On Software Implementation of High Performance GHASH Algorithms

by

Iqbal Muhammad Umair

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There have been several modes of operations available for symmetric key block ciphers, among which Galois Counter Mode (GCM) of operation is a standard. GCM mode of operation provides confidentiality with the help of symmetric key block cipher operating in counter mode. The authentication component of GCM comprises of Galois hash (GHASH) computation which is a keyed hash function. The most important component of GHASH computation is carry-less multiplication of 128-bit operands which is followed by a modulo reduction. There have been a number of schemes proposed for efficient software implementation of carry-less multiplication to improve performance of GHASH by increasing the speed of multiplications. This thesis focuses on providing an efficient way of software implementation of high performance GHASH function as being proposed by Meloni et al., and also on the implementation of GHASH using a carry-less multiplication instruction provided by Intel on their Westmere architecture.

The thesis work includes implementation of the high performance GHASH and its comparison to the older or standard implementation of GHASH function. It also includes comparison of the two implementations using Intel's carry-less multiplication instruction. This is the first time that this kind of comparison is being done on software implementations of these algorithms. Our software implementations suggest that the new GHASH algorithm, which was originally proposed for the hardware implementations due to the required parallelization, can't take advantage of the Intel carry-less multiplication instruction PCLMULQDQ. On the other hand, when implementations are done without using the PCLMULQDQ instruction the new algorithm performs better, even if its inherent parallelization is not utilized. This suggest that the new algorithm will perform better on embedded systems that do not support PCLMULQDQ.

Acknowledgements

I am grateful to all who have guided me and helped me to reach this milestone, but there are few who I would like to specially mention here. First of all, I would like to thank my graduate supervisor, Professor Anwar Hasan, who has been a great support and guide thorough out my program. I would also like to thank Professor Catherine Gebotys and Professor Mark Aagaard for reviewing my work. I would also like to take this opportunity to thank my family who has always been there for me, and encouraged me at every step. In the end I would also like to thank all the graduate students in my lab, who from time to time have provided me with useful advice and guidance.

Contents

List of Tables

List of Figures

List of Abbreviations

AD	additional Authentication Data
AES	Advanced Encryption Standard
C	Ciphertext
FPGA	Field Programmable Gate Array
GCM	Galois Counter Mode
GF	Galois Field
GHASH	Galois Hash
GMAC	Galois Message Authentication Code
IV	Initialization Vector
K	random Key
LSB	Least Significant Bit
MAC	Message Authentication Code
MD4/5	Message Digest
MNH	Meloni, Negre and Hasan
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology
OML	Ordinary Multiplication
P	Plaintext
PRU	Polynomial Reduction Unit
SHA	Secure Hash Algorithm
T	authentication Tag
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction

The three main goals of information systems security, namely confidentiality, integrity, and availability have always been a point of interest to the cryptographic research world. Apart from being highly secure an important required feature for some practical information systems is to perform cryptographic operations at high speed. Block ciphers have proven themselves to be useful for this purpose. Among many block ciphers, Advanced Encryption Standard (AES) is one of the widely used symmetric key block ciphers [3]. National Institute of Standards and Technology (NIST) standardized the operation of symmetric key block ciphers in the Galois Counter Mode (GCM) due to its suitability for efficient implementation in hardware as well as software. A lot of research work has been done on proposing efficient ways of implementation, and its usage in different types of networks and applications. GCM provides data authentication/integrity by using the Galois hash (GHASH), and supports data confidentiality via encryption/decryption operations of AES. Below we give a brief overview of previous research work related to the implementations of GHASH.

1.1 Brief Overview of Previous Work on GHASH

Over the past years, there have been various schemes proposed for improving data authentication component of AES-GCM based systems, i.e., GHASH. One of the research papers has proposed a GCM variation in [9], where the authors have addressed the slowness of computation of GHASH and the problem of memory requirements for the pre-computed GHASH. There is also an efficient GHASH implementation on FPGA proposed in [25] which uses a parallel architecture for the polynomial multiplication in Galois fields [26]. Another efficient implementation has been presented in [5], again using the multiplier from [26], but this time combined with the pipeline method for higher throughput. Hardware implementation on a per key basis has been proposed in [7], and the GHASH has been implemented using Verilog, resulting in improved throughput. In [18], an efficient implementation has also been proposed for GHASH using Intel's PCLMULQDQ instruction [17]. The work in [18] attempts to optimize the assembler implementation of GHASH algorithm, and performs better than the standard implementation. Finally Intel itself has proposed an optimized implementation of GHASH in GCM, using their own PCLMULQDQ instruction [13].

Although there have been several implementations proposed for GHASH, they all have one thing in common: they are all trying to implement or improve the standard GHASH algorithm. Performance of the algorithm becomes slow as the number of blocks being processed increases, since the number of 128-bit Galois field multiplications in the standard GHASH algorithm is almost as many as the number of blocks. Although there have been schemes proposed for utilizing parallel hardware to overcome some problems, but in terms of software implementation it is hard to mimic that level of parallelism.

Recently, a new GHASH algorithm has been proposed by Meloni, Negre and Hasan [29]. We will refer to this new algorithm as MNH GHASH. This algorithm replaces all extension field multiplications in excess of 127 by an equal number of polynomial reduction operations. This algorithm has been primarily designed for dedicated hardware implementations to take advantage of its inherent parallelism. To the best of our knowledge, no work has been reported yet investigating the performance of the algorithm when implemented using software on a general purpose processor.

1.2 Contributions

The work presented in this thesis is with regard to faster computation and timing analysis of different implementations of GHASH. The main contributions are as follows:

- Using common place instructions, perform software design and implementation of the MNH GHASH algorithm recently proposed by Meloni et al.[29].
- Implement the above mentioned GHASH algorithm [29] using Intel's new 64-bit carry-less multiplication PCLMULQDQ instruction[17].
- Compare performance of the software implementations of the new and the standard GHASH with and without Intel's carry-less multiplication instruction.

There have been other implementations of GHASH presented in the past, e.g., [13, 5, 18]. These are mainly to improve the implementation of the standard GHASH algorithm and most of them are either FPGA or ASIC implementations. On the other hand, in this work we deal with the new MNH GHASH algorithm

[29], and present software implementations and comparison. Unlike previous research papers on this topic, our work is not two implementations of the standard GHASH algorithm, but rather comparison of two different algorithms.

1.3 Organization

There are four more chapters in this thesis, starting with Chapter 2 in which we give related background on Galois field (GF) arithmetic, GCM and cryptographic hash functions. In Chapter 3, we discuss algorithms for computing GHASH in GCM, and explain how using minimal polynomials we can improve the GHASH algorithm as presented by Meloni et al. [29]. In Chapter 4, the software implementation, analysis and results obtained from them are discussed. In Chapter 5 some concluding remarks are made on analysis and results. This chapter also includes some discussions on possible future work.

Chapter 2

Background

In order to better understand the GHASH, which is used in GCM, we first need to understand how GCM mode works for a block cipher. Furthermore, to understand GHASH we need to deal with Galois field operations, and to understand GCM we need to know a bit about cryptographic hash functions and how are they computed. This chapter starts with a brief introduction on Galois fields, some of its basic operations and a bit about characteristic polynomials. Then we discuss cryptographic hash functions. The chapter ends with some discussions on GCM operation.

2.1 Galois Fields

Galois fields are most widely used in coding theory and field of information security. A Galois field has a finite number of elements, with which one can perform addition, subtraction, multiplicaion and division (by the non-zero element). The Galois field of q elements is denoted as GF(q). The value of q must be a prime or a prime power. If q is prime (respectively, prime power), field GF(q) is referred to as prime (respectively, extension) field. For example, a prime Galois field is GF(2), which can be extended to field GF(2^m), where m can be any integer greater than 1 [31].

2.1.1 Polynomial Representation of Galois Field

Although several representations for finite fields have been proposed, the one using polynomial basis has been the most useful, specially when it comes to large fields. In order to give a more general representation of a Galois field in polynomial basis, assume a Galois field GF(p^n), where p is prime. Let us assume that $F(x)$ is an irreducible polynomial, whose coefficients belong to GF(p), and is of degree n. An irreducible polynomial does not have any polynomial as its factor which has a degree greater than 0 or smaller than n. Since $F(x)$ is a polynomial of degree n, it is often convenient to write it as follows [15]:

$$F(x) = x^n + f(x) \tag{2.1}$$

where,

$$f(x) = \sum_{j=0}^{n-1} f_j x^j, \{f_j \in \mathrm{GF}(p)\}$$

Now, if we assume that a root of *F(x)* is β, then any element B in field GF(p^n) can be represented as follows,

$$B(\beta) = b_{n-1}\beta^{n-1} + b_{n-2}\beta^{n-2} + b_{n-3}\beta^{n-3} + ... + b_1\beta^1 + b_0 = \sum_{j=0}^{n-1} b_j\beta^j \tag{2.2}$$

where, $b_j \in$GF(p), and the polynomial basis of GF(p^n) over GF(p) is formed using $\{1, \beta, \beta^2, \beta^3, ...\beta^{n-2}, \beta^{n-1}\}$.

2.1.2 Galois Field Arithmetic

In order to further proceed with our discussion, it is important to give a brief introduction to some basic Galois field operations. In the next few paragraphs, we will look into Galois field addition, multiplication and the concept of minimal polynomial (importance of which we will see in the next chapter).

Addition Operation in Galois Field

Addition in Galois field is a very simple operation. For example, if we have Galois field elements $C(x)$ and $D(x)$, in polynomial basis form for field GF(p^n), then their addition would be modulo p addition of the corresponding coefficients of $C(x)$ and $D(x)$. A better example could be in case of binary field GF(2^n). Let $C(x)$ be $x^2 + x + 1$, and $D(x)$ be $x + 1$. Then their sum $S(x)$ is

$$S(x) \equiv C(x) + D(x) \equiv ((x^2 + x + 1) + (x + 1)) \bmod 2$$

$$S(x) \equiv (x^2 + (x + x) + (1 + 1)) \bmod 2 \equiv x^2$$

Another way of looking at addition in binary field from the implemenatation perspective is to observe that if the elements are stored in bit form, then addition is nothing but XORing of corresponding bits.

Multiplication Operation in Galois Field

Multiplication in Galois field is a little more complicated operation than addition. For multiplication of two elements of GF(p^n), first the polynomials corresponding to the filed elements are multiplied and then they go through a modular reduction using polynomial $F(x)$, which as mentioned earlier is an irreducible polnomial of degree n. To illustrate a small example, let us assume that we have two elements $C(x)$ and $D(x)$ of Galois field GF(2^3). Let $C(x)$ be x^2 and $D(x)$ be x, and the field defining irreducible polynomial $F(x)$ be $x^3 + x + 1$. Then we can multiply $C(x)$ and $D(x)$ as follows,

$$M(x) \equiv C(x) \cdot D(x) \bmod F(x),$$

$$M(x) \equiv (x^2) \cdot (x) \bmod (x^3 + x + 1),$$

$$M(x) \equiv x^3 \bmod (x^3 + x + 1) \equiv x + 1.$$

Minimal Polynomial in Galois Field

An important concept related to Galois fields, which is worth mentioning here, is minimal polynomial. The minimum polynomial of any element α of field GF(p^n) is a polynomial $M(x)$, such that $M(\alpha) = 0$, and its coefficients are in field GF(p) [31]. For example if we have element 0 in Galois field GF(2^m), then its minimal polynomial will be x, and similarly for element 1 the minimal polynomial is $x + 1$. Now let us look at a more elaborate example. Let us consider field GF(2^4) with field defining polynomial to be $F(x) = x^4 + x + 1$. Let α be a root of $F(x)$. Then for field element $\alpha^2 + \alpha$, we have minimal polynomial $x^2 + x + 1$, which can be verified as follows,

$$M(x) \equiv x^2 + x + 1,$$

$$M(\alpha^2 + \alpha) \equiv (\alpha^2 + \alpha)^2 + \alpha^2 + \alpha + 1,$$

$$M(\alpha^2 + \alpha) \equiv \alpha^4 + \alpha^2 + \alpha^2 + \alpha + 1,$$

$$M(\alpha^2 + \alpha) \equiv \alpha^4 + \alpha + 1,$$

Since α is root of $F(x)$, hence $F(\alpha) = \alpha^4 + \alpha + 1 = 0$, resulting in $M(\alpha^2 + \alpha) = 0$, i.e., *M(x)* is the minimal polynomial of $\alpha^2 + \alpha$.

2.2 Hash Functions in Cryptography

A cryptographic hash function can be considered to be an algorithm which takes blocks of data and convert them to strings often referred to as tags. A tag can be viewed as a finger print of the message or representation of message and is unique. In a normal hash function, there is no concept of key; but we will briefly look into keyed hash functions or message authentication code (MAC) as well, since the hash function of our interest GHASH, which is used in GCM, uses keys. In short, a hash functions provides an easy and efficient way of representing a message of arbitrary length and produces a tag of finite bits string, which helps in signing messages and resolves the issue of high computation and message overhead costs involved when computing digital signatures without hash functions [34].

Hash functions are generally expected to be easily computed, and even if one bit changes in the message the whole hash function generated again should not be the same. That is they need to be highly sensitive to any change, and satisfy other properties. Below we discuss a couple of important security properties of hash functions.

2.2.1 Security Properties of Hash Functions

A cryptographic hash function should be one-way and collision resistance. One-wayness, which is also known as preimage resistance, guarantees that it is ideally impossible to create the message back from a hash function or hash tag.

Collision resistance is one of the most important properties or requirements for hash functions. It implies that there are no two input messages which can produce the same hash tag. A hash function's collision resistance can be either weak or strong. In the weak case, one message is already given and the attacker tries to find a second message which can produce the same hash tag. In the strong case, the attacker has an opportunity to select any two messages and see if it is possible to get the same hash tag from both.

2.2.2 Types of Hash Algorithms

Hash algorithms can be divided into two major categories: dedicated hash functions and block ciphers based hash functions [34]. It is also worth mentioning that hash functions can be keyed or not keyed. Our hash function of interest, i.e., GHASH is a keyed hash function. A keyed hash function uses both the message and the key for computing a hash tag and is generally used in Message Authentication Code (MAC). Un-keyed hash functions on the other hand are used mostly in error detection codes, and their computations does not require any key.

Dedicated Hash Functions

As the name suggests, dedicated hash functions are specifically designed for computing hashes and do not usually rely upon complex computations like discrete logarithm or integer factorization. Examples of dedicated hash functions are MD4, MD5 and various SHAs.

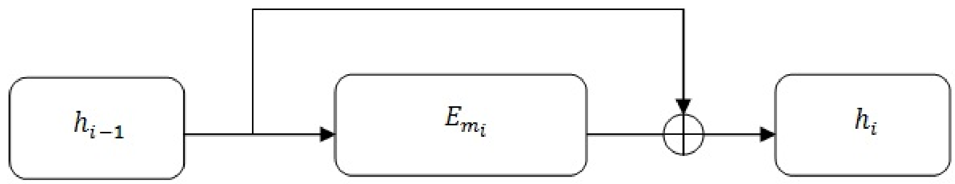

Figure 2.1: Simplified Davies-Meyer Hash function

Block Cipher based Hash Functions

In terms of computation speeds, the block cipher based hash functions are bit slower as compared to the dedicated ones, but they give an added advantage of using the same block cipher which is being used for encryption. There are again many different methods proposed in the past to generate hash tags using block ciphers. In Fig. 2.1, a very basic method known as Davies-Meyer method [6] is shown.

In the Davies-Meyer method, the hash is constructed by taking previous hash value h_{i-1} as input to a block cipher encryption function E. Using the ith message block m_i as the key and then whatever output is generated is XORed with the previous hash value to obtain new ith hash value h_i. In the case of first hash tag, usually a pre-computed specific initial hash value h_0 is used.

2.3 Galois Counter Mode (GCM)

Galois Counter Mode (GCM) is a recommended mode of operation for symmetric key block ciphers by NIST [17, 33]. Galois counter mode of operation handles confidentiality through encryption in the counter mode and authentication is taken care by computation involving a secure hash function. The Galois field used for easiness of hardware/software implementation is binary field GF(2^{128}). GCM provides encryption using the symmetric block cipher AES (Advanced Encryption Standard) [33].

The term GMAC is often heard in the context to GCM, which only means that if our input data does not contain any information which is needed to be encrypted then the operation of GCM could be just called GMAC. In that case it is only providing data authentication, and it is needless to say that authentication provided by GCM is far stronger than any error detecting code or check sum [33]. GCM also provides lot of opportunities for pre-computations and parallelized implementation [33]. For example, even the length of input data is not required in advanced; but if we know it, it is fixed, and if we also know about the initialization vector, then lot of block cipher computations related to invocation can be done beforehand [33].

2.3.1 GCM Operation

As mentioned earlier, GCM is composed of two parts: authentication and encryption. Data authentication is achieved via the keyed hash function GHASH and encryption via block cipher AES in the counter mode. As cryptographic hash functions have already been discussed earlier, a brief description on block ciphers and counter mode of operation will be given below.

Block Ciphers

A block cipher causes the input data to go through a particular transformation, and in every transformation, a fixed amount of data from input is taken, which is called a block. The operations of block ciphers are dependent on a random key,

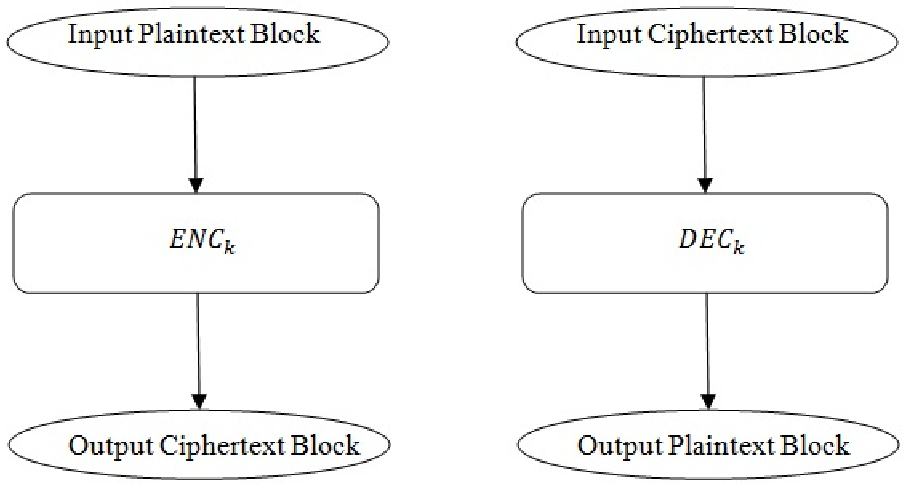

Figure 2.2: Simple Block Cipher in ECB Mode

say K, regulates the transformation which input block goes through. To achieve confidentiality, the block ciphers uses two functions that are inverse of each other, and one is called encryption and the other decryption. To present a visual and simplified example of a block cipher, Fig. 2.2 shows a simple block cipher in electronic code book mode.

So, as we can see in the above figure, it has two functions: one for encryption ENC_K, and the other funtion DEC_K, where $DEC_K = ENC_K^{-1}$.

Counter Mode of Operation for Block Ciphers

There are different modes of operations for block ciphers. The mode used in Fig. 2.2 is known as electronic code book mode. The mode that is most important from the GCM perspective is Counter Mode of operation. One important feature of the counter mode of operation is that it doesn't require two functions like the example of block cipher we saw earlier. It only needs forward cipher function,

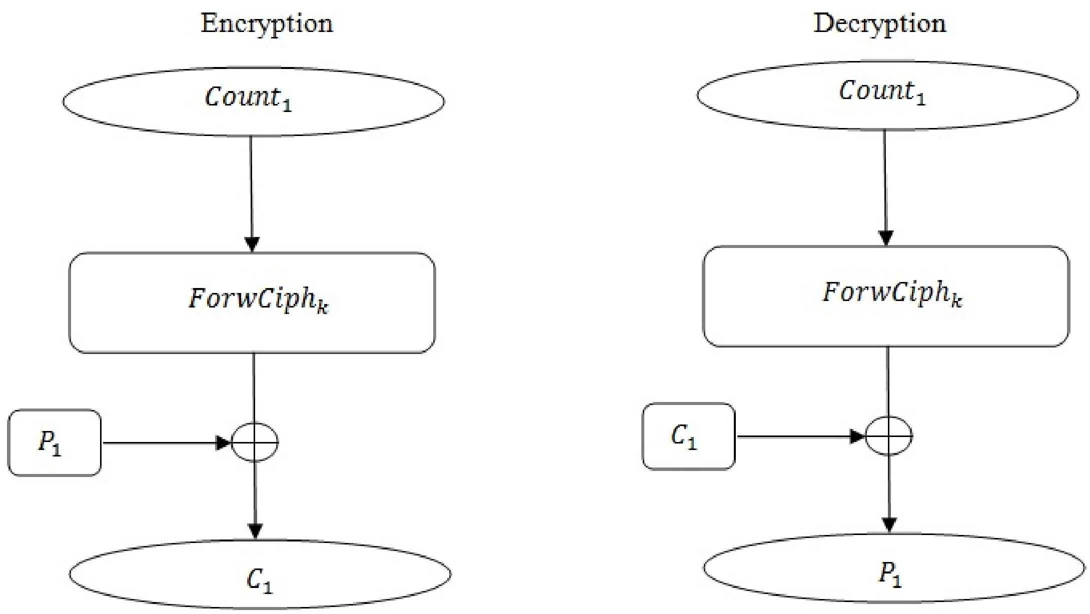

Figure 2.3: Encryption and Decryption in counter mode of operation

which is advantageous from the implementation point of view. In this mode of operation, forward cipher function transformations are applied on counter blocks, which are special input blocks, and have the property of being distinct per block under same key. The output from those transformations is XORed to produce the ciphertext. To get the plaintext back, the same counter goes through the forward cipher function and, is then XORed with the ciphertext [32]. To give a simple example, let us consider a counter block X_1 , and we apply forward cipher function with key K on as $Forw_K(X_1)$. Now we XOR it with plaintext P_1, then cipher text C_1will be $P_1 \oplus Forw_K(X_1)$. To get back P_1, we just need to get forward cipher function applied on the same counter, that is , $Forw_K(X_1)$, and then XORing it to C_1, which will give us back our plaintext P_1. A simplified block diagram of Counter Mode of operation of block ciphers can be seen in Fig. 2.3.

As we can see from Fig. 2.3, $Count_1$ is a distinct counter block for a block, P_1 of plaintext, whereas the $ForwCiph_K$, is the forward cipher function, with key K, and cipher text is C_1, and similarly in decryption we just reverse the XORing operation.

GCM Specification

As we already know, GCM requires a block cipher for establishing confidentiality feature. Let us assume that the cipher block function is using random key K, and the input required is composed of the Plaintext P, which is the actual data to be encrypted. Plaintext P is usually broken into blocks of 128 bits long except for the last block. If the last block is not already 128 bits, then extra zeros are padded. Another part of input comprises of additional Authentication Data *(AD),* which is not encrypted and used only for the purpose of authentication. The length restriction for blocks of this data is the same as for the plaintext [33, 28]. The final part of input is Initialization Vector *(IV),* which is a nonce, and is unique in reference to the context, and has a main role in invocation of forward cipher function. Its construction and properties are discussed in more details in the NIST specification [33]. The resulting output of GCM operation is Ciphertext *(C)* and authentication Tag *(T)*. The decryption part takes, initialization vector, ciphertext, authentication data, and tag as input, and using initialization vector and cipher text it produces the plaintext. A simplified version of encryption and decryption blocks can be seen in Fig. 2.4. It does not include how authentication part works in GCM, which is part of our following discussion.

Now as we can see in the figure, input data for encryption goes through a block called $GCTR_K$, which is nothing much but a modified counter block operation as discussed earlier and uses the block cipher for encryption with key, K. The ciphertext output is also broken into blocks and with the same length restrictions and padding as plaintext, and the authentication tag produced is 128 bits in length.

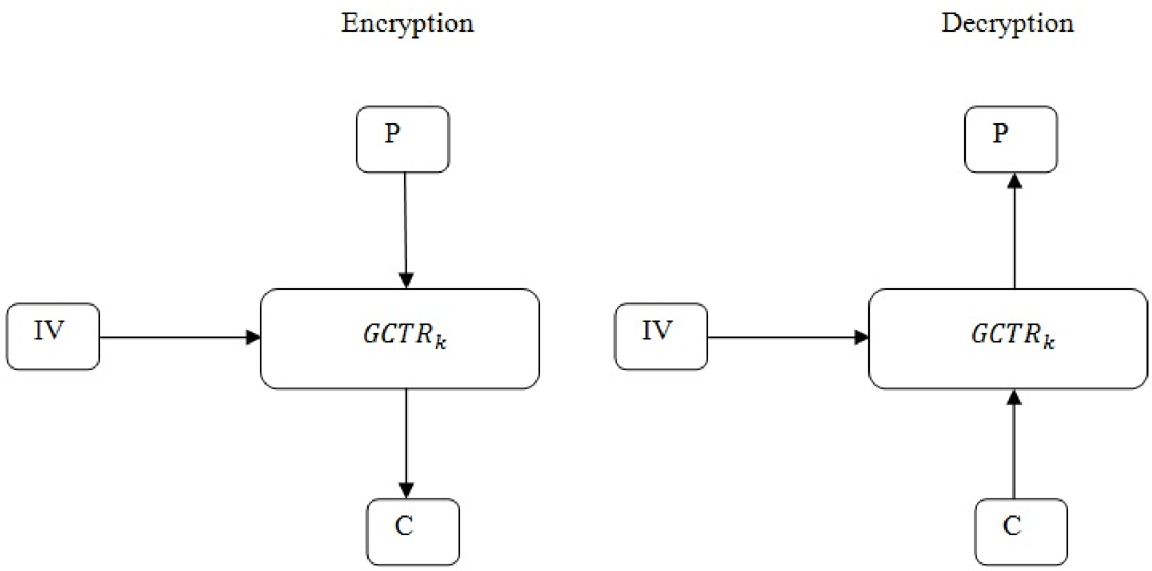

Figure 2.4: Encryption and Decryption in GCM.

Although shorter tags can be created, but due to some security concerns they are not encouraged by NIST.

To create authentication tag, the ciphertext along with unencrypted authentication data is passed through a $GHASH_H$ block and then through a $GCTR_K$ block. Similarly at the receiving end during decryption, the authentication tag is computed using the received authentication data, which is in clear, and ciphertext again using hash subkey H. The computer tag is then compared to the received tag. If the tags are the same, then it's a pass, else the authentication fails. A simplified version of authentication is shown in Fig. 2.5.

Hence, we can see from the above discussion on GCM that GCM provides authentication as well as confidentiality, and it allows some of the data to be in clear, which is the additional authentication data. Such data in practice could contain any addresses or any other information related to the encrypted data. If there is no data to be encrypted then it can just act as authentication mechanism called GMAC, which again can be classified in the category of block cipher based

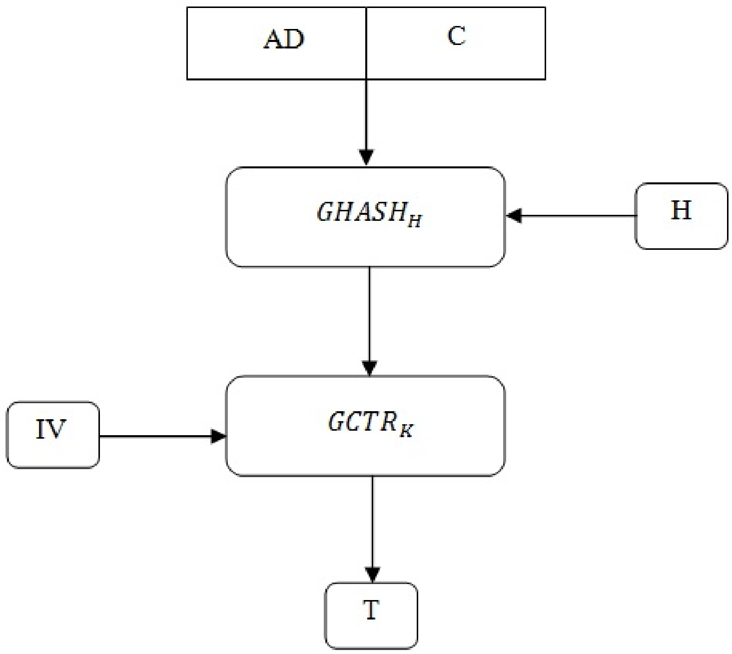

Figure 2.5: Authentication in GCM

message authentication algorithms as we discussed previously. In the next chapter, GHASH is discussed in more detail with mathematical specifications.

Chapter 3

GHASH Algorithms and Implementation Issues

As discussed in the previous chapter, one of the modes for symmetric key block cipher recommended by NIST is GCM [33], which can provide encryption/decryption and authentication (i.e., integrity of data) at the same time. In case of authentication computation, GCM has to generate a tag using keyed hash function also known as GHASH. In this chapter, the discussion will start with a brief introduction, followed by a description of the standard GHASH algorithm [28]. A brief description will also be provided on parallelized computation of GHASH as discussed in [29, 28]. We will then review a new GHASH algorithm proposed by Meloni *et al.* Finally, we discuss the carry-less multiplication schemes that can be used in GHASH computations.

3.1 Standard GHASH Computation

There have been several proposals in the past for improving GCM implementation. Some researchers have proposed faster computation of the associated symmetric block cipher itself [11, 14], while some have tried to come up with faster ways of multiplication [4, 27]. There have also been several implementations proposed for efficient implementation of AES-GCM combined [21, 9, 5]. Although these proposed schemes vary, but in their core the GHASH algorithm is almost same, that is involving as many GF(2^{128}) multiplications as the number of blocks.

The Galois counter mode of operation provides opportunities for parallelization of computation steps. However, the computation of GHASH, which involves GF(2^{128}) multiplications, poses a bottleneck to the whole operation. There has been a solution proposed by the authors of GCM itself [28], which we will see later in the chapter, but that method increases the number of required multipliers. Below, we first give a description of the standard GHASH algorithm as specified in [28, 33].

3.1.1 GHASH Description

As a brief description of how GCM handles confidentiality has been given in the previous chapter, let us assume that the associated block cipher uses block size of m bits. In order to authenticate data, the ciphertext generated by the block cipher goes through a series of GF(2^{128}) multiplications, using a key say H, which is also in GF(2^{128}). Also, assume that the block cipher being used for encryption and decryption is AES.

In order to understand the operation of GHASH, assume that there is an input data stream of bits P, divided into n blocks of size m bits. Let us represent those blocks as $P_1, P_2, P_3 \ldots\ldots P_n$, and they are all m-bit long as mentioned. There might be an exception with the last block P_n, which might not be m bits in length. In order to fix the length of the last block, if it falls short of m bits, extra 0's are padded to it to increase its length to m bits [33]. The hash key H, which we have mentioned earlier, is also m bits in length. Based on the GCM specification in [33], the input blocks are 128 bits in length and could be either actual input data, which is the output from ciphertext, or just additional authentication data (which is also divided into equal block sizes of m bits as seen in the previous chapter), but to avoid any confusion and to give a more formal definition of algorithm, we will assume that any kind of block used as an input will be represented by P_i, where $i = 1, 2, 3, \ldots. n$. So, using all these representations we can define the resultant or required GHASH as follows:

$$GHASH_H(P) = P_1 H^n + P_2 H^{n-1} + P_3 H^{n-2} + \cdots + P_n H \tag{3.1}$$

where, hash subkey H, is obtained by applying block cipher to the zero block, i.e., all of its bits are zero, and let us assume that GHASH computed using key H to be represented as $GHASH_H$.

We assume a scenario of two parties communicating using GCM-AES, and decide to use one shared key, K as the session key. Now, they can actually pre-compute H, which will be nothing but application of AES encryption using key K on a zero block. This H can also be shared between the two parties and use throughout the session, without any need to compute H every time GHASH computation is performed.

As it can be seen from Eq. (3.1), computation of GHASH is nothing but a series of multiplication and addition operations in field GF(2^m). In a more formal form, and also as described by its authors [28], the operation can be represented in Algorithm 3.1.

Algorithm 3.1 GHASH Standard [29, 33]

Input: P, H

Output: T_n

Steps: $T_0 \leftarrow 0$

for $i = 1$ **to** n **do**

  $T_i \leftarrow (T_{i-1} \oplus P_i) \cdot H$

end for

return T_n

Algorithm 3.1 can be graphically represented as Fig. 3.1. The zero block T_0 in Fig. 3.1 can be seen as the step in Algorithm 3.1,where variableT_0 is initialized to be 0 , and the tag T_n produced in the last step of figure actually represent the the computed GHASH tag.

From Fig. 3.1 it is evident that if we have n blocks of m-bits each, it will require n multiplications in GF(2^m). It can also be seen from Fig. 3.1 and Algorithm 3.1, that overall architecture of GHASH computation has essence of feedback in it. Using this feedback characteristic a more compact graphical representation using one multiplier and XOR operator in feedback can be seen in Fig. 3.2. This is a more practical approach to the GHASH implementation as it requires less hardware. On the other hand, it takes more time to compute GHASH.

In order to determine the computation time of GHASH using the feedback structure of Figure 3.2, let us assume that delay due to XOR-operation of whole block is d_{xor}, and delay due to one multiplication is d_{mul}. The total delay for computing GHASH using this architecture can be approximated as,

$$d_{total} = (d_{xor} + d_{mul}) \cdot n \tag{3.2}$$

The multiplication used in GHASH function in field GF(2^{128}) is carry-less mul-

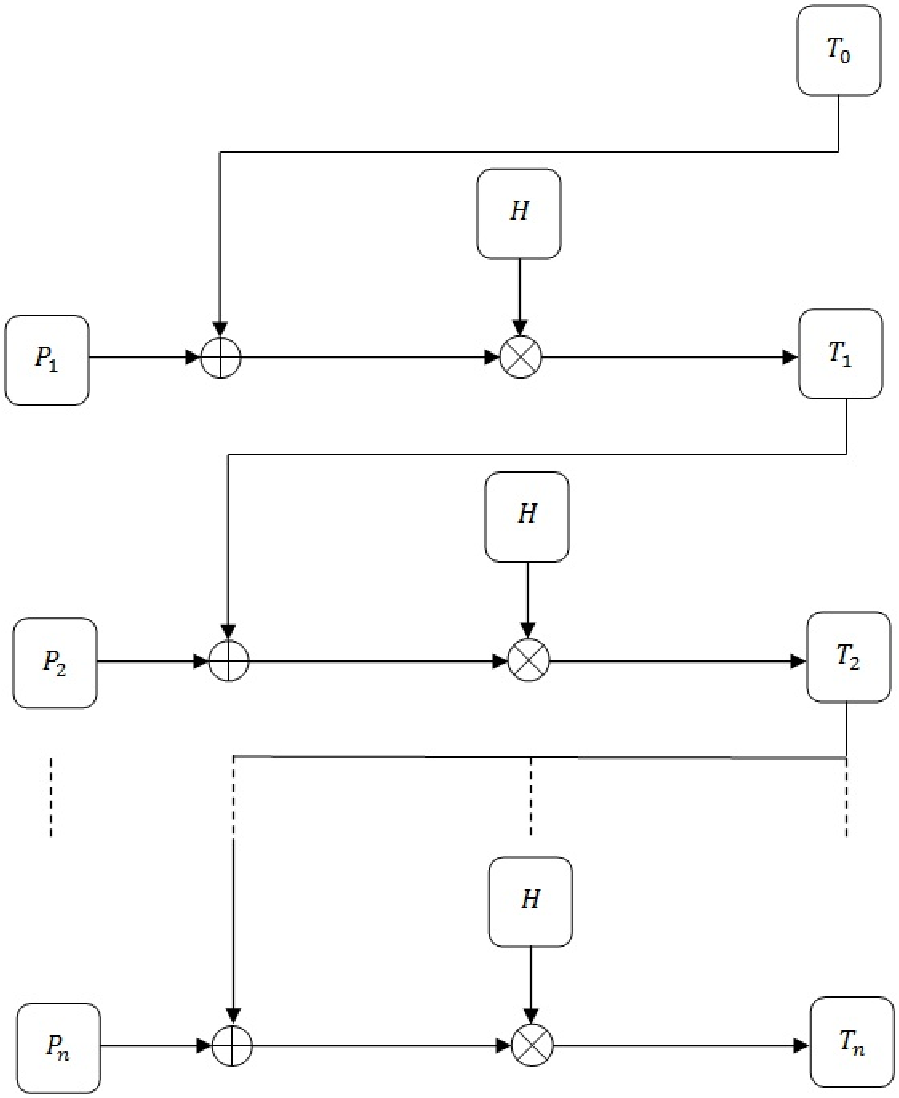

Figure 3.1: GHASH Computation

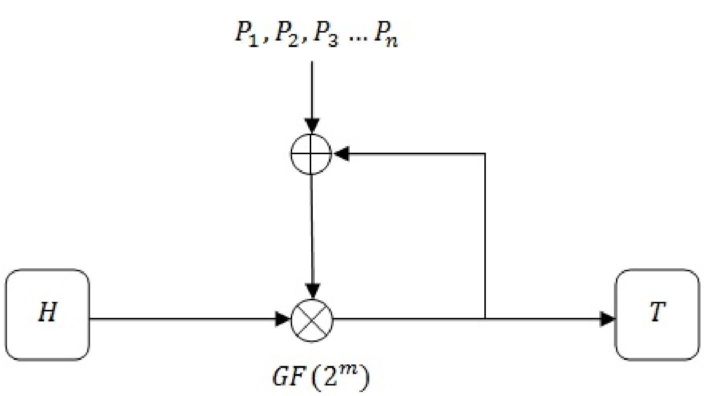

Figure 3.2: Feedback architecture for GHASH

tiplication. Intel has proposed a carry-less instruction to multiply two 64 bit operands, and has used it to come up with efficient software implementations of GHASH. It has also been mentioned earlier, GHASH operation can be parallelized, but has its restrictions in practical implementation. Before we move onto Intel's proposed scheme, we give a brief overview of parallel architecture for GHASH computation.

3.1.2 Parallel Architecture For GHASH

GHASH formulation allows its computation to be parallelized [28, 35]. To understand a parallel architecture, assume that we have g multipliers and adders, and also assume that data stream P is our input. Now for simplification we assume that g is a factor of n, where n is the number of blocks P is divided into, depending on the block size. Let us assume the block size to be m bits. Now we divide P again, but into g sections $S_1, S_2, S_3 \ldots . S_g$, with each section having $\frac{n}{g}$ blocks. From this we can now redefine the GHASH, using key H as follows [29],

$$GHASH_H = S_1 H^g + S_2 H^{g-1} + S_3 H^{g-2} + ... + S_g H \quad (3.3)$$

and we define all the S_i's as in [29],

$$S_i = P_i \left(H^g\right)^{n/g-1} + P_{i+g} \left(H^g\right)^{n/g-2} + ... + P_{n-g+i} \left(H^g\right)^0 \quad (3.4)$$

Now, these S_i's, can be computed in parallel in $(\frac{n}{g} - 1)$ steps with, a delay of $(\frac{n}{g} - 1)(d_{mul} + d_{xor})$, where d_{mul} is delay due to a single multiplier and d_{xor} is a delay due to single XOR gate. Additional multiplier delay d_{mul}, will also be included due to multiplication of all S_i's with their respective H^i's, where$1 \leq i \leq g$, and assuming that their values are already computed. We can also represent this in a more graphical form as in Fig. 3.3 [29].

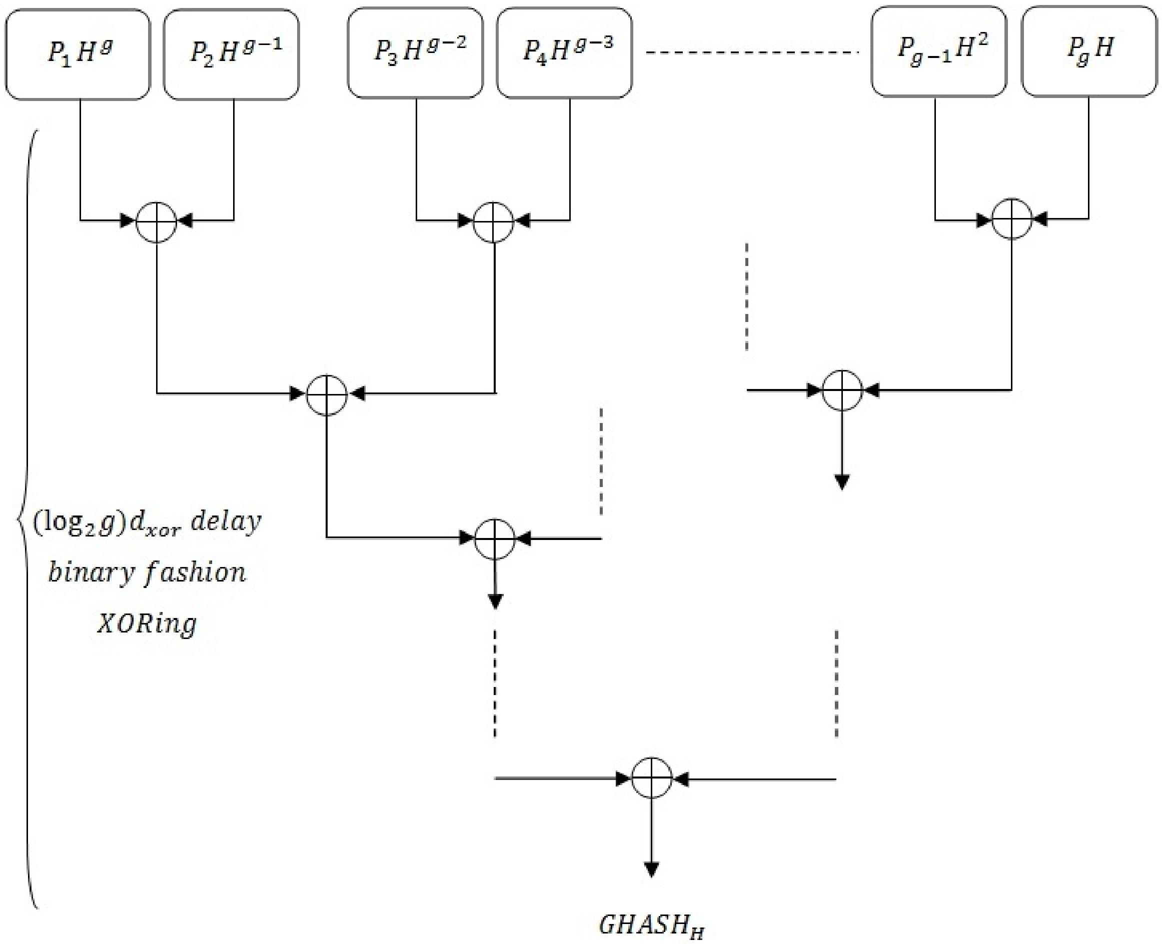

Figure 3.3: Parallel GHASH Computation

After all the parallel computations, we can add all the result in a binary tree fashion (to allow parallel XORing operations), which can be seen in the Fig. 3.3. This addition (XOR operations) will require a delay of, $\log_2 g$, and as a result will add to the delay component, and total delay then can be computed as follows,

$$d_{total} = \left(\frac{n}{g} - 1\right) \cdot (d_{mul} + d_{xor}) + d_{mul} + (\log_2 g) \cdot d_{xor} \tag{3.5}$$

3.2 Characteristic Polynomial Based GHASH

A high performance GHASH computation algorithm has been proposed in [29], based on the concept of characteristic or minimal polynomial. For the purpose of GHASH, in [29] a characteristic polynomial for an element E in field GF(2^m) is defined to be a polynomial $\chi_E(t)$ of degree m with all the coefficients belonging to GF(2) such that $\chi_E(E) = 0$. Now, in case of GHASH computation, let us assume that the characteristic polynomial for hash sub-key H be χ_H. If the characteristic polynomial is irreducible, then it can be shown that it is the minimal polynomial as defined in Chapter 2. Now let us assume that $\chi_H(H)$ can be mathematically represented as,

$$\chi_H(H) = \sum_{i=0}^{m} c_i H^i = 0 \tag{3.6}$$

Since, all the c_i's are either 0 or 1, and we know that degree of χ_H is m, hence $c_m = 1$ and we can write,

$$H^m = \sum_{i=0}^{m-1} c_i H^i \tag{3.7}$$

Now, let us consider the following polynomial of degree m

$$G = P_1 H^m + P_2 H^{m-1} + P_3 H^{m-2} + \cdots + P_m H \tag{3.8}$$

where, all the P_i's are in field GF(2^m). If we apply modular reduction on G using χ_H, we get,

$$\begin{aligned} G \bmod \chi_H = c_0 P_1 + (P_m + c_1 \cdot P_1) \cdot H + ... + \\ (P_3 + c_{m-2} P_1) \cdot H^{m-2} + (P_2 + c_{m-1} P_1) \cdot H^{m-1} \end{aligned} \tag{3.9}$$

Since, c_i's can only have a 0 or 1, the term $(P_{m-i+1} + c_i \cdot P_1)$is no computation if $c_i = 0$ and an addition in GF(2^m) if $c_i = 1$. These operations can be represented in form of a circuit as shown in Fig. 3.4.The registers shown in the figure are loaded in the following sequence: $P_1, P_2, ..., P_n \rightarrow Y_{m-1}, Y_{m-2} ... Y_0$. We can see from Fig.

3.4 and Eq. (3.9) that, the addition computations can be peroformed in parallel. This circuit to perform these parallel operations is called Polynomial Reduction Unit (PRU) [29].

Now, let us consider a polynomial similar to Eq. (3.8) but of degree $n > m$ and we can break that polynomial as follows,

$$G = ((...((P_1 H^m + P_2 H^{m-1} + ... + P_{m+1})H + P_{m+2})H + ... + P_{n-1})H + P_n)H$$

which can be simplified after applying modulo reduction χ_H as [29],

$$\begin{aligned} G \bmod \chi_H = \quad & ((...((P_1 H^m + P_2 H^{m-1} + ... + P_{m+1} \bmod \chi_H)H \\ & +P_{m+2} \bmod \chi_H)H + ... + P_{n-1} \bmod \chi_H)H \\ & +P_n \bmod \chi_H)H \bmod \chi_H \end{aligned} \tag{3.10}$$

Basically, the idea here is to replace $n-m+1$ multiplications by H with that many polynomial reductions using a circuit shown in Fig. 3.4. In a more formal way and as defined in [29] the algorithm can be represented as in Algorithm 3.2. For the sake of simplicity this GHASH algorithm, which is proposed by Meloni, Negre, and Hasan [29], and from here and onwards will be referred to as the MNH GHASH algorithm.

We clearly see that compared to older Algorithm 3.1, the new one requires fewer number of multiplications when $n \geq m$. For example, if we have n blocks to compute GHASH, the older algorithm will require n multiplications, but the MNH algorithm restricts the number of multiplications to $m - 1$, and replaces rest of multilpications with $n - m + 1$ parallel rounds of a PRU, which mainly comprises of XOR operation, and AND operations for fixed c_i's. One important thing to note here is that, inside the first loop, the computations of all the Y_i's , and Y_0, represent operation of PRU, and are computed in parallel at every iteration of j.

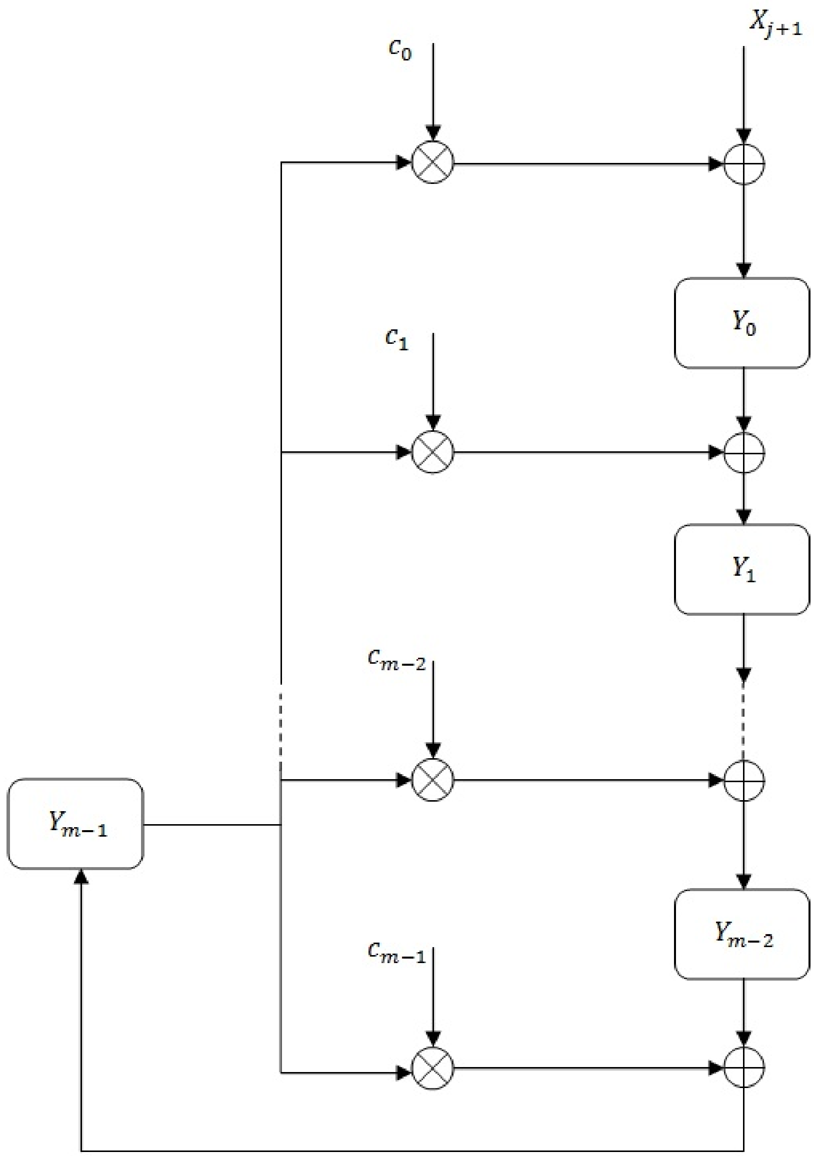

Figure 3.4: Polynomial Reduction Unit

Algorithm 3.2 MNH GHASH Algorithm

Input: $P = P_1, P_2, ..., P_n$, $\chi_H(H) = \sum_{i=0}^{m} c_i H^i$ where, $(n \geq m)$

Output: $GHASH_H(P) = P_1 H^n + P_2 H^{n-1} + P_3 H^{n-2} + \cdots + P_n H$

Steps:

$P_1, P_2, ..., P_n \rightarrow Y_{m-1}, Y_{m-2}...Y_0$

$T \rightarrow 0,\ P_{n+1} = 0$

for $j = m$ **to** n **do**

$Y_{m-1} \rightarrow C$

$Y_i \leftarrow Y_{i-1} + c_i C,\ m-1 \geq i \geq 1 \begin{cases} in \quad parallel \end{cases}$

$Y_0 \leftarrow P_{j+1} + c_0 C \begin{cases} done & in \\ parallel & with \\ Y_i s \end{cases}$

endfor

for $i = m-1$ **down to** 1 **do**

$T \leftarrow (T + Y_i) \cdot H$

endfor

return $(T + Y_0)$

Such parallel computations are easily possible in special purpose hardware. On the hand, this is not the case in software using general purpose processors. In software these computations are most likely to be performed in sequence which will be discussed in the next chapter. A brief description will be also given on how to compute the characteristic polynomial required for the operation of algorithm. For now, we give an example to clarify the operation of Algorithm 3.2.

Let us assume that, we have GF(2^4), i.e., $m = 4$, and the reduction polynomial is $x^4 + x + 1$. Now, let us assume that we have five blocks $P_1 = 1, P_2 = x, P_3 = x + 1, P_4 = x^2, P_5 = x^2 + 1$, for computation of $GHASH_H$, where $H = x^3$ and $P_6 = 0$. Now, from the assigned value of H, we can get value of c_i's through characteristic polynomial which is $x^4 + x^3 + x^2 + x + 1$ for the given H. The expression for $GHASH_H$ is

$$GHASH_H(P) = P_1H^5 + P_2H^4 + P_3H^3 + P_4H^2 + P_5H \tag{3.11}$$

Now, using the MNH GHASH algorithm we will first assign input block values to PRU registers as follows,

$$\begin{aligned}
Y_3 &= P_1 = 1 \\
Y_2 &= P_2 = x \\
Y_1 &= P_3 = 1 + x \\
Y_0 &= P_4 = x^2
\end{aligned}$$

Now, to apply the PRU iteration for $j = 4$,

$$\begin{aligned}
C &= Y_3 = 1 \\
Y_3 &= Y_2 + C = x + 1 \\
Y_2 &= Y_1 + C = x \\
Y_1 &= Y_0 + C = x^2 + 1 \\
Y_0 &= P_5 + C = x^2
\end{aligned}$$

Again, applying PRU iteration for $j = 5$,

$$
\begin{gathered}
C = Y_3 = x + 1 \\
Y_3 = Y_2 + C = 1 \\
Y_2 = Y_1 + C = x^2 + x \\
Y_1 = Y_0 + C = x^2 + x + 1 \\
Y_0 = P_6 + C = x + 1
\end{gathered}
$$

Now, applying iterations of multiplicaiton loop, starting with $i = 3$ and $T = 0$,

$$T = (T + Y_3) \cdot H = (0 + 1) \cdot x^3 = x^3$$

for $i = 2$,

$$T = (T + Y_2) \cdot H = (x^3 + x^2 + x) \cdot x^3 = 1 + x^3$$

again, for $i = 1$,

$$T = (T + Y_1) \cdot H = (x^3 + 1 + x^2 + x + 1) \cdot x^3 = 1 + x^3$$

Now, for the final step we add Y_0 and T,

$$GHASH_H(P) = Y_0 + T = 1 + x + 1 + x^3 = x^3 + x$$

As mentioned earlier the MNH algorithm restricts combined multiplication and XOR operations to $m-1$, and rest of the multiplicaitons are replaced by $n-m+1$ PRU operations. Hence, the GHASH computation time using the MNH algorithm is

$$d_{total} = (n - m + 1) \cdot (d_{xor} + d_{and}) + (m - 1)\,(d_{mul} + d_{xor}) \tag{3.12}$$

3.3 Implementation Issues in Software

The most challenging task in the software implementation of GHASH using either the standard or the MNH algorithm is the multiplication in GF(2^{128}). As mentioned earlier, such multiplication can be performed by first multiplying two polynomials of degree less than 128 over the ground field GF(2) and then reducing the resultant polynomial of degree 254 or less using the field defining polynomial of degree 128. The polynomial multiplication over GF(2) can be viewed as a carry-less multiplication, which is described below. We also present Intel's new instruction to speed-up such carry-less multiplication and its use to the Karatsuba algorithm.

3.3.1 Carry-less Multiplication

Carry-less multiplication in simple words can be defined as multiplication of two operands, with no propagation and generation of carries during the process. Let us assume that we have two operands, X & Y and we represent them in an array of m bits.

$$X = [x_1, x_2, x_3, ..., x_m]$$

$$Y = [y_1, y_2, y_3, ..., y_m]$$

The carry-less product generated will be of size $2m - 1$ bits and let us call it Z, where Z can be represented as,

$$Z = [z_{2m-1}, z_{2m-2}, z_{2m-3}, ..., z_2, z_1]$$

The resultant or output of multiplication can mathematically be also represented as in [17],

$$z_i = \begin{cases} \bigoplus\limits_{j=1}^{i} x_j y_{i-j}, & 1 \le i \le m, \\ \bigoplus\limits_{j=i-m+1}^{m} x_j y_{i-j}, & m+1 \le i \le 2m-1 \end{cases}$$

It is also evident from the equation above the result is similar to integer multiplication, but without any carry. A small example to understand it in a better way is as follows, assume $X = [1100]$ and $Y = [1100]$.

$$
\begin{array}{c}
1100 \\
1100 \\
- - -- \\
0000 \\
0000\times \\
1100 \times \times \\
1100 \times \times\times \\
- - - - - - - \\
1010000
\end{array}
$$

As, it can be seen the result of normal multiplication of X and Y should be 144 i.e., in binary [10010000], but result of carry-less muliplication is [1010000], which is equvalent to 80.

3.3.2 Efficient Carry-less Multiplication for Large Operands

For GHASH, the size of the operands for carry-less multiplication is m or 128 bits. There are basically two types of techniques used in efficient software implementations of carry-less multiplication of such large size operands: look-up table and the Karatsuba methods.

Look-up table based implementation is based on two major steps. First is preprocessing, where all the tables are generated in $GF(2^{128})$ and stored. Second step involves finding the right matches based on the given input and XOR all the matches to obtain the output. Further details for the look-up table based is not that relevant for our discussion, so has been avoided, but the key idea is that, the scheme involves memory storage cost and can be inefficient when high performance (speed) is required.

The other popular technique is to use a carry-less Karatsuba algorithm. The multiplication is then followed by a reduction algorithm. A brief introduction to Karatsuba algorithm is presented, followed by a brief description of the modified Karatsuba algorithm.

3.3.3 Basics of Karatsuba Algorithm

The Karatsuba algorithm was named after its inventor, Anatolii A. Karatsuba. The Karatsuba algorithm enables faster multiplication of two n-digit numbers, and has proven to be faster than traditional algorithms. The older algorithm also called ordinary multiplication (OML) [20, 19], has an algorithmic complexity of $O(n^2)$, which is reduced to $O(n^{log_2 3})$ by the Karatsuba algorithm.

In order to get better understanding of algorithm, assume two n-digits numbers, a and b, and let them be in base B. Also, assume a positive integer less than n and call it x, such that we can divide the two numbers as follows,

$$a = a_1 \cdot B^x + a_0, \tag{3.13}$$

$$b = b_1 \cdot B^x + b_0. \tag{3.14}$$

We also have to make sure that B^x, is greater than a_0 and b_0, and as a result product of a and b, p can be represented as

$$p = a \cdot b = c_2 \cdot B^{2x} + c_1 B^x + c_0 \tag{3.15}$$

where,

$$c_2 = a_1 \cdot b_1$$

$$c_1 = a_1 \cdot b_0 + a_0 b_1$$

$$c_0 = a_0 \cdot b_0$$

Now, it appears that we need to perform four multiplications, but Karatsuba, reduced these to three multiplications at the cost of extra additions as follows,

$$c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - c_2 - c_0$$

So, by computing c_1 as above the number of multiplications has been reduced by one. Let us see a small example to verify working of this algorithm. Let us assume that we want to multiply two 3-digit numbers, $a = 123$ and $b = 456$, the base used is 10, and the value of x used is 1. So, we can split those two numbers like Eq. (3.13) and Eq. (3.14),

$$a = 123 = 12 \cdot 10^1 + 3$$

$$b = 456 = 45 \cdot 10^1 + 6$$

So, values of c_2, c_1, and c_0, can be computed as,

$$c_2 = a_1 \cdot b_1 = 12 \cdot 45 = 540$$

$$c_0 = a_0 \cdot b_0 = 3 \cdot 6 = 18$$

$$c_1 = (a_1 + a_0) \cdot (b_1 + b_0) - c_2 - c_0 = (12 + 3) \cdot (45 + 6) - 540 - 18 = 207$$

Now, using Eq. (3.15), we can compute the final result as,

$$p = a \cdot b = c_2 \cdot B^{2x} + c_1 B^x + c_0 = 540 \cdot 10^2 + 207 \cdot 10 + 18 = 56088$$

3.3.4 Karatsuba Algorithm for GHASH

Intel has proposed the use of the Karatsuba algorithm for computing GF (2^{128}) field multiplications, using their PCLMULQDQ instruction. Since the instruction can multiply operands of 64-bits long, only one recursion of the Karatsuba algorithm is needed. If we assume that we have two 128-bit operands, A and B, then to apply the Karatsuba algorithm we will need to divide them into two parts each 64 bit long represented as $A[A1 : A0]$ and $B[B1 : B0]$, where ":" corresponds to concatenation [16]. As we saw in the previous section, for the Karatsuba algorithm we compute c_2, c_1 and c_0; here we compute their equivalents, $[G_1 : G_0]$, $[E_1 : E_0]$ and $[D_1 : D_0]$, respectively. To understand the multiplication by splitting into two 64 bit halves, let us assume that in polynomial form A and B can be represented as (addition in following equations is not normal addition, but addition used in field arithmetic, which is equivalent to XOR operation),

$$A = A_1x^{64} + A_0$$

$$B = B_1x^{64} + B_0$$

Similarly,$[G_1 : G_0]$, $[E_1 : E_0]$ and $[D_1 : D_0]$ can be computed as,

$$A_1B_1 = [G_1 : G_0] = G_1x^{64} + G_0$$

$$A_0B_0 = [D_1 : D_0] = D_1x^{64} + D_0$$

$$(A_0 + A_1) \cdot (B_1 + B_0) = [E_1 : E_0] = E_1x^{64} + E_0$$

and we can write the product of A and B as,

$$\begin{aligned} A \cdot B &= \left(A_1x^{64} + A_0\right) \cdot \left(B_1x^{64} + B_0\right) \\ A \cdot B &= A_1B_1x^{128} + ((A_0 + A_1) \cdot (B_1 + B_0) + A_1B_1 + A_0B_0) \\ &\quad x^{64} + A_0B_0 \end{aligned} \tag{3.16}$$

Now, substituting values of $[G_1 : G_0]$, $[E_1 : E_0]$ and $[D_1 : D_0]$, in Eq. (3.16), and simplifying we get,

$$A \cdot B = G_1 x^{192} + (G_1 + G_0 + D_1 + E_1)\, x^{128} + (D_1 + D_0 + G_0 + E_0)\, x^{64} + D_0 \quad (3.17)$$

So, Eq. (3.17) represents the final product and shows how using 64 bit halves the carry-less multiplication of 128-bit operands can be performed. Implementation details will be discussed in the next chapter.

Next step right after multiplication is modulo reduction of multiplication result using $g(x) = x^{128} + x^7 + x^2 + x + 1$. In terms of implementation, this can be achieved in a simpler way, by only using some shifts and XOR operations which we will see in the next chapter.

Chapter 4

Software Implementation of GHASH Algorithms

In the previous chapter we discussed the standard GHASH algorithm [33] and the new MNH GHASH algorithm [29]. There we also mentioned about Intel's new instruction PCLMULQDQ [17] and its usage towards the GHASH computation. In this current chapter we will look at the performance of the old and the new GHASH algorithms. Here our discussion will be primarily based on software implementations of those two GHASH algorithms. At the start of the chapter, we will look into the field multiplication algorithm suggested in [33], and how a modified implementation of this algorithm is done, followed by a small example in $\mathrm{GF}(2^4)$. A brief descripton of Gordon's algorithm for computing characteristic polynomials is presented, again followed by a small example in $\mathrm{GF}(2^4)$. In the later sections of the chapter, GHASH implementation using Intel's carry-less instruction PCLMULQDQ is presented. The chapter also includes implementation results and a comparison between the standard and the MNH GHASH algotithms.

4.1 GHASH Building Blocks

It is evident from the discussion in the previous two chapters that the most important part of GHASH computation is carry-less multiplication. If we closely look at the standard way of computing GHASH, it is nothing but field multiplication and XOR operation done repeatedly. The carry-less multiplication algorithm used in our peformance comparison is the modified implementation of multiplication algorithm proposed in [33]. In order to implement the MNH GHASH algorithm, one of the most important ingredients required is computation of the characteristic polynomial χ_H, for the corresponding sub-key H [29]. In order to compute characteristic polynomial, implementation of Gordon's algorithm, as presented in [29], has been implemented. Before, we can discuss the implementation results, brief discussions with examples are provided on the modified multiplication algorithm, Gordon algorithm, and algorithms needed to utilize PCLMULQDQ instruction.

4.1.1 Implementation of Standard GF(2^m) Multiplication

An algorithm of multiplication in GF(2^{128}) has been given in [33]. The operations involved in it are based on right shifts and XORing. The algorithm's implementation is modified to use the left shift operations instead of the right shift. The modification allows us to avoid bit reflection, which was required previously as mentioned in [33, 17], and it also makes algorithm easier to understand. The modified scheme is described in Algorithm 4.1, and in the literature it is known as the least significant bit first multiplication algorithm.

In order to clarify the working of Algorithm 4.1, we give a small example. Let us start by assuming that we have two 4-bit blocks, $A = 0110$ and $B = 0011$ in field GF(2^4). The field polynomial used for modulo reduction is $x^4 + x + 1$, i.e.,

Algorithm 4.1 Least Significant Bit First Multiplication.

Input : A, B (Input blocks) and R is reduction polynomial block

Output : $D_m = A \cdot B$

Steps :

$A \rightarrow a_{m-1}a_{m-2}...a_2a_1a_0$(Input block A represented as bits string)

$D_0 \leftarrow 00\cdots0,\ E_0 \leftarrow B$

for $i = 0$ **to** $m-1$ **do**

$$D_{i+1} \leftarrow \begin{cases} D_i & if\ a_i = 0 \\ D_i \oplus E_i & if\ a_i = 1 \end{cases}$$

$$E_{i+1} \leftarrow \begin{cases} E_i \ll 1 & if\ MSB(E_i) = 0 \\ (E_i \ll 1) \oplus R & if\ MSB(E_i) = 1 \end{cases}$$

endfor

return D_m

$x^4 \equiv x + 1$, which is represented as a 4-bit block, $R = 0011$. Let us initialize the values of D_0 and E_0 as follows

$$D_0 = 0^4 = 0000,$$
$$E_0 = B = 0011.$$

Now, we can also represent block A as

$$A = a_3a_2a_1a_0 = 0110$$

Now, going thorugh iterations of the for loop, for $i = 0$, $a_0 = 0$ and $MSB(E_0) = 0$ we have,

$$D_1 = D_0 = 0000$$
$$E_1 = E_0 \ll 1 = 0110$$

Now, for $i = 1$, $a_1 = 1$ and $MSB(E_1) = 0$ we have,

$$D_2 = D_1 \oplus E_1 = 0110$$
$$E_2 = E_1 \ll 1 = 1100$$

Now, for $i = 2$, $a_2 = 1$ and $MSB(E_2) = 1$ we have,

$$D_3 = D_2 \oplus E_2 = 1010$$
$$E_3 = (E_2 \ll 1) \oplus R = 1011$$

Again, for $i = 3$, $a_3 = 0$ and $MSB(E_3) = 1$ we have our result D_4 as follows,

$$D_4 = D_3 = 1010.$$

Hence, our result for the carry-less multiplication of blocks $A = 0110$ and $B = 0011$, over field GF(2^4) using Algorithm 4.1 is 1010.

4.1.2 Gordon's Algorithm

In order to calculate the characteristic polynomial, Gordon's method is used, which can be represented in mathematical form as follows,

$$\chi_H(t) = \prod_{i=0}^{m-1} \left(t + H^{2^i} \right) \tag{4.1}$$

where, χ_H is the characteristic polynomial of hash sub-key H, and where, $H \in$ GF(2^m), this can be reresented in form of Algorithm 4.2 [29].

The **for** loop in Algorithm 4.2 starts from $i = 1$, not $i = 0$ as suggested by Eq. (4.1). It is because the initialization step $\chi_H \leftarrow t + H$ already represents the stage when $i = 0$. In order to clarify the working of Algorithm 4.2, let us assume that we have, $H = x^3$. The field we are using is, GF(2^4), and the field polynomial $x^4 + x + 1$. Let us begin with the initialization step,

$$\chi_H = t + H = t + x^3$$
$$Z = H = x^3$$

Now, the first iteration of for loop, when $i = 1$,

Algorithm 4.2 Gordon's Algorithm [29]

Input : $H \in \mathrm{GF}(2^m)$

Output : χ_H (characteristic polynomial of H)

Steps :

$\chi_H \leftarrow t + H$

$Z \leftarrow H,$

for $i = 1$ **to** $m - 1$ **do**

$Z \leftarrow Z^2$

$\chi_H \leftarrow \chi_H \cdot t + \chi_H \cdot Z$

endfor

return χ_H

$$Z = Z^2 = (x^3)^2 = x^6 = x^3 + x^2$$

$$\begin{aligned}
\chi_H &= \chi_H \cdot t + \chi_H \cdot Z \\
\chi_H &= (t + x^3) \cdot t + (t + x^3) \cdot (x^3 + x^2) \\
\chi_H &= t^2 + tx^2 + x^3 + x
\end{aligned}$$

When $i = 2$,

$$Z = Z^2 = (x^3 + x^2)^2 = x^3 + x^2 + x + 1$$

$$\begin{aligned}
\chi_H &= \chi_H \cdot t + \chi_H \cdot Z \\
\chi_H &= (t^2 + tx^2 + x^3 + x) \cdot t + (t^2 + tx^2 + x^3 + x) \cdot (x^3 + x^2 + x + 1) \\
\chi_H &= t^3 + (1 + x + x^3)t^2 + (1 + x)t + x^2 + x^3
\end{aligned}$$

Again when $i = 3$,

$$Z = Z^2 = (x^3 + x^2 + x + 1)^2 = x^3 + x$$

$$\begin{aligned}
\chi_H &= \chi_H \cdot t + \chi_H \cdot Z \\
\chi_H &= (t^3 + (1 + x + x^3)t^2 + (1 + x)t + x^2 + x^3) \cdot t \\
&\quad +(t^3 + (1 + x + x^3)t^2 + (1 + x)t + x^2 + x^3) \cdot (x + x^3) \\
\chi_H &= t^4 + t^3 + t^2 + t + 1
\end{aligned}$$

Now, χ_H is our required characteristic polynomial.

In order to implement Gordon's algorithm, only two major components are needed: a field multiplier and a XOR operator. For XOR operations, Intel's intrinsic XOR operation is used, and for field multiplications, the carry-less multplier modified implementation discussed in the previous Section 4.1.1 is used.

4.1.3 Intel's PCLMULQDQ instruction

Intel proposed the PCLMULQDQ instruction in 2010, for carry-less multiplication on their Westmere architecture [17]. This instruction can be used to multiply two operands, which are 64 bits in length. This instruction provides a faster way of computing carry-less multiplication as compared to the methods available before it [17]. This instruction can further be used to compute carry-less multiplication of two 128-bit operands, as we will see in the next subsection. In its assembly usage form this instruction can be written as [17],

pclmulqdq immbyte, reg1, reg2

where, $reg1$ and $reg2$ are two 128-bit registers. The carry-less multiplication is performed on a quadword (8 bytes) of $reg1$ and a quadword of register $reg2$. The selection of the quadwords from $reg1$ and $reg2$ depends on the value of *immbyte* (the result gets stored in $reg2$, and in C instruction can be used by calling a function which returns the result of multiplication). If we assume that $reg1$, $reg2$ and *immbyte* are represented by referring to their number of bits as

$$\text{reg1 } [127:0]$$
$$\text{reg2 } [127:0]$$
$$\text{immbyte } [7:0]$$

then, we can rperesent the selection of quadwords, on basis of *immbyte* values as in Table 4.1,

immbyte (in hex)	Quadword Selection
0x00	reg2 [63 : 0], reg1[63 : 0]
0x01	reg2 [63 : 0], reg1[127 : 64]
0x10	reg2 [127 : 64], reg1[63 : 0]
0x11	reg2 [127 : 64], reg1[127 : 64]

Table 4.1: Selection of quadwords

In terms of software implementation intrinsic function for the PCLMULQDQ can be used. Intel allows the use of the intrinsic function without explicitly specifying PCLMULQDQ [2]. A small example is also given on how to use this intrinsic function in [2], in C language. The intrinsic function *_mm_clmulepi64_si128( )*, can be formally defined as [2],

```
__m128i _mm_clmulepi64_si128 (__m128i a1, __m128i a2, const int immbyte )
```

The definition above means that function returns a value of type __m128i, and as inputs it takes two 128-bit parameters and a constant integer *immbyte*, which decides the halves of $reg1$ and $reg2$ are to be taken for multiplication using the criteria as shown in Table 4.1.

4.1.4 Intel's Karatsuba Implementation using PCLMULQDQ

In chapter 3, we have already discussed Intel's modified Karatsuba algorithm and how it works in terms of polynomial arithmetic. In this subsection we will look more closely in terms of implementation. A more formal definition of Intel's carry-less Karatsuba algorithm as proposed in [16] is presented in Algorithm 4.3.

In Algorithm 4.3, X and Y represent the two blocks to be multiplied and are divided into two halves. In case of GF(2^{128}) field, X_1 and Y_1 are the upper 64-bit

Algorithm 4.3 Intel's modified Karatsuba algorithm [16]

Input : $X = [X_1 \ : \ X_0]$, $Y = [Y_1 \ : \ Y_0]$

Output : $X \cdot Y$

Steps :

$$\begin{aligned}
[Z_1 \ : \ Z_0\,] &= X_1 \cdot Y_1 \\
[W_1 \ : \ W_0] &= X_0 \cdot Y_0 \\
[V_1 \ : \ V_0\,] &= (X_1 \oplus X_0) \cdot (Y_1 \oplus Y_0) \\
X \cdot Y &= [Z_1 \ : \ Z_0 \oplus Z_1 \oplus W_1 \oplus V_1 \ : \ W_1 \oplus Z_0 \oplus W_0 \oplus V_0 \ : \ W_0]
\end{aligned}$$

Return $X \cdot Y$

halves, and X_0, Y_0 represent the lower 64-bit halves of 128-bit operands, which are X and Y respectively. The symbol ":" represents concatenation of blocks, and the symbol "$\cdot$" represents carry-less multiplication operator.

Below we use a small example to clarify the working of Algorithm 4.3. In order to keep things simple, assume that we have two blocks: X and Y in a field GF(2^4). These blocks can be divided into two halves of 2 bits each to keep the representation consistent with Algorithm 4.3.

$$\begin{aligned}
X &= [\,X_1 \ : \ X_0\,] = [01:10] \\
Y &= [\,Y_1 \ : \ Y_0\,\,] = [00:11]
\end{aligned}$$

Th expected result of multiplication is 1010, and the steps of operation can be seen below, where "$\cdot$" represents carry-less multiplication with XOR operations in the third step.

$$\begin{array}{ccccccc}
[\,Z_1 \ : \ Z_0\,] & = & X_1 \cdot Y_1 & = & 01 \cdot 00 & = & [00:00] \\
[\,W_1 \ : \ W_0\,] & = & X_0 \cdot Y_0 & = & 10 \cdot 11 & = & [01:10] \\
[\,V_1 \ : \ V_0\,] & = & (X_1 \oplus X_0) \cdot (Y_1 \oplus Y_0) & = & 11 \cdot 11 & = & [01:01]
\end{array}$$

Now, the final step of computing product involves XOR operations, and concatenating four 64-bit blocks to produce a 256-bit output.

$$\begin{aligned} X \cdot Y &= [Z_1 \ : \ Z_0 \oplus Z_1 \oplus W_1 \oplus V_1 \ : \ W_1 \oplus Z_0 \oplus W_0 \oplus V_0 \ : \ W_0] \\ X \cdot Y &= [00 : 00 \oplus 00 \oplus 01 \oplus 01 : 01 \oplus 00 \oplus 10 \oplus 01 : 10] \\ X \cdot Y &= [00 : 00 : 10 : 10] \end{aligned}$$

Hence, the result is same as expected result.

As we can see from Algorithm 4.3, the first three steps involve only multiplication, and the last step involves multiple XOR operations. In terms of implementation, the three carry-less multiplications are implemented using the PCLMULQDQ instruction, in the same manner as we discussed in last subsection. Implementation of the XOR operations can be again done using the intrinsic function for XORing two 128-bit operands. Similar to PCLMULQDQ, intrinsic funtion for XOR operation defined in [1], can be represented as shown below

```
__m128i _mm_xor_si128 (__m128i x, __m128i y)
```

where x and y are two 128-bit operands.

4.1.5 Efficient Reduction Modulo Implementation

In [16, 17] Intel has proposed a modular reduction algorithm by taking into consideration field defining polynomial $x^{128} + x^7 + x^2 + x + 1$. The algorithm is then implemented in combination with the carry-less Karatsuba algorithm explained in the previous section. A more formal description of this modular reduction is given in Algorithm 4.4.

In terms of implementation, a combined implementation of this algorithm with carry-less Karatsuba as presented in [17] is used. The combined implementation

Algorithm 4.4 Modular Reduction in GF(2^{128}) [17, 16]

Input : $[X_4,\ X_3,\ X_2,\ X_1]$, where X_4, X_3, X_2, X_1, are each 64-bit long.

Output : $[Y_1,\ Y_0]$ (128-bit long reduciton result, where Y_1, Y_0, are each 64-bit long)

Steps :

$U\ = X_4 \gg 63$

$V\ = X_4 \gg 62$

$W\ = X_4 \gg 57$

$Z = U \oplus V \oplus W \oplus X_3$

Now, using Z we form $[X_4\ :\ Z]$, and proceed as follows,

$[P_1\ :\ P_0]\ = [X_4\ :\ Z] \ll 1$

$[Q_1\ :\ Q_0]\ = [X_4\ :\ Z] \ll 2$

$[R_1\ :\ R_0]\ = [X_4\ :\ Z] \ll 7$

$[H_1\ :\ H_0]\ = [P_1\ :\ P_0] \oplus [Q_1\ :\ Q_0] \oplus [R_1\ :\ R_0] \oplus [X_4\ :\ Z]$

$Y_1 = H_1 \oplus X_2$

$Y_0 = H_0 \oplus X_1$

Return $[Y_1,\ Y_0]$

serves the purpose of GF(2^{128}) field multiplication, and then using it implementations of the standard and the MNH GHASH algorithms are done.

4.2 GHASH Implementation Results

4.2.1 Implementation using Common Place Instructions

A software implementation of the standard and the MNH GHASH function has been done using the C programming language and without using Intel's special carry-less multiplication instruction. The 128-bit multiplication is performed by using the modified version of the algorithm provided by NIST in [33] (see Section 4.1.1). The standard GHASH implementation can be viewed in Appendix A.1. In order to compute the characteristic polynomial, a Maple code has been written. The input or hash sub-key value (H), used for the Maple code is selected to be a large 128-bit random value with half of its bits are 1 and half of it are 0. The result of the Maple code is in polynomial form, and then that resultant characteristic polynomial is used in the C code for the MNH GHASH in form of an array of 1's and 0's. The MNH GHASH implementation has been included in Appendix A.2. Both GHASH algorithms are compared in terms of computation time. In order to increase accuracy of timing result each algorithm was run 10,000 times for each value of input blocks (each block is 128 bits long), and then average time was obtained by dividing the total by 10,000. The system used for running the implementations was Xeon E3-1270 (quad-core 3.4GHz) and the operating system used was Linux Ubuntu Server 11.10 (with gcc 4.6.1). Computation time was calculated using *'time'* command in Linux. The values of the computation time for the standard and high performance GHASH can be seen in Table 4.2, and a

No. of Blocks	Standard GHASH (x 10^{-4}sec)	MNH GHASH(x 10^{-4}sec)
128	3.056	3.000
192	4.681	3.316
256	6.236	3.615
384	9.357	4.179
512	12.470	4.758
768	18.707	5.855
1024	24.930	6.980
1280	31.170	8.144
1536	37.400	9.240

Table 4.2: Computation Time of Implementations with Customary Instructions

graphical representation of results can be seen in Fig. 4.1.

As it can be seen from Fig. 4.1, the MNH GHASH shows smaller delay than the standard GHASH, and the delay result improves as the number of blocks increases. The improved delay is due to the fact that the MNH GHASH algorithm keeps the number of 128-bit multiplication operations fixed at 127. The rest of the 128-bit multiplications, which are part of the standard GHASH algorithm, are replaced by PRU computations as discussed in Section 3.2. PRU computations in terms of implementation are much more faster than the implementation of 128-bit multiplication. In case of software implmentation, the PRU operations are not implemented in parallel, but rather computed sequentially, as it is not feasible to compute 128 operations exactly in parallel using software implementation on our processors. It is interesting to note that even though the PRU operations are not occuring in parallel, but still the new algorithm gives better results than the standard one.

4.2.2 Implementation using PCLMULQDQ Instruction

The GHASH algorithms have been implemented using Intel's PCLMULQDQ instruction. In order to perform 128-bit multiplication, the algorithm mentioned by Intel in [17], and which we also discussed in Section 4.1.4 is used. The implementa-

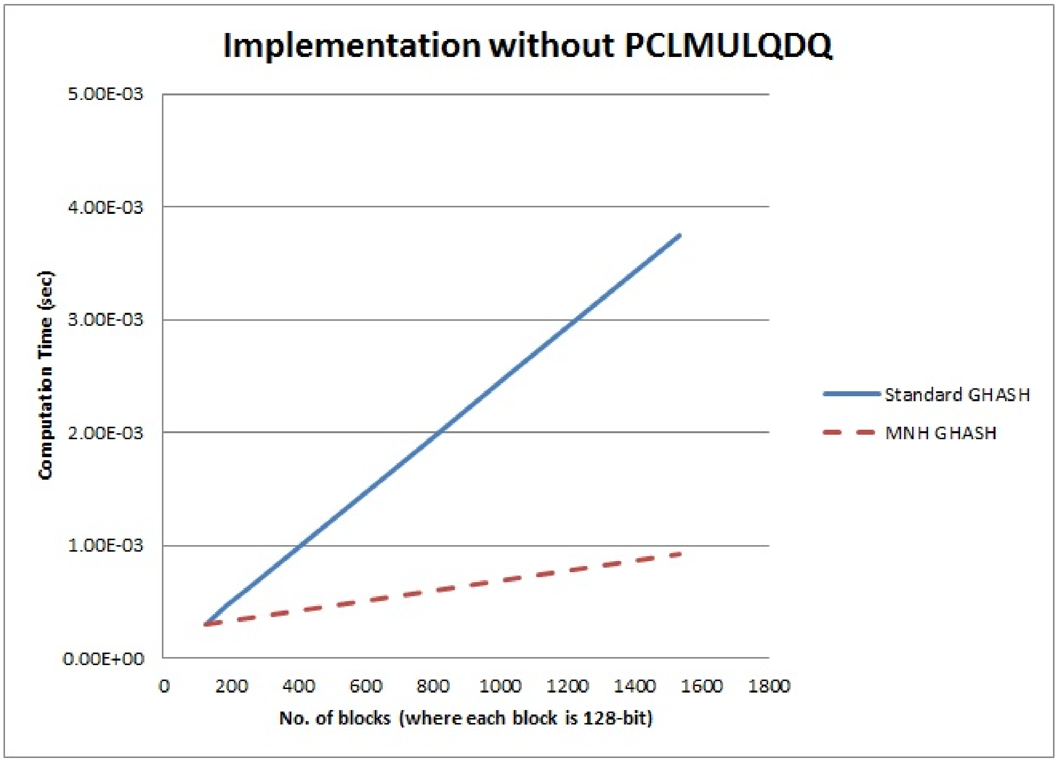

Figure 4.1: Performance Comparison of Implementations with Customary Instructions

tion of reduction algorithm discussed in Section 4.1.5 is used in combination to get 128-bit multiplication result. The characteristic polynomial for the MNH GHASH algorithm is obtained in a similar way as mentioned in Section 4.2.1. The results of computation time for the two algorithms can be seen in Table 4.3, and graphical representation of the results can be seen in Fig. 4.2.

As it can be seen from Fig. 4.2, the MNH GHASH algorithm does not perform well when compared to the standard one. The reason for better performance of the standard GHASH algorithm in this case is due to the usage of Intel's PCLMULDQ instruction, which really speeds up the carry-less multiplication operation. Another reason for the improved performance of the standard GHASH algorithm is that, the software implementation of the MNH GHASH algorithm is not able to utilize the parallelism required by the PRU operations, as we also mentioned it in section

No. of Blocks	Standard GHASH (x 10^{-4}sec)	MNH GHASH(x 10^{-4}sec)
128	0.095	0.103
192	0.137	0.378
256	0.181	0.660
384	0.267	1.216
512	0.357	1.766
768	0.531	2.891
1024	0.704	3.990
1280	0.880	5.107
1536	1.052	6.220

Table 4.3: Computation Time of Implementations with PCLMULQDQ

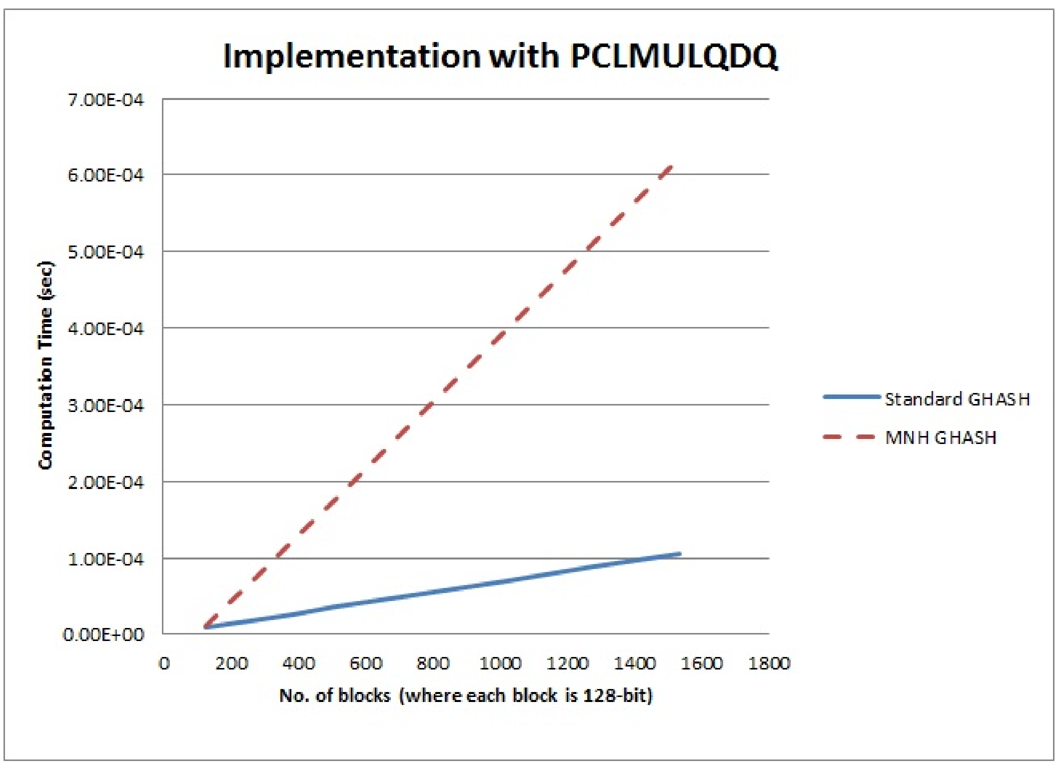

Figure 4.2: Performance Comparison of Implementations with PCLMULQDQ

4.2.1.

Chapter 5

Concluding Remarks

5.1 Summary

In this thesis, software implementations of the MNH GHASH are compared with those of the standard GHASH. In implementations, where Intel's PCLMULQDQ instruction is not used, the MNH GHASH has performed well as compared to the standard GHASH. In contrast, implementations where Intel's PCLMULQDQ instruction was used, the standard GHASH has proven to be better in performance than the MNH GHASH. In its core, the MNH GHASH algorithm attempts to reduce the number of multiplications required to compute the GHASH by using multiple XOR operations in parallel. Regardless of using or not using PCLMULQDQ, the implmentations have not been able to take advantage of parallelism present in the MNH GHASH algorithm. The parallelism can be utilized by having a hardware polynomial reduction unit, which today's main stream processors do not have. Even though parallelism is not utilized by our software implmentations due to the above-mentioned limitations, the MNH algorithm performs better than the standard implementation in case where PCLMULQDQ is not used. This suggests, that on architectures which are older than Westmere, or architectures which do not support Intel's PCLMULQDQ instruction, the MNH GHASH algorithm will perform better than the standard GHASH one.

5.2 Future Work

As we discussed, due to the inability of software implementations to exploit the parallelism of the MNH GHASH algorithm, the latter has not performed better than the standard GHASH algorithm. It is hard to compute 128 XOR operations, which are needed for the polynomial reduction unit, in exact parallel through software implementation. It is however possible to try on high end systems with multi-core and/or programmable logic equipped processors to at least do some part of computation in parallel. If, for example, we can perform four XOR operation in parallel, we can reduce 128 bit sequential XOR operations to 32 rounds of XOR operations, with each round having 4 XOR operations. Further work can be done on exploiting parallelism available on various high end systems to speed up software implementation of the MNH GHASH algorithm.

Appendix A

Software Implementation of Algorithms

A.1 Standard GHASH without PCLMULQDQ

```
// Uses implementation of multiplication from section 4.1.1
#include <stdint.h>
#include <inttypes.h>
#include <wmmintrin.h>
#include<emmintrin.h>
#include<smmintrin.h>
#include <stdio.h>
#include <time.h>
struct aes_block {    uint64_t a;    uint64_t b; };
void gfmulos(__m128i x, __m128i y, __m128i *res);
void print_m128i_with_string(char* string,__m128i data);
int main () {
unsigned long long a[1537]=
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
unsigned long long b[1537] =
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here                                  The parallelism can be uti
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
  __m128i X[1537] ;
  int i = 0;
  //Input Initialization
```

```
  for(i = 0 ; i <=1536 ;i++){
    a[i] = a[i] * 10000000000000000000ULL; // enlarging input
    b[i] = b[i] * 10000000000000000000ULL; // enlarging input
    X[i] = _mm_set_epi64((__m64)a[i],(__m64)b[i] );
  }
  __m128i H =  _mm_set_epi64((__m64)5708010131839353156ULL,
            (__m64)3405470159317640703ULL);   //Hash Sub-key
    /// Standard GHASH
  __m128i temp = {0x00, 0x00};
  int k = 0; // for multiple runs... to magnify timing results
 // The commented for loop is only used when running the code for
 // timing analysis
 // for(k =0 ; k<=10000; k++){
        for(i = 0 ; i <=1535 ;i++){
              temp =  _mm_xor_si128(temp, X[i]);
                gfmulos(temp,H,&temp);
      }
 // } //----------------
   print_m128i_with_string("TagResultf:", temp);
}
void print_m128i_with_string(char* string,__m128i data){
        unsigned char *pointer = (unsigned char*)&data;
        int i;
        printf("%-40s[0x",string);
        for (i=0; i<16; i++)
        printf("%02x",pointer[i]);
        printf("]\n");
}

void gfmulos(__m128i x, __m128i y,__m128i *res) {
   __m128i R = _mm_set_epi64((__m64)0ULL,(__m64)135ULL ) ;
   __m128i Z = _mm_set_epi64((__m64)0ULL,(__m64)0ULL );
   __m128i V = y;        __m128i X = x;
   int i = 0;
    __m64 ch;
    __m64 che;
    for(i=0; i < 128 ; i++ ){
        if(i<64){ ch = (__m64) X[0]; }
         else { ch = (__m64) X[1];}
        uint64_t ch1 = (uint64_t) ch&1ULL;
        if(ch1){
            Z = _mm_xor_si128(Z, V);          }
          che = (__m64) V[1];
         uint64_t che1 = (uint64_t) che&9223372036854775808ULL;
         if(che1) {
```

```
            __m64 ad = (__m64)V[0];
            uint64_t tx1 = (uint64_t) ad&9223372036854775808ULL;
            V =  _mm_slli_epi64 (V, 1); //
            if (tx1){
            V[1] = (uint64_t) _mm_or_si64((__m64)V[1],(__m64)1ULL);
            }
            V = _mm_xor_si128(V, R);
          } else {
             __m64 ad1 = (__m64)V[0];
           uint64_t tx2 = (uint64_t) ad1&9223372036854775808ULL;
            V = _mm_slli_epi64 (V, 1);//
               if (tx2){
              V[1] = (uint64_t) _mm_or_si64((__m64)V[1],(__m64)1ULL);
                   }
          }
         if(i<64){ X[0] = (uint64_t) _mm_srli_si64((__m64)X[0],1);}
         else { X[1] = (uint64_t)   _mm_srli_si64((__m64)X[1],1); }
   }
         *res = Z;
}
```

A.2 High Perfromance GHASH without PCLMULQDQ

```
// Uses modified implementation of multiplication from section 4.1.1,
// and array for 'ci' values (which is assumed as precomputed)
// is constructed from characteristic polynomial computed using
// Gordon's Algorithm Maple implementation described in Appendix A.5.
#include <stdint.h>
#include <inttypes.h>
#include <wmmintrin.h>
#include<emmintrin.h>
#include<smmintrin.h>
#include <stdio.h>
#include <time.h>
struct aes_block {     uint64_t a;     uint64_t b; };
void gfmulos(__m128i x, __m128i y, __m128i *res);
void print_m128i_with_string(char* string,__m128i data);
int main () {
unsigned long long a [1537]=
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
```

```
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
```

```
unsigned long long b[1537] =
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
  __m128i  X[1537] ;
  int i = 0;
  //Input Initialization
  for(i = 0 ; i <=1536 ;i++){
    a[i] = a[i] * 10000000000000000ULL; // enlarging input
    b[i] = b[i] * 10000000000000000ULL; // enlarging input
    X[i] = _mm_set_epi64((__m64)a[i],(__m64)b[i] );
  }
  __m128i  H =  _mm_set_epi64((__m64)5708010131839353156ULL,
             (__m64)3405470159317640703ULL);   //Hash Sub-key
    /// Standard GHASH
  __m128i temp = {0x00, 0x00};
  int k = 0; // for multiple runs... to magnify timing results
 // New GHASH  ---------------
  int j = 0;
  // Values of ci set based on characteristic polynomial
  // from Maple Code
     int ci[129] ={1,0,1,0,1,0,0,0,1,1,0,0,0,1,0,0,
                   0,0,0,0,1,1,1,1,1,1,1,1,0,1,1,1,
                   1,0,1,1,0,1,0,1,1,1,1,0,1,0,0,0,
                   0,0,1,1,0,1,1,1,1,1,1,1,0,0,0,1,
                   1,1,0,0,1,1,0,1,0,1,1,0,1,1,1,0,
                   1,1,1,0,1,0,0,0,1,1,0,1,1,0,0,0,
                   1,1,1,0,1,1,0,0,1,1,1,1,0,0,0,1,
                   0,0,1,0,0,0,0,1,0,0,1,0,1,1,0,1,1};
   __m128i Y[128];
 // The commented for loop is only used when running the code for
 // timing analysis
 //for(k = 0; k <=10000; k++){
   for(i=0; i<=127; i++){
      Y[i] = X[127-i];
    }
   for(j = 127 ; j<= 1535 ; j++){
     __m128i C = Y[127];
     for(i = 127; i >=1 ; i--){
     if(ci[i]==1){Y[i] = _mm_xor_si128(Y[i-1], C);}
     else {Y[i] = Y[i-1];}
     }
     if( ci[0] == 1){Y[0] = _mm_xor_si128(X[j+1], C);}
     else {Y[0] = X[j+1];}
```

```
    }
    for(i = 127 ; i >=1 ;i--){
      temp =  _mm_xor_si128(temp, Y[i]);
      gfmulos(temp,H, &temp);
    }
    temp = _mm_xor_si128 ( temp, Y[0]);
  //}
    print_m128i_with_string("TagResultf:", temp);
}
void print_m128i_with_string(char* string,__m128i data){
        unsigned char *pointer = (unsigned char*)&data;
        int i;
        printf("%-40s[0x",string);
        for (i=0; i<16; i++)
        printf("%02x",pointer[i]);
        printf("]\n");
}

void gfmulos(__m128i x, __m128i y,__m128i *res) {
    __m128i R = _mm_set_epi64((__m64)0ULL,(__m64)135ULL ) ;
    __m128i Z = _mm_set_epi64((__m64)0ULL,(__m64)0ULL );
    __m128i V = y;          __m128i X = x;
    int i = 0;
     __m64 ch;
     __m64 che;
     for(i=0; i < 128 ; i++ ){
        if(i<64){ ch = (__m64) X[0]; }
         else { ch = (__m64) X[1];}
        uint64_t ch1 = (uint64_t) ch&1ULL;
        if(ch1){
            Z = _mm_xor_si128(Z, V);            }
          che = (__m64) V[1];
         uint64_t che1 = (uint64_t) che&9223372036854775808ULL;
         if(che1) {
           __m64 ad = (__m64)V[0];
           uint64_t tx1 = (uint64_t) ad&9223372036854775808ULL;
           V =  _mm_slli_epi64 (V, 1); //
           if (tx1){
           V[1] = (uint64_t) _mm_or_si64((__m64)V[1],(__m64)1ULL);
           }
           V = _mm_xor_si128(V, R);
         } else {
            __m64 ad1 = (__m64)V[0];
          uint64_t tx2 = (uint64_t) ad1&9223372036854775808ULL;
           V = _mm_slli_epi64 (V, 1);//
```

```
            if (tx2){
           V[1] = (uint64_t) _mm_or_si64((__m64)V[1],(__m64)1ULL);
                }
        }
        if(i<64){ X[0] = (uint64_t) _mm_srli_si64((__m64)X[0],1);}
        else { X[1] = (uint64_t)   _mm_srli_si64((__m64)X[1],1); }
  }
        *res = Z;
}
```

A.3 Standard GHASH with PCLMULQDQ

```
// Uses multiplication implementation from Intel's PCLMULQDQ white paper
// , which is combination of algorithms in 4.1.4 and 4.1.5 Sections.
#include <stdint.h>
#include <inttypes.h>
#include <wmmintrin.h>
#include<emmintrin.h>
#include<smmintrin.h>
#include <stdio.h>
#include <time.h>
struct aes_block {     uint64_t a;     uint64_t b; };
void gfmulintel (__m128i a, __m128i b, __m128i *res);
void print_m128i_with_string(char* string,__m128i data);
int main () {
unsigned long long a [1537]=
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
```

```
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
unsigned long long b[1537] =
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
  __m128i X[1537] ;
  int i = 0;
  //Input Initialization
  for(i = 0 ; i <=1536 ;i++){
    a[i] = a[i] * 1000000000000000000ULL; // enlarging input
    b[i] = b[i] * 1000000000000000000ULL; // enlarging input
    X[i] = _mm_set_epi64((__m64)a[i],(__m64)b[i] );
  }
```

```
  __m128i  H =  _mm_set_epi64((__m64)5708010131839353156ULL,
            (__m64)3405470159317640703ULL);   //Hash Sub-key
    /// Standard GHASH
  __m128i temp = {0x00, 0x00};
  int k = 0; // for multiple runs... to magnify timing results
 // The commented for loop is only used when running the code for
 // timing analysis
 // for(k =0 ; k<=10000; k++){
        for(i = 0 ; i <=1535 ;i++){
              temp =  _mm_xor_si128(temp, X[i]);
                gfmulintel(temp,H,&temp);
      }
 // } //————————————
   print_m128i_with_string("TagResultf:", temp);
}
void print_m128i_with_string(char* string,__m128i data){
        unsigned char *pointer = (unsigned char*)&data;
        int i;
        printf("%-40s[0x",string);
        for (i=0; i<16; i++)
        printf("%02x",pointer[i]);
        printf("]\n");
}
void gfmulintel (__m128i a, __m128i b, __m128i *res){
   __m128i tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
   tmp7, tmp8, tmp9, tmp10, tmp11, tmp12;
   __m128i XMMMASK = _mm_setr_epi32(0xffffffff,0x0,0x0,0x0);
   tmp3 = _mm_clmulepi64_si128(a, b, 0x00);
   tmp6 = _mm_clmulepi64_si128(a, b, 0x11);
   tmp4 = _mm_shuffle_epi32(a,78);
   tmp5 = _mm_shuffle_epi32(b,78);
   tmp4 = _mm_xor_si128(tmp4, a);
   tmp5 = _mm_xor_si128(tmp5, b);
   tmp4 = _mm_clmulepi64_si128(tmp4, tmp5, 0x00);
   tmp4 = _mm_xor_si128(tmp4, tmp3);
   tmp4 = _mm_xor_si128(tmp4, tmp6);
   tmp5 = _mm_slli_si128(tmp4, 8);
   tmp4 = _mm_srli_si128(tmp4, 8);
   tmp3 = _mm_xor_si128(tmp3, tmp5);
   tmp6 = _mm_xor_si128(tmp6, tmp4);
   tmp7 = _mm_srli_epi32(tmp6, 31);
   tmp8 = _mm_srli_epi32(tmp6, 30);
   tmp9 = _mm_srli_epi32(tmp6, 25);
   tmp7 = _mm_xor_si128(tmp7, tmp8);
   tmp7 = _mm_xor_si128(tmp7, tmp9);
```

```
    tmp8 = _mm_shuffle_epi32(tmp7, 147);
    tmp7 = _mm_and_si128(XMMMASK, tmp8);
    tmp8 = _mm_andnot_si128(XMMMASK, tmp8);
    tmp3 = _mm_xor_si128(tmp3, tmp8);
    tmp6 = _mm_xor_si128(tmp6, tmp7);
    tmp10 = _mm_slli_epi32(tmp6, 1);
    tmp3 = _mm_xor_si128(tmp3, tmp10);
    tmp11 = _mm_slli_epi32(tmp6, 2);
    tmp3 = _mm_xor_si128(tmp3, tmp11);
    tmp12 = _mm_slli_epi32(tmp6, 7);
    tmp3 = _mm_xor_si128(tmp3, tmp12);
    *res = _mm_xor_si128(tmp3, tmp6);
}
```

A.4 High Perfromance GHASH with PCLMULQDQ

```
// Uses multiplication implementation as presented in
// Intel's PCLMULQDQ white paper, and which is combination
// of algorithms in section 4.1.4 and 4.1.5 , and also uses
// Gordon's algorithm in a similar way as in Appendix A.2
#include <stdint.h>
#include <inttypes.h>
#include <wmmintrin.h>
#include<emmintrin.h>
#include<smmintrin.h>
#include <stdio.h>
#include <time.h>
struct aes_block {     uint64_t a;     uint64_t b; };
void gfmulintel (__m128i a, __m128i b, __m128i *res);
void print_m128i_with_string(char* string,__m128i data);
int main () {
unsigned long long a [1537]=
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
unsigned long long b[1537] =
{1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,0ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
// 192 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 256 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,
// 384 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
//512 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 768 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
// 1024 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
```

```
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,
//1280 till here
0ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,5ULL,6ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,1ULL,2ULL,3ULL,4ULL,
1ULL,2ULL,5ULL,6ULL,0ULL};
  __m128i X[1537] ;
  int i = 0;
```

```
//Input Initialization
for(i = 0 ; i <=1536 ;i++){
  a[i] = a[i] * 10000000000000000000ULL; // enlarging input
  b[i] = b[i] * 10000000000000000000ULL; // enlarging input
  X[i] = _mm_set_epi64((__m64)a[i],(__m64)b[i] );
}
__m128i  H =  _mm_set_epi64((__m64)5708010131839353156ULL,
           (__m64)3405470159317640703ULL);   //Hash Sub-key
  /// Standard GHASH
__m128i temp = {0x00, 0x00};
int k = 0; // for multiple runs... to magnify timing results
// New GHASH ---------------
int j = 0;
// Values of ci set based on characteristic polynomial
// from Maple Code
   int ci[129] ={1,0,1,0,1,0,0,0,1,1,0,0,0,1,0,0,
               0,0,0,0,1,1,1,1,1,1,1,1,0,1,1,1,
               1,0,1,1,0,1,0,1,1,1,1,0,1,0,0,0,
               0,0,1,1,0,1,1,1,1,1,1,1,0,0,0,1,
               1,1,0,0,1,1,0,1,0,1,1,0,1,1,1,0,
               1,1,1,0,1,0,0,0,1,1,0,1,1,0,0,0,
               1,1,1,0,1,1,0,0,1,1,1,1,0,0,0,1,
               0,0,1,0,0,0,0,1,0,0,1,0,1,1,0,1,1};
 __m128i Y[128];
// The commented for loop is only used when running the code for
// timing analysis
//for(k = 0; k <=10000; k++){
 for(i=0; i<=127; i++){
   Y[i] = X[127-i];
  }
 for(j = 127 ; j<= 1535 ; j++){
   __m128i C = Y[127];
   for(i = 127; i >=1 ; i--){
   if(ci[i]==1){Y[i] = _mm_xor_si128(Y[i-1], C);}
   else {Y[i] = Y[i-1];}
   }
   if( ci[0] == 1){Y[0] = _mm_xor_si128(X[j+1], C);}
   else {Y[0] = X[j+1];}
 }
 for(i = 127 ; i >=1 ;i--){
   temp =  _mm_xor_si128(temp, Y[i]);
   gfmulintel(temp,H, &temp);
 }
 temp = _mm_xor_si128 ( temp, Y[0]);
//}
```

```
    print_m128i_with_string("TagResultf:", temp);
}
void print_m128i_with_string(char* string,__m128i data){
        unsigned char *pointer = (unsigned char*)&data;
        int i;
        printf("%-40s[0x",string);
        for (i=0; i<16; i++)
        printf("%02x",pointer[i]);
        printf("]\n");
}
void gfmulintel (__m128i a, __m128i b, __m128i *res){
    __m128i tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
    tmp7, tmp8, tmp9, tmp10, tmp11, tmp12;
    __m128i XMMMASK = _mm_setr_epi32(0xffffffff,0x0,0x0,0x0);
    tmp3 = _mm_clmulepi64_si128(a, b, 0x00);
    tmp6 = _mm_clmulepi64_si128(a, b, 0x11);
    tmp4 = _mm_shuffle_epi32(a,78);
    tmp5 = _mm_shuffle_epi32(b,78);
    tmp4 = _mm_xor_si128(tmp4, a);
    tmp5 = _mm_xor_si128(tmp5, b);
    tmp4 = _mm_clmulepi64_si128(tmp4, tmp5, 0x00);
    tmp4 = _mm_xor_si128(tmp4, tmp3);
    tmp4 = _mm_xor_si128(tmp4, tmp6);
    tmp5 = _mm_slli_si128(tmp4, 8);
    tmp4 = _mm_srli_si128(tmp4, 8);
    tmp3 = _mm_xor_si128(tmp3, tmp5);
    tmp6 = _mm_xor_si128(tmp6, tmp4);
    tmp7 = _mm_srli_epi32(tmp6, 31);
    tmp8 = _mm_srli_epi32(tmp6, 30);
    tmp9 = _mm_srli_epi32(tmp6, 25);
    tmp7 = _mm_xor_si128(tmp7, tmp8);
    tmp7 = _mm_xor_si128(tmp7, tmp9);
    tmp8 = _mm_shuffle_epi32(tmp7, 147);
    tmp7 = _mm_and_si128(XMMMASK, tmp8);
    tmp8 = _mm_andnot_si128(XMMMASK, tmp8);
    tmp3 = _mm_xor_si128(tmp3, tmp8);
    tmp6 = _mm_xor_si128(tmp6, tmp7);
    tmp10 = _mm_slli_epi32(tmp6, 1);
    tmp3 = _mm_xor_si128(tmp3, tmp10);
    tmp11 = _mm_slli_epi32(tmp6, 2);
    tmp3 = _mm_xor_si128(tmp3, tmp11);
    tmp12 = _mm_slli_epi32(tmp6, 7);
    tmp3 = _mm_xor_si128(tmp3, tmp12);
    *res = _mm_xor_si128(tmp3, tmp6);
}
```

A.5 Gordon's Algorithm in Maple

```
// Input is variable 'A' . For actual results ,
// a large value of input is used. x+1 here
// is just to give an example.
A := x+1
G := GF(2, 128, x^128+x^7+x^2+x+1);
aa := G:-ConvertIn(A);
T := x;
tt := G:-ConvertIn(T);
XA := G:-`+`(aa, tt);
Z := aa;
for i to 127 do
aa := G:-`*`(aa, aa);
XA := G:-`+`(G:-`*`(XA, tt), G:-`*`(XA, aa))
end do;
y := x^128+x^7+x^2+x+1;
XA := G:-ConvertOut(XA);
result := `mod`(y+XA, 2)
```

References

[1] 128 bit XOR intrinsic function usage. http://msdn.microsoft.com/en-us/library/fzt08www.aspx. Accessed: 16/08/2012. 45

[2] PCLMULQDQ intrinsic function usage. http://msdn.microsoft.com/en-us/library/cc664767.aspx. Accessed: 15/08/2012. 43

[3] Using AES-GCM IETF DRAFT. http://tools.ietf.org/html/draft-ietf-smime-cms-aes-ccm-and-gcm-00. Accessed: 16/08/2012. 1

[4] J. Bajard, L. Imbert, and G. Jullien. Parallel Montgomery multiplication in $GF(2^k)$ using trinomial residue arithmetic. In *17th IEEE Symposium on Computer Arithmetic (ARITH)*, 2005. 19

[5] Tianshan Chen, Wenjie Huo, and Zhenglin Liu. Design and Efficient FPGA Implementation of Ghash Core for AES-GCM. In *Computational Intelligence and Software Engineering (CiSE)*, pages 1–4. IEEE, 2010. 2, 3, 19

[6] Jean Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prashant Puniya. Merkle–damgard revisited: How to construct a hash function. pages 430–448. Springer–Verlag, 2005. 11

[7] Jeremie Crenne, Pascal Cotret, Guy Gogniat, Russell Tessier, and Jean-Philippe Diguet. Efficient key-dependent message authentication in reconfigurable hardware. In *FPT'11*, pages 1–6, 2011. 2

[8] Ashwini M. Deshpande, Mangesh S. Deshpande, and Devendra N. Kayatanavar. FPGA Implementation of AES Encryption and Decryption. In *International Conference on Control, Automation, Communicatin and Energy Conservation*, 2009.

[9] Mohamed Abo El-Fotouh and Klaus Diepold. Galois Substitution Counter Mode (GSCM). In *Enterprise Distributed Object Computing Conference Workshops, 12th*, pages 199–206. IEEE, 2008. 2, 19

[10] AJ Elbirt. Fast and Efficient Implementation of AES Via Instruction Set Extensions. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*. IEEE, 2007.

[11] Bulens et al. Implementation of the AES–128 on Virtex–5 FPGAs. In *Progress in Cryptology – AFRICACRYPT*, 2008. 19

[12] Chetna Sangwan et al. VLSI Implementation of Advanced Encryption Standard. In *2012 Second International Conference on Advanced Computing I& Communication Technologies*. IEEE, 2012.

[13] Vinodh Gopal et al. *Optimized Galois Counter Mode Implementation on Intel Architecture Processors*. White paper, Intel Corporation, 2010. 2, 3

[14] T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In *Cryptographic Hardware and Embedded Systems – CHES*, 2005. 19

[15] Jorge Guajardo, Sandeep S. Kumar, Christof Paar, and Jan Pelzl. Efficient Software–Implementation of Finite Fields with Applicaitons to Cryptography. In *Acta Appl Math*. Springer Science, 2006. 6

[16] Shay Gueron and Michael Kounavis. Efficient implementation of the galois counter mode using a carry–less multiplier and a fast reduction algorithm. pages 549–553. Elsevier North–Holland, Inc., 2010. 35, 43, 44, 45, 46

[17] Shay Gueron and Michael E. Kounavis. *Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode*. White paper, Intel Corporation, 2010. 2, 3, 11, 31, 37, 38, 42, 45, 46, 48

[18] Krzysztof Jankowski and Pierre Laurent. Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions. In *IEEE Transactions on Computers*. IEEE Computer Society, 2011. 2, 3

[19] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. In *Soviet Physics Doklady*, 1963. 33

[20] A. A. Karatsuba. The Complexity of Computations. In *Proceedings of the Steklov Institute of Mathematics*, 1995. 33

[21] Mehran Mozaffari Kermani and Arash Reyhani Masoleh. Efficient and High–Performance Parallel Hardware Architectures for the AES–GCM. In *IEEE Transactions on Computers*. IEEE, 2011. 19

[22] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *CRYPTO*, pages 95–107, 1994.

[23] Kuan Jen Lin, Chin-Mu Hsiao, and Ching Hung Jhan. Exploring HW/SW Codesign of AES Algorithm Using Custom Instructions. In *The 13th IEEE International Symposium on Consumer Electronics (ISCE2009)*. IEEE, 2009.

[24] Julio Lopez and Ricardo Dahab. High-speed software multiplication in F_{2^m}. In *INDOCRYPT'00*, pages 203–212, 2000.

[25] Yang Lu, Guochu Shou, Yihong Hu, and Zhigang Guo. The Research and Efficient FPGA Implementation of GHASH Core for GMAC. In *E–Business and Information System Security EBISS '09.* IEEE, 2009. 2

[26] Arash Reyhani Masoleh and M. Anwar Hasan. Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over $GF(2^m)$. In *IEEE Transactions on Computers.* IEEE, 2004. 2

[27] E. D. Mastrovito. Multiplication of Multidigit Numbers on Automata. In *PhD thesis, Dept. of Electrical Eng., Linkping Univ., Sweden*, 1991. 19

[28] D.A. McGrew and J. Viega. The Galois/Counter Mode of Operation (GCM). In *Submission to NIST Modes of Operation Process*, 2004. 15, 18, 19, 20, 23

[29] Nicolas Meloni, Christophe Negre, and M. Anwar Hasan. High performance GHASH and impacts of a class of unconventional bases. In *Journal of Cryptographic Engineering*, pages 201–218. Springer–Verlag, 2011. 2, 3, 4, 18, 21, 23, 25, 26, 37, 38, 40, 41

[30] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[31] Jorge Castineira Moreira and Patrick Guy Farrell. *Essentials of Error–Control Coding.* John Wiley and I& Sons Ltd., 2006. 6, 8

[32] NIST. Recommendation for Block Cipher Modes of Operation: Methods and Techniques . NIST Special Publication 800–38A, 2001. 14

[33] NIST. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800–38D, 2007. 11, 12, 15, 18, 19, 20, 21, 37, 38, 47

[34] Christof Paar and Jan Pelzl. *Understanding Cryptography A Textbook for Students and Practitioners.* Springer Science, 2010. 9, 10

[35] Akashi Satoh. High–Speed Parallel Hardware Architecture for Galois Counter Mode . In *IEEE International Symposium on Circuits and Systems, ISCAS 2007.* IEEE, 2007. 23