# An Architecture for the AES-GCM Security Standard

by

Sheng Wang

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, 2006

# Author's Declaration

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

The forth recommendation of symmetric block cipher mode of operation SP800-38D, *Galois/Counter Mode of Operation* (GCM) was developed by David A McGrew and John Viega. GCM uses an approved symmetric key block cipher with a block size of 128 bits and a universal hashing over a binary Galois field to provide confidentiality and authentication. It is built specifically to support very high data rates as it can take advantage of pipelining and parallel processing techniques.

Before GCM, SP800-38A only provided confidentiality and SP800-38B provided authentication. SP800-38C provided confidentiality using the counter mode and authentication. However the authentication technique in SP800-38C was not parallelizable and slowed down the throughput of the cipher. Hence, none of these three recommendations were suitable for high speed network and computer system applications.

With the advent of GCM, authenticated encryption at data rates of several Gbps is now practical, permitting high grade encryption and authentication on systems which previously could not be fully protected. However there have not yet been any published results on actual architectures for this standard based on FPGA technology.

This thesis presents a fully pipelined and parallelized hardware architecture for AES-GCM which is GCM running under symmetric block cipher AES on a FPGA multi-core platform corresponding to the IPsec ESP data flow.

The results from this thesis show that the round transformations of confidentiality and hash operations of authentication in AES-GCM can cooperate very efficiently within this pipelined architecture. Furthermore, this AES-GCM hardware architecture never unnecessarily stalls data pipelines. For the first time this thesis provides a complete FPGA-based high speed architecture for the AES-GCM standard, suitable for high speed embedded applications.

# Acknowledgements

I would like to thank my supervisor, Professor Cathy Gebotys, for all her advices, guidance and encouragement. I would also like to thank Ann Lee, Suze Yang, my parents and brothers for their love and support.

I greatly appreciate the scholarships awarded to me by *the Natural Sciences and Engineering Research Council of Canada (NSERC), the University of Waterloo,* and *the Department of Electrical and Computer Engineering at the University of Waterloo.*

I am also grateful for the perfect campus environment provided by *the University of Waterloo*. On the campus, I enjoy not only the various academic activities, but also the diverse sports, such as swimming, golf, squash, volleyball, aerobics, curling and juggling.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# Introduction

## 1.1 Motivation

Since 2001, the National Institute of Standards and Technology (NIST) has published a total of four recommendations for Block Cipher Modes of Operation, specifically SP800-38A[1], SP800-38B[21], SP800-38C[22], and SP800-38D[2]. A block cipher mode of operation is an algorithm that uses a symmetric key block cipher to provide confidentiality, authentication or both for information security.

In SP800-38A, NIST recommends five confidentiality modes of operation for use with an underlying symmetric key block cipher algorithm: Electronic Codebook (ECB) mode, Cipher Block Chaining (CBC) mode, Cipher Feedback (CFB) mode, Output Feedback (OFB) mode, and Counter (CTR) mode. These five modes can be separated into two groups: one is a non-feedback mode group, including ECB and CTR; one is a feedback mode group, including CBC, CFB and OFB. In the feedback modes, the current computation/execution step depends on the result of the previous step. Therefore, to implement these kinds of modes in hardware, an iterative architecture is typically adapted for low throughput requirements rather than a pipelined architecture. In contrast, the use of ECB or CTR mode, or non-feedback modes, supports pipelined or parallelized architecture designs for processing high-speed data flows.

In SP800-38B, NIST recommends a message authentication code (MAC) algorithm that is based on a symmetric key block cipher. This cipher-based MAC is referred to as CMAC, or the Cipher Block Chaining MAC algorithm (CBC-MAC).

In SP800-38C, NIST recommends a mode of operation, called CCM, based on an approved symmetric key block cipher algorithm whose block size is 128 bits. CCM provides the confidentiality and the authenticity of data by combining the techniques of the CTR mode and the CBC-MAC.

The throughput of the CTR mode implementation, which can be pipelined, is much larger than that of CBC-MAC implementation, which can not be parallelizable. Specifically, none of these three recommendations, SP800-38A, B, and C, can be adapted for high speed network and computer system applications. Therefore, there is a compelling need for a mode of operation which can not only efficiently provide confidentiality, but authentication at high speed.

As the forth security standard of Block Cipher Mode of Operation, SP800-38D, *Galois/Counter Mode of Operation (GCM)*, fills the need above. GCM features the use of an approved symmetric key block cipher with a block size of 128 bits and a universal hash function that is defined over a binary Galois field. The most recently approved symmetric key block cipher with a block size of 128 bits is the Advanced Encryption Standard (AES) algorithm that is specified in Federal Information Processing Standard (FIPS) Pub.197 [3]. The specified universal hash function in GCM is defined over a binary Galois field (GF) and is a 128-bit polynomial multiplier over GF ($2^{128}$), called GHASH. GHASH can provide a secure, parallelizable, and efficient authentication mechanism. For the confidentiality mechanism of GCM, the CTR mode embedded by ECB mode, called GCTR, is adopted using an underlying block cipher. GCM, i.e. SP800-38D, was officially published in April 2006. However there are no known FPGA (field programmable gate array) architectures or implementations of this standard.

## 1.2 Work Objective

The motivation behind this thesis is to demonstrate and analyze GCM's practical performance and area cost by implementing it on a realistic hardware platform. In the rest of this thesis, AES-GCM will refer to GCM with AES as the symmetric block cipher.

Presented in this thesis are the hardware architectures for AES-GCM, including iterative-AES module, pipelined-AES module, GHASH module, and Key-expanded module. These GCM elements are then integrated together along with control logic to implement the entire new security standard, AES-GCM.

To verify the feasibility, efficiency and cost of each hardware module in AES-GCM, AES-GCM has been demonstrated using an advanced multi-core FPGA. The architectural design was synthesized, timing simulated, and downloaded to the FPGA prototype platform.

## 1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 gives an overview of the mathematical definitions over GF and confidentiality mode of operation concepts used in AES-GCM. This chapter also provides an introduction to the FPGA device structures and the FPGA prototype platform system used to create AES-GCM. Chapter 3 presents security standards, AES, and GCM. In chapter 4, the proposed hardware architectures of AES-GCM are presented. The bit parallel multiplier over GF, and the pipelined AES discussed in chapter 2 and 3 are chosen as the modules to build AES-GCM. A methodology, discussed in chapter 4, to verify the AES-GCM hardware implementation is established relying on the prototype platform. Finally, chapter 5 provides summary and future work.

