbers up to $b^{k/2} = 2^{786\,432}$, i.e., 786 432 bit in length, which is sufficient to support Gentry–Halevi's FHE scheme for the small setting with a lattice dimension of 2048.

192-bit Wide Pipelines It is often the case in our hardware FFT implementation that we need to perform a sequence of modular operations (additions, subtractions, and multiplications by powers of 2). If we were to implement this as 64-bitwide operations, we would need to reduce the result modulo p between each stage of the pipe. Although the process to reduce a value modulo p is quite fast, it still requires a lot of hardware. It turns out that, if we extend each 64-bit value to 192 bits (by padding with zeros on the left) and run the pipeline with 192-bitwide values, then we can avoid the modulo p operations after each pipeline stage by taking advantage of the fact that 2192 mod p is 1. We do this as follows:

1) Addition: Suppose we wish to compute x + y. There are two cases: If we get a carry out from the 192nd bit, then we have trunc(x+y)+2^{192}, which is the same as trunc(x+y)+1 modulo p (where trunc(z) returns the least significant 192 bits of z). If it did not carry out, then the result is just x + y. We can implement this efficiently in hardware using circular shifting operations.

2) Multiplication by a Power of 2: First, let us consider multiplication by 2. Suppose we have a 192-bit value x and we wish to compute 2x. There two cases. If the most significant bit of x is zero, then we simply shift all 1-bits to the left. If the top bit is set, then we need to compute trunc(2x) + 2192, which is the same as trunc(2x) + 1 modulo p. In both case, it is just a left circular shift by 1 bit. Thus to compute 2 j ∗ x, we simply do a left circular shift by j bits.

3) Subtraction: Since 2^{96} mod p = −1, we can simply rewrite x − y as x + 2^{96}y. The 2^{96} is a constant shift. For the final reduction from 192 bits back down to 64 bits, as above, we can represent a 192-bit number z as

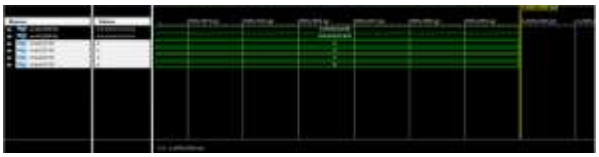
$$z = 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f$$

where a, b, c, d, e, and f are each 32 bits

$$\begin{aligned} z &\equiv 2^{160}a + 2^{128}b + 2^{96}c + 2^{64}d + 2^{32}e + f \\ &\equiv -(2^{32}-1)a - 2^{32}b - c + (2^{32}-1)d + 2^{32}e + f \\ &\equiv (2^{32}e+f)+(2^{32}d+a)-(2^{32}b+c)-(2^{32}a+d). \end{aligned} \quad (7)$$

IV. RESULTS & DISCUSSIONS

Experimental results of the proposed system were shown .



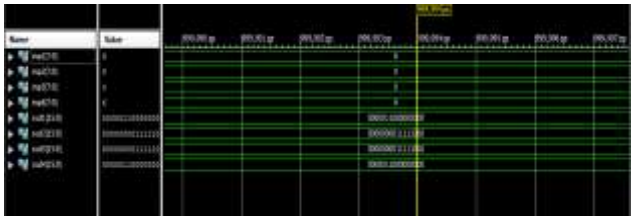


Figure 3 Encryption result

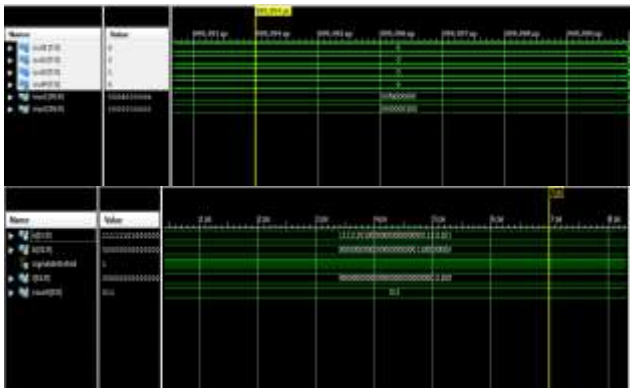


Figure 4 Decryption result

These works assume perfect channel state information (CSI) is available at the receiver. Receiver algorithms for the realistic case when CSI is not available Efficient receiver structure was proposed for RADIX FFT scheme over frequency-flat fading channels, based on generalized maximum-likelihood sequence estimation space-time block codes and space-time trellis codes are two very different transmit diversity schemes. Space-time block codes are constructed from known orthogonal designs, achieves full diversity, are easily decodable by maxim um likelihood decoding via linear processing at the receiver, but suffers from a lack of coding gain. On the other hand, space-time trellis codes possess both diversity and co ding gain, yet is complex to decode (since maximum likelihood sequence estimation is require), and arduous to design. For RADIX FFTs, coding gain is only achieved if it is concatenated with an outer code, such as a TCM code or a TurboTCM Code . This was mentioned as an ongoing, exciting area of research

References

1. G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, “Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation,” IEEE Trans. Ind. Electron., vol. 58, no. 7, pp. 3101–3109, Jul. 2011.
2. S. Yazaki and K. Abe, “VLSI design of Karatsuba integer multipliers and its evaluation,” IEEE Trans. Electron., Inf. Syst., vol. 128, no. 2, pp. 220–230, Feb. 2008
3. A. Schönhage and V. Strassen, “Schnelle Multiplikation Großer Zahlen,” Computing, vol. 7, no. 3, pp. 281–292, 1971.

4. P. Montgomery, "Modular multiplication without trial division," Math.Comput., vol. 44, no. 170, pp. 519–521, 1985.
5. S. Yazaki and K. Abe, "An optimum design of FFT multi-digit multiplier and its VLSI implementation," Bull. Univ. Electro-Commun., vol. 18,no. 1, pp. 39–45, 2006.
6. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comput., vol. 19, no. 90, pp. 297–301,1965.
7. L. Jia, Y. Gao, and H. Tenhunen, "A pipelined shared-memory architecture for FFT processors," in Proc. 42nd IEEE Midwest Symp. Circuits Syst., vol. 2. Aug. 1999, pp. 804–807.
8. K. Kalach and J. P. David, "Hardware implementation of large number multiplication by FFT with modular arithmetic," in Proc. 3rd Int. IEEENEWCAS Conf., Jun. 2005, pp. 267–270.
9. J. Solinas, "Generalized mersenne numbers," Blekinge College Technol., Karlskrona, Sweden, Tech. Rep. 06/MI/006, 1999

Authors Profile:

S.Syamkumar is pursuing his M. Tech in Department of Electronics and Communication Engineering at Universal College of Engineering & Technology, Guntur. His specialization is VLSID

Ch. Umasankar is an Assistant professor in the Department of Electronics and Communication Engineering at Universal College of Engineering & Technology, Guntur. He has published several papers on his interested area of VLSI signal processing.

****