# Chapter 2
# Background

This chapter provides the concepts necessary in order to understand GCM: Section 2.1 introduces to the concepts of finite fields, and the polynomial multiplier over binary finite fields followed by a discussion of its different implementation architectures. Section 2.2 is a description of ECB and CTR modes of operation which are the core of confidentiality of AES-GCM. Section 2.3 is an overview of FPGA technology and the multi-core FPGA prototype platform used to implement AES-GCM in this thesis.

## 2.1 Mathematical Background

The fundamentals of AES and GHASH are based on operations over the finite field. This section provides an introduction to these operations. The concepts and methods have been gathered from [4], [5], and [6].

### 2.1.1 Finite Fields

A field can be considered as a set whose elements form a group G under two operations: multiplication indicated by symbol "·" and addition indicated by symbol "+". These operations obey the basic algebraic properties. The relative finite field concepts are list as follows:

**Concept 1**. (F, + , ·) is a field if the following properties hold:

- The elements of F form a group under addition.
- The non-zero elements of F form a group under multiplication.
- The addition and multiplication operations are commutative, i.e.x + y = y + x and xy = yx for all x, y $\in$ F.

- The multiplication operation can be distributed through the addition operation, i.e. $x\cdot (y + x) = x\cdot y + x\cdot z$ for all x, y, and z $\in$ F.

**Concept 2**. A field F with a finite number of elements is a finite field.

**Concept 3**. A non-zero element of a finite field F is said to be a primitive element or generator of F if its powers cover all nonzero field elements.

**Concept 4**. A unique finite field exists for every prime number. These fields are denoted $GF(p^m)$ where p is prime and m is a positive integer. One kind of field which is commonly used in cryptography applications is the binary finite fields $GF(2^m)$ where m is a large integer.

**Concept 5**. A basis for GF $(2^m)$ over GF(2) is a set of m linearly independent elements of $GF(2^m)$. Any element of $GF(2^m)$ can be represented as an algebraic sum of the basis elements.

The binary field $GF(2^m)$ contains $2^m$ elements. Each element is represented by the selected basis. The most common representation is based on polynomial basis. With the polynomial basis $\alpha = \{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, the elements of GF $(2^m)$ can be represented as polynomial of degree m-1 as follows:

$$GF(2^m) =\{A|A= a_0 + a_1\alpha +\cdot\cdot\cdot + a_{m-1}\alpha^{m-1}, \text{where } a_j \in GF(2), 0 \leq j \leq m-1\}$$

where $\alpha$ is the root of any irreducible polynomial F(x) of degree m over GF(2).

Let

$$F(x) = 1 + f_1x + f_2x^2 + \cdot\cdot\cdot + f_{m-1}x^{m-1} + x^m$$

where $f_i \in GF(2)$, $0 \leq i \leq m-1$. The irreducible polynomial F(x) is often referred to as the field polynomial. The arithmetic in AES-GCM is based on polynomial basis and uses the polynomial $F(x) = 1 +x + x^2 + x^7 + x^{128}$ as field polynomial.

### 2.1.2 Operations over Binary Finite Fields GF ($2^m$)

Both operations, field addition and field multiplication, map a pair of field elements A and B onto another field element C, all A, B, and C $\in$GF($2^m$). The following introduction on field addition and multiplication is based on polynomial basis. The field elements A, B, and C are the following polynomials, respectively:

$A(\alpha) = a_0+a_1\,\alpha +\cdot\cdot\cdot+a_{m-1}\,\alpha^{m-1}$

$B(\alpha) = b_0+b_1\,\alpha +\cdot\cdot\cdot+b_{m-1}\,\alpha^{m-1}$

$C(\alpha) = c_0+c_1\,\alpha +\cdot\cdot\cdot+c_{m-1}\,\alpha^{m-1}$.

#### 2.1.2.1 Field Addition

Over a finite field GF($2^m$), a field addition of two elements A and B consists of adding the two polynomials together. Because the coefficients in A and B are over GF(2) and each pair of coefficients are added independently, their sum C is written as

$$\mathrm{C}(\alpha) = \mathrm{A}(\alpha) + \mathrm{B}(\alpha) = \sum_{\mathrm{i=0}}^{\mathrm{m-1}} (\mathrm{a_i} + \mathrm{b_i})\alpha_{\mathrm{i}}. \qquad (1)$$

The pair of coefficients addition $a_i + b_i$ in (1) is performed modulo 2 and translated to an exclusive- OR (XOR) operation in FPGA technology. That is to say that the field addition in (1) is computed by an m-bit XOR operation and does not require a carry chain.

#### 2.1.2.2 Field Multiplication

Field multiplication over a finite field GF($2^m$) is executed by straightforward multiplying two polynomials A($\alpha$) and B($\alpha$), then dividing the resulting 2m-bit polynomial by F($\alpha$); the m-bit remainder is the result C($\alpha$). The product C of field elements *A* and *B* is expressed as

$$C(\alpha) = A(\alpha) \times B(\alpha) \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i\alpha_i = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} a_i b_j \alpha^{i+j} \bmod F(\alpha) \qquad (2)$$

Commonly, people implement field multiplication in three methods with various efficiencies: Bit Serial multiplier, Bit Parallel multiplier and Digital Serial multiplier. In the following sections, the corresponding Algorithms to these three multipliers are presented.

### 2.1.2.2.1 Bit Serial Multiplier

By expanding B($\alpha$) and distributing A($\alpha$) to B($\alpha$)'s terms in (2), C($\alpha$) changes to

$$C(\alpha) = b_{m-1}\alpha^{m-1}A(\alpha) + \cdots + b_1\alpha A(\alpha) + b_0A(\alpha) \bmod F(\alpha).$$

By repeatedly grouping and factoring out $\alpha$, C($\alpha$) changes further to

$$C(\alpha) = (\cdots(((A(\alpha)b_{m-1})\alpha + A(\alpha)b_{m-2})\alpha + \cdots + A(\alpha)b_1)\alpha + A(\alpha)b_0) \bmod F(\alpha) \quad (3)$$

Starting with the inner most parentheses and moving out, Algorithm 1 performs the operation iteratively to compute the right hand side of (3). This algorithm can be used to compute the product of A($\alpha$) and B($\alpha$) in bit-level.

**Algorithm 1** Bit-Serial Multiplication

```
Input: $A(\alpha)$, $B(\alpha)$, and $F(\alpha)$
Output: $C(\alpha) = A(\alpha) \times B(\alpha) \bmod F(\alpha)$
  1. $C(\alpha) \leftarrow 0$
  2. for $i = m - 1$ downto 0 do
  3. $C(\alpha) \leftarrow \alpha C(\alpha) \bmod F(\alpha)$
  4. if $(b_i = 1)$ then
  5. $C(\alpha) \leftarrow C(\alpha) + A(\alpha)$.
```

The Bit Serial multiplier is a direct implementation of Algorithm 1. Totally 128 clock cycles are needed for calculating a multiplication over GF($2^{128}$) if $A(\alpha)$ can be loaded in parallel. The Figure 1 is the hardware structure of the bit serial multiplier.

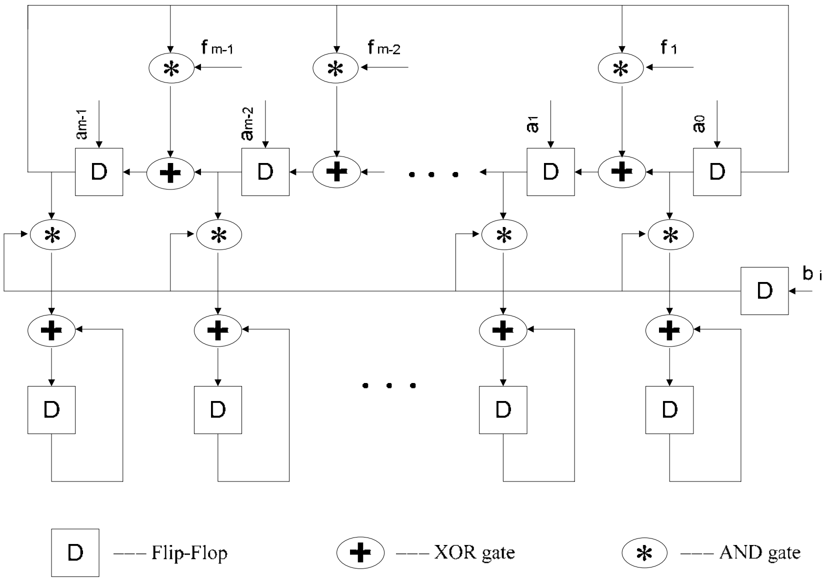


**Figure 1. Bit Serial Multiplier over GF($2^m$)**

### 2.1.2.2.2 Bit Parallel Multiplier

Compared to the bit serial multiplier which needs m clock cycles to complete a multiplication over GF($2^m$), a bit parallel multiplier can complete computation in only 1 clock cycle over the same GF. (Because the circuit delays are very different between the bit serial multiplier and the bit parallel multiplier, the minimum clock period of clock for parallel multiplier is much larger than the minimum one for serial multiplier. i.e., 1 clock cycle computation time for parallel multiplier should be roughly equal to several or tens clock cycles computation time for serial multiplier.) A dedicated polynomial basis finite field bit parallel multiplier has been proposed in [6], called the Mastrovito multiplier. This multiplier is adapted to a fixed field polynomial F(α) . The implementation procedure of the Mastrovito multiplier is described as follows.

First, rewrite polynomial elements A, B, and C in coefficient vector format as

$A= (a_0, a_1, \cdots, a_{m-1}),$

$B= (b_0, b_1, \cdots, b_{m-1}),$

and C= $(c_0, c_1, \cdot\cdot\cdot, c_{m-1})$.

Similarly, let $\alpha^i A=((\alpha^i A)_0, (\alpha^i A)_1, \ldots, (\alpha^i A)_{m-1})$.

Then, rewrite the field multiplication equation (2) in matrix-vector form multiplication as follows:

$$C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha_i = A(\alpha) \times B(\alpha) = A(b_0 + b_1 \alpha + \cdot\cdot\cdot + b_{m-1} \alpha^{m-1})$$

$$= (A, A\alpha, A\alpha^2, \ldots, A\alpha^{m-1}) \cdot B^T \qquad (4)$$

Equate the coefficients of $\alpha^i$, i=0, 1, …, m-1, on both sides of equation (4),

$c_0 = ((A)_0, (\alpha A)_0, \ldots, (\alpha^{m-1} A)_0) \cdot B^T$

$c_1 = ((A)_1, (\alpha A)_1, \ldots, (\alpha^{m-1} A)_1) \cdot B^T$

.

.

.

$c_{m-1} = ((A)_{m-1}, (\alpha A)_{m-1}, \ldots, (\alpha^{m-1} A)_{m-1}) \cdot B^T$

Thus,

$$C^T = ((A)^T, (\alpha A)^T, \ldots, (\alpha^{m-1} A)^T) \cdot B^T \qquad (5)$$

Let **M** denote the multiplication matrix $((A)^T, (\alpha A)^T, \ldots, (\alpha^{m-1} A)^T)$, then (5) can be rewritten like below for short,

$$C^T = M \cdot B^T \qquad (6)$$

Therefore, M is determinated by A and polynomial basis $\alpha = \{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$. Given $\alpha^i A$, the coefficient vector of $\alpha^{i+1} A$ mod F(α) can be computed as follows, $\alpha^{i+1} A = \alpha \cdot \alpha^i A$ mod F(α). Obviously, the cost of generating $\alpha^{i+1} A$ is the Hamming weight of F(α) – 2 XOR gates.

The bit parallel serial multiplier is a direct implementation based on (5). Architecture of the Mastrovito multiplier is shown in Figure 2, which comprises of two parts, one is named f-network, and one is named IP-network. The f-network generates the multiplication matrix **M**, which contains XOR gates only. The IP-network performs the matrix-vector multiplication as shown in (6) and consists of m inner product modules, each of these product modules consists of m AND gates and m-1 XOR gates.

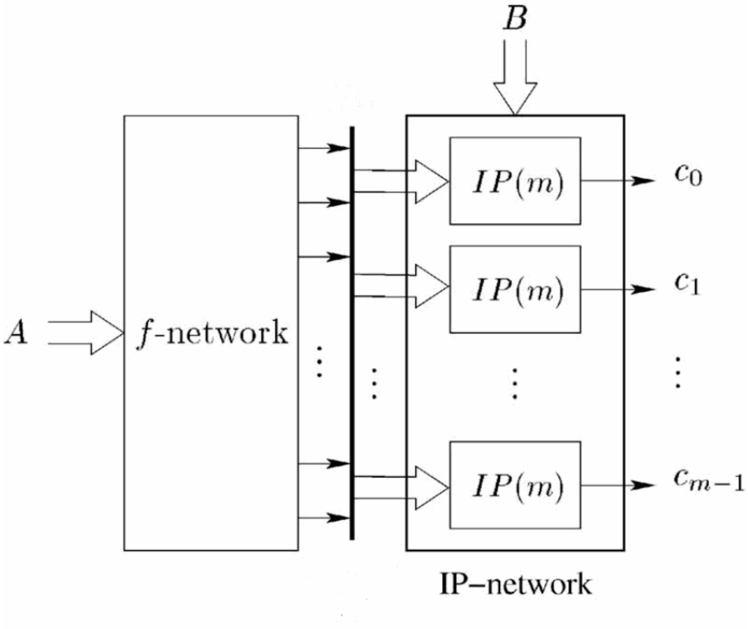


**Figure 2. Architecture of Bit Parallel Multiplier over GF($2^m$).[17]**

The bit parallel multiplier essentially unrolls the 128 loops of the bit serial multiplier; therefore, the area cost is 128 times of bit serial one.

### 2.1.2.2.3 Digit Serial Multiplier

A digit serial multiplier is a compromise between the serial and parallel methods and trades off circuit area against computation time.

Let g be an integer less than m and let s = $\lceil m/g \rceil$. Let B(α) to be separated into s pieces, each piece is organized in $\alpha^{ig}B_i(\alpha)$, where i and $B_i(\alpha)$ are defined like below,

$$B_i(\alpha) = \sum_{j=0}^{g-1} b_{ig+j} x^j \quad \text{for } 0 \leq i \leq s\text{-}2$$

$$B_i(\alpha) = \sum_{j=0}^{(m \bmod g)-1} b_{ig+j} x^j \quad \text{for } i = s\text{-}1$$

Then the (3) can be rewritten

$$C(\alpha) = A(\alpha)\,(\alpha^{(s-1)g} B_{s-1}(\alpha) + \cdots + \alpha^g B_1(\alpha) + B_0(\alpha)) \bmod F(\alpha)$$

$$=(\cdots((A(\alpha)B_{s-1}(\alpha))\alpha^g + A(\alpha)B_{s-2}(\alpha))\alpha^g + \cdots)\alpha^g + A(\alpha)B_0(\alpha) \bmod F(\alpha) \qquad (7)$$

Equation (7) can be computed using Algorithm 2.

**Algorithm 2** Digital-Serial Multiplication

Input: $A(\alpha)$, $B(\alpha)$, and $F(\alpha)$

Output: $C(\alpha) = A(\alpha)B(\alpha) \bmod F(\alpha)$

```
$C(\alpha) \leftarrow B_{s-1}(\alpha)A(\alpha) \bmod F(\alpha)$;
for $k = s - 2$ downto 0 do
    $C(\alpha) \leftarrow \alpha^g C(\alpha)$;
    $C(\alpha) \leftarrow B_k(\alpha)A(\alpha) + C(\alpha) \bmod F(\alpha)$;
```

The digital serial multiplier is implemented based on Algorithm 2. The multiplication is completed in $\lceil m/g \rceil$ iterations which is g times faster than the bit serial multiplier, and requires fewer resources than the bit parallel multiplier.

For more detailed description with respect to digital serial multiplier, please refer to [7] in which a digit serial multiplier is proposed based on look-up tables.

## 2.2 Confidentiality Mode of Operation Background

Two modes of operation for Symmetric Key Block Ciphers, ECB and CTR, are selected to create the confidentiality in AES-GCM because they can admit pipelined, parallelized implementations and have minimal computational latency for high data rates. These modes are introduced below and more details can be obtained from [1].

### 2.2.1 Electronic Codebook Mode (ECB)

The ECB mode is defined as follows and shown in Figure 3:

ECB Encryption: $C_j = CIPH_K(P_j)$ for $j = 1 \dots n$,

ECB Decryption: $P_j = CIPH^{-1}_K(C_j)$ for $j = 1 \dots n$,

where, $CIPH_K(P_j)$ is the forward cipher function of the block cipher algorithm, such as AES, under the key $K$ applied to the plaintext $P_j$; $CIPH^{-1}_K(C_j)$ is the inverse cipher function of the block cipher algorithm under the key $K$ applied to the ciphertext $C_j$.

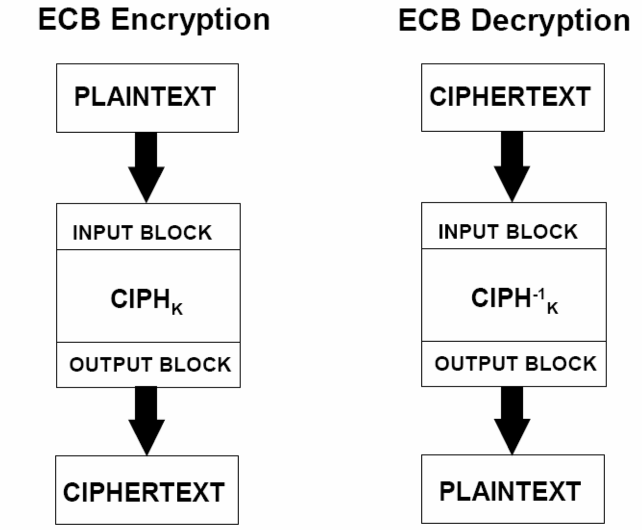


**Figure 3. ECB Encryption and ECB Decryption [1]**

In ECB encryption and ECB decryption, multiple forward cipher functions and inverse cipher functions can be computed in parallel or pipeline. In the GCTR module of AES-GCM, ECB encryption block is embedded into a CTR block (see Figure 21).

### 2.2.2 Counter Mode (CTR)

The CTR mode is a confidentiality mode also that features the application of the block cipher to a set of input data groups, called counters, to produce a set of keystreams that are XORed with the plaintext to produce the ciphertext, and vice versa. The CTR mode is defined as follows and shown in Figure 4.

CTR Encryption:

$O_j = CIPH_K(T_j)$ for j = 1, 2 … n;

$C_j = P_j$ XOR $O_j$ for j = 1, 2 … n-1;

$C^*_n = P^*_n$ XOṚ $MSB_u(O_n)$.

CTR Decryption:

$O_j = CIPH_K(T_j)$ for j = 1, 2 … n;

$P_j = C_j$ XOṚ $O_j$ for j = 1, 2 … n-1;

$P^*_n = C^*_n$ XOṚ $MSB_u(O_n)$.

The symbols used in the CTR encryption and decryption are:

$T_j$----the counters for the jth input data group,

$O_j$---- the key stream for the jth input data group,

$P_j$ ---- the jth plaintext group,

$C_j$---- the jth ciphertext group.

$C^*_n$ ----the last group of the ciphertext, which may be a partial group.

$P^*_n$ ----the last group of the plaintext, which may be a partial group.

$MSB_u(O_n)$---- the bit string consisting of the u most significant bits of the bit string $O_n$.

In CTR encryption and CTR decryption, only the forward cipher function is invoked on each counter group, no inverse cipher function. The resulting key streams are XORed with the corresponding plaintext or ciphertext blocks to produce the ciphertext or plaintext blocks. For the last group, which may be a partial group of u bits, the most significant u bits of the last output group are used for the XOR operation; the remaining bits of the last output group are discarded. The forward cipher functions can be performed in parallel and pipelined.

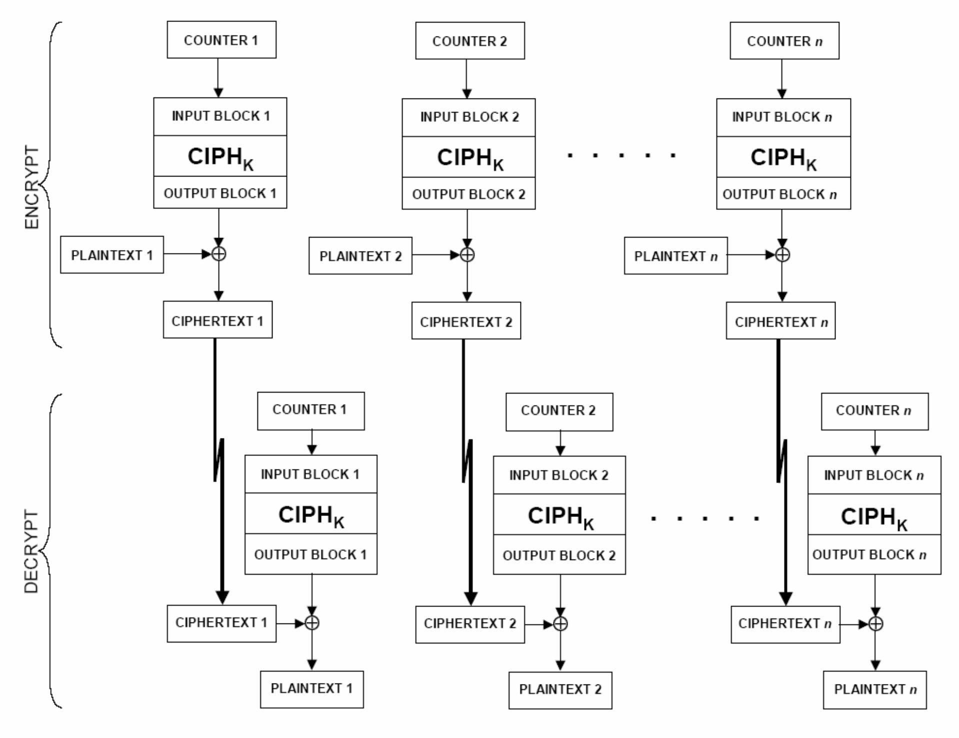


**Figure 4. CTR Encryption and CTR Decryption [1]**

Both CTR encryption and CTR decryption are invoked in AES-GCM encryption and AES-GCM decryption, respectively.

## 2.3 Field Programmable Gate Arrays (FPGA)

The thesis presents the architecture of FPGA implementation of AES-GCM. The common implementation approaches are corresponding to three different technologies. They are:

- Application Specific Integrated Circuits (ASICs)
- Software-Programmed General Purpose CPU (SPGPC)
- Field Programmable Gate Arrays (FPGAs)

**ASIC**s are specifically designed for a fixed solution, and are thus very efficient. However, the circuit cannot be changed after fabrication. This requires a redesign of the chip if any modification needs to be done.

**SPGPC**s are a flexible solution. CPUs execute a set of instructions to perform an algorithm. By changing the software code, the functionality of the system is altered without touching the hardware. But the SPCGPC's efficiency is much lower than that of an ASIC.

**FPGA**s offer a compromise between the ASIC and the SPGPC, achieving higher performance than software, while maintaining a higher level of flexibility than hardware.

### 2.3.1 Advantages of FPGA in Cryptographic Applications

The following attributes of the FPGA technology are particularly advantageous for cryptographic applications [8].

**Algorithm Agility**: More and more security applications intend to be algorithm independent and allow switching encryption algorithms on the flying. The encryption algorithm can be chosen through the negotiation made by two communication parties.

**Algorithm Upload**: From a cryptographic point of view, algorithm upload can be necessary because a current algorithm is out of date or broken; a new algorithm is created. The security designer of the corresponding security company can upload the new bit streams of security standard to reconfigure FPGA device through the networks.

**Throughput**: Although FPGA implementations are typically slower than ASIC implementations, FPGA implementations are obviously faster than software implementations. In a cryptosystem, if a software solution is chosen for clients, then, a FPGA implementation should be adapted for servers in high-speed backbones.

**Cost Efficiency**: The production costs of an ASIC are often too high for a small number of servers in security systems. Thus, the use of FPGAs is a common alternative. Furthermore, this is the one of reasons why the FPGA is chosen for security research in institutes and universities.

Therefore, it is often best to choose an FPGA to implement cipher, such as AES-GCM standard. The CMC-FPGA-prototype-platform was chosen in this thesis for prototyping since it represents a generalized multi-core platform, appropriate for security applications. This FPGA platform will be discussed next.

### 2.3.2 FPGA VirtexII pro xc2vp100

A traditional FPGA is usually an integrated circuit consisting of

- Configurable Logic Blocks (CLBs),
- Input/Output Blocks (IOBs) and
- Programmable routing resources.

For Xilinx FPGA family VirtexII pro contains not only CLBs, IOBs, and memory arrays, but also it has IBM 400 MHz PowerPC™ processors and 622 Mbps to 6.25 Gbps full duplex serial MGT(multi-gigabit transceivers). More specifically, Table 1 shows all the main resources of the VirtexII pro xc2vp100 targeted in this thesis.

**Table 1. Virtex-II Pro Resources**

| Device | PowerPC405 Processor Blocks | CLB(4 slices) | | 18x18 Bit Multiplier Blocks | Block SelectRAM+ | | DCM | Max User I/O Pads |
|---|---|---|---|---|---|---|---|---|
| | | Slices | Max Distr. RAM (Kb) | | 18 Kb Blocks | Max Block RAM (Kb) | | |
| xc2vp100 | 2 | 44096 | 1378 | 444 | 444 | 7992 | 12 | 1164 |

### 2.3.2.1 Configurable Logic Blocks (CLBs)

The CLBs in the VirtexII pro are comprised of both combinational and sequential logic. The combinational logic can be configured to become possible Boolean functions. Flip-Flops are provided to support sequential logic and can be utilized or bypassed depending on the configuration.

One CLB has four slices. Each slice is identical and contains:

- Two function generators F and G
- Two storage elements
- Arithmetic logic gates
- multiplexers
- Fast carry look-ahead chain

- Horizontal cascade chain

A general slice structure of VirtexII pro is shown in Figure 5. The function generators F and G can be configured as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM+ memory (to be discussed in section 2.3.2.2). The multiplexers, MUXF5 and MUXFX can provide any function of five, six, seven, or eight inputs when combined with function generators. The two storage elements can be configured either edge-triggered flip-flops or level-sensitive latches. Each CLB has internal fast interconnect and connects to a switch matrix to access general routing resources.

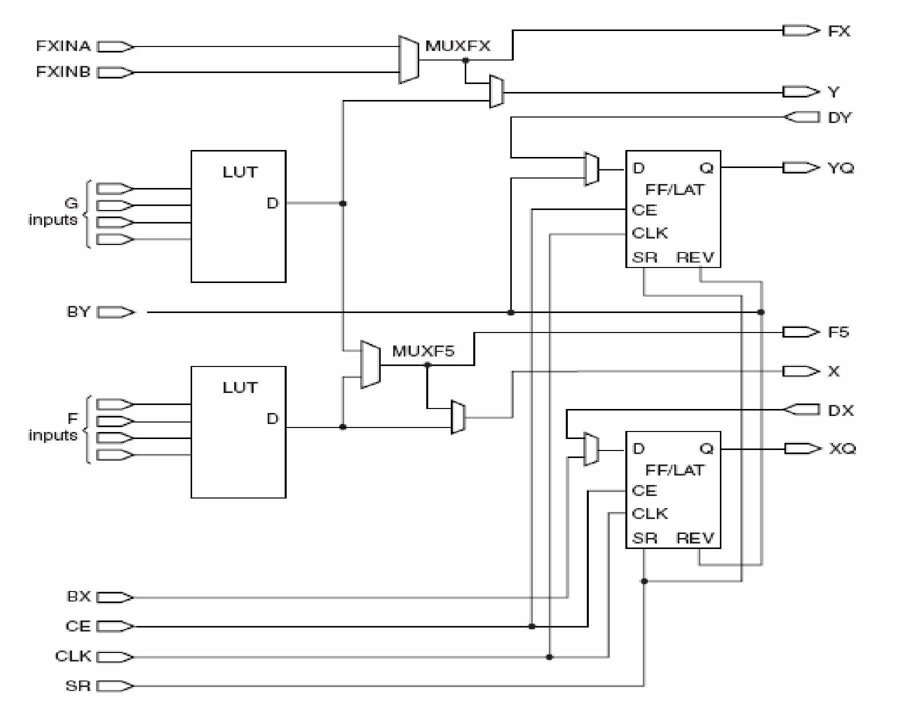


**Figure 5. General Slice in Virtex-ll Pro [15]**

### 2.3.2.2 Distributed SelectRAM+

As described in section 2.3.2.1, the Virtex-II Pro family devices feature distributed SelectRAM modules which are implemented by using function generators of the CLB resource. Distributed SelectRAM memory writes synchronously and reads asynchronously.

However, a synchronous read can be implemented using the register that is available in the same slice. One LUT can be configured as a 16x1-bit RAM which is cascadable for a deeper and wider memory implementation.

A primitive is a component in Xilinx software's library, which is available for instantiation without the need for specifying a definition in HDL code. One CLB which include 8 LUTs can be used to form up to a 128x1-bit RAM module primitive (see Table 2). These primitives are actually Distributed SelectRAM modules. Two 16 x 1 RAM resources can be combined to form a dual-port 16 x 1 RAM with one dedicated read/write port and a second read-only port. One port writes into both 16 x1 RAMs simultaneously, but the second port reads independently.

Table 2 shows the number of LUTs used in each distributed SelectRAM+ primitives.

**Table 2. Resources Used by Distributed Memory [15]**

| Primitive name | RAM Size | Type | # of LUTs |
|---|---|---|---|
| 16x1S | 16x1 bits | single port | 1 |
| 16x1D | 16x1 bits | dual port | 2 |
| 32x1S | 32x1 bits | single port | 2 |
| 32x1D | 32x1 bits | dual port | 4 |
| 64x1S | 64x1 bits | single port | 4 |
| 64x1D | 64x1 bits | dual port | 8 |
| 128x1S | 128x1 bits | single port | 8 |

For Single-port Distributed SelectRAM+ (see Figure 6), synchronous writes address lines WG[4:1] and asynchronous reads lines A[4:1] use the same shared address lines A[3:0] from the outside.

For Dual-port Distributed SelectRAM+ (see Figure 7), besides one LUT used in single-port mode, the second LUT uses the same address lines A[3:0] for synchronous write and another set of address lines DPRA[3:0] are used for the second asynchronous read.

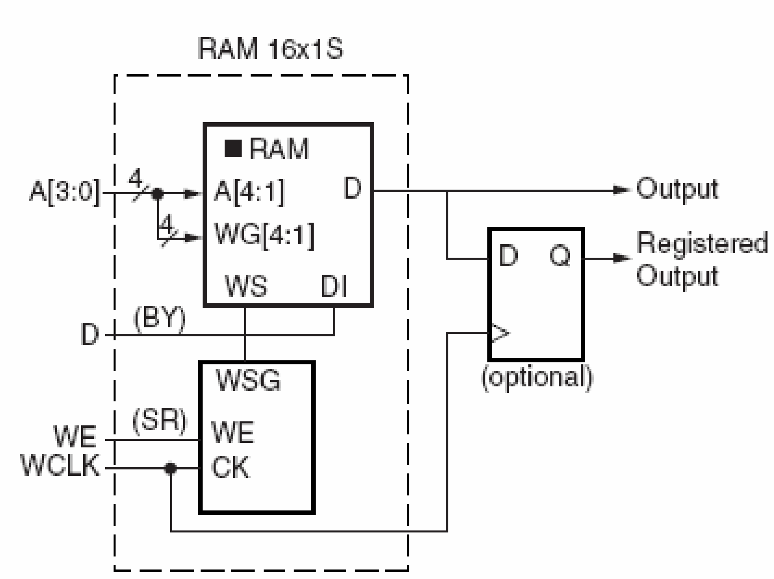


**Figure 6. Single-Port Distributed SelectRAM+ [15]**

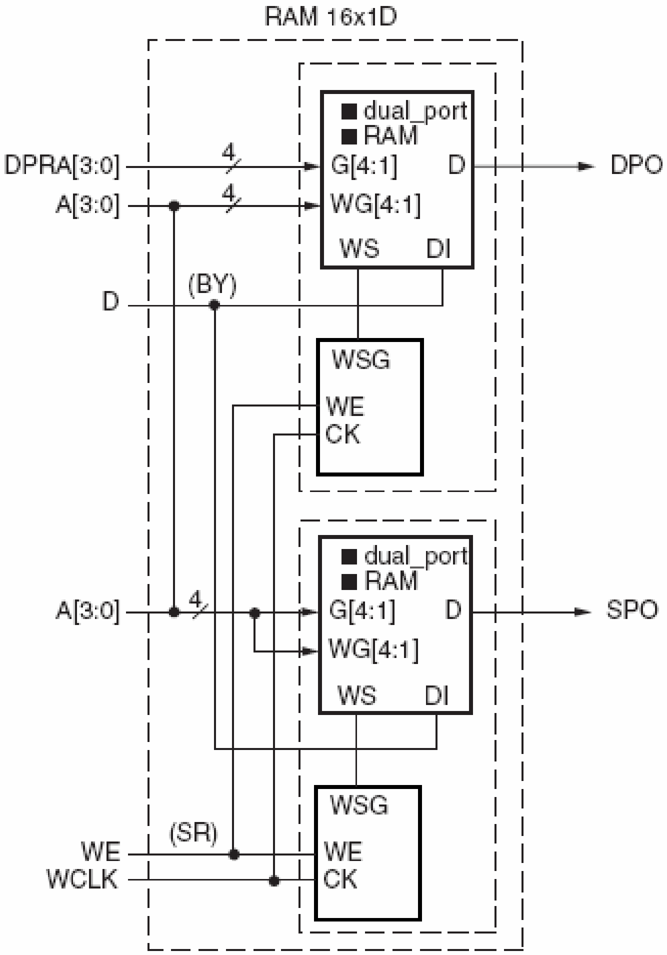


**Figure 7. Dual-Port Distributed SelectRAM+ [15]**

### 2.3.2.3 Block SelectRAM+

In addition to distributed memory, the Virtex-II Pro family devices include a large amount of 18 Kb block SelectRAM+ (BRAM) resources, programmable from 16K x 1 bit to 512 x 36 bit, in various depth and width configurations. These SelectRAM+ blocks are real dual-port RAM blocks which can be configured as FIFO easily. In the VirtexII pro xc2vp100, there are 444 18 Kb SelectRAM+ blocks that make it totally 1MB block SelectRAM+, Table 3 shows the supported memory configurations for single-port and dual-port modes.

**Table 3. Block SelectRAM Configuration Modes [15]**

| | |
|---|---|
| 16Kx1 bit | 2Kx9 bits |
| 8Kx2 bits | 1Kx18 bits |
| 4Kx4 bits | 512x36 bits |

Each port in SelectRAM+ Blocks has the following inputs: Clock, Clock Enable, Write Enable, Set/Reset, Address, and separate Data and Parity buses for inputs and outputs. Both write and read operations are synchronous.

As a single-port RAM, the block SelectRAM+ has access to the 18 Kb memory locations in any of the configurations in Table 3. Each block SelectRAM+ cell is a fully synchronous memory as illustrated in Figure 8. Input data bus and output data bus widths are identical.

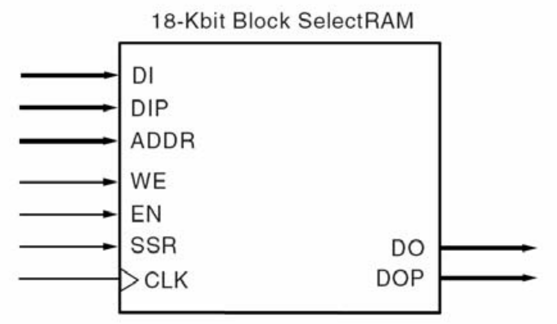


**Figure 8. Block SelectRAM in Single-Port Mode [15]**

As a dual-port RAM, each port of block SelectRAM+ has access to a common 18 Kb memory resource. These are fully synchronous ports with independent control signals for each port as illustrated in Figure 9. The data widths of the two ports can be configured independently.

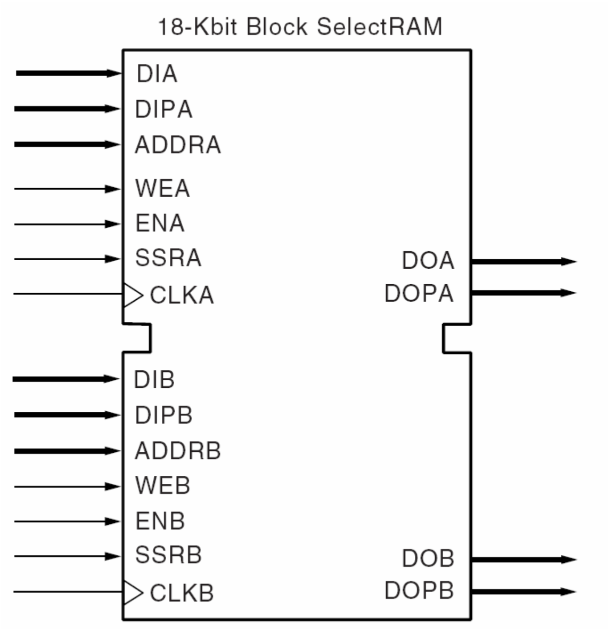


**Figure 9. Block SelectRAM in Dual-Port Mode[15]**

Since the SelectRAM+ blocks have a regular array structure, the place-and-route tools take advantage of this feature to achieve optimum system performance and fast compile times. [15].

Distributed SelectedRAM+ and Block SelectedRAM+ are used for storage components in the AES-GCM implementation discussed in chapter 4. For example, the single port mode of the SelectedRAM+ or Block SelectedRAM+ is used to implement the S-box Look Up tables (see section 3.1.2); Also the dual port mode of  the SelectedRAM+ or Block SelectedRAM+ is used to implement a 11x128-bit FIFO.

## 2.4 The CMC-Prototyping Platform

The CMC-Prototyping Platform [13][14][16][24] is a prototype system provided by Canadian Microelectronics Corporation (CMC) to Canadian universities for research purpose only. It consists of:

- AMIRIX AP1000 PCI Platform FPGA Development Board
- Xilinx Parallel-IV download cable kit and dual serial adaptor cable
- IBM Intellistation Z Pro workstation
- EDA tools for FPGA Development
- AP1000 PCI Platform FPGA Development Kit Baseline Platform

The major focus is the AP1000 FPGA Board. Figure 10 illustrates the AP1100 FPGA board hardware architecture. The DDR SDRAM banks provide 64-bit Data width for the two IBM PowerPC microprocessors which are the hardware IP cores inside the FPGA VirtexII pro xc2vp100. If the FPGA works in Base-line mode which is established by AMIRIX, all these memory blocks are controlled by the soft IP cores inside the FPGA. There are also synchronous SRAMs providing large data storage space. Configuration Flash, program Flash and System ACE are accessible through the local bus interface. The kernel driver is stored in program flash on the board. A ramdisk image is also stored in the program flash. By default, the AP1000 will load the kernel and mount the ramdisk when powered on. U-Boot which is stored in Program Flash is a bootloader program that provides the ability to load Linux, as well as a monitor program that allows access to the AP1000 resources. U-Boot is transferred to DDR-SDRAM memory for execution when it is selected to run.

The Processor Bus Dual PCI Bridge provides an interface to other devices on the local PCI bus. Through this local PCI bus, FPGA Virtex-II Pro can communicate with host PC by a PCI-to-PCI bridge device. The FPGA also has access to a PMC module if installed and an Ethernet transceiver for network access. Furthermore, the extra high-speed network interfaces can be implemented through the two Gigabit Ethernet PHY devices that are connected directly to the Virtex-II Pro if the relative soft MAC layer IP cores are downloaded to FPGA.

Expansion connectors are available through the Expansion I/O ports on the board. These expansion ports allow either cabling or custom PCB daughter cards to be directly connected to the Virtex-II Pro. CompactFlash, Ethernet, and RS-232D connectors are accessible from outside the chassis.

The Virtex-II pro can work in either of the two work modes on CMC-prototype platform, one is called Base-line mode which is supposed to be the default mode for applying the FPGA on board; one is called Stand-alone mode which is a fully custom mode.

If the Virtex-II pro works in Base-line mode, approximately 93 % of the FPGA resources are free[24]. There are two internal buses inside it (see Figure 10), one is IBM Processor Local Bus (PLB), and one is IBM On-Chip Peripheral Bus (OPB). Two PowerPC CPU hardware IP cores and DDR-SDRAM Controller IP core are allocated on the 64-bit 80MHz PLB bus. Interrupt Controller and UART core are allocated on 32-bit 40MHz OPB bus. The PLB bus indirectly connects to the local PCI bus on the board; and the OPB bus also indirectly connects to the external local bus on the board. All these made IP cores above set up a System on a Chip (SoC) environment (i.e. so-called Base-line mode) for further application. About 7% resource of the Virtex-II pro are occupied by these IP cores.

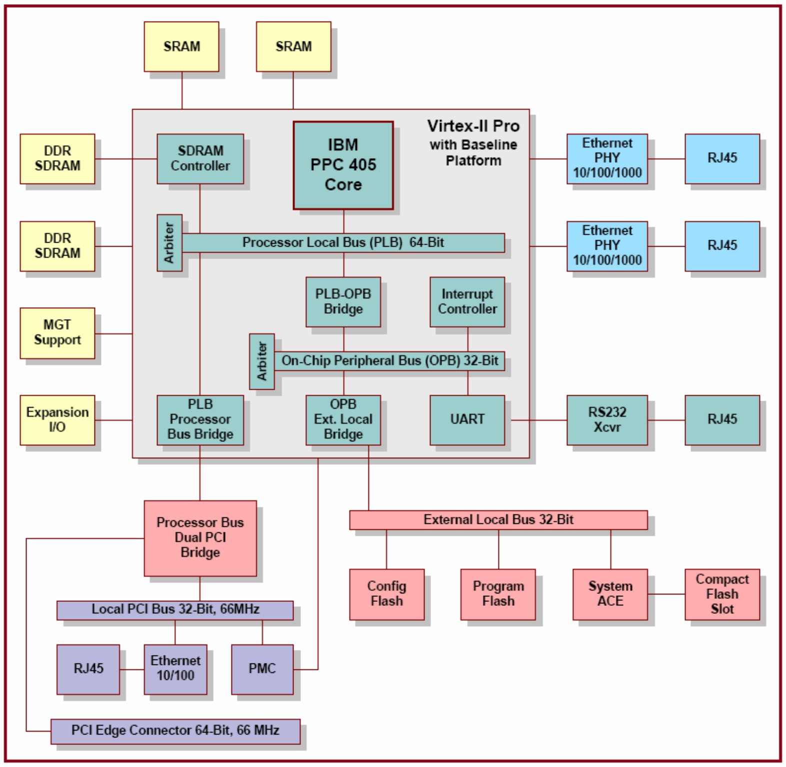


**Figure 10. AP1100 Board Architecture [24]**

If the Virtex-II pro works in Stand-alone mode, all the resources on the FPGA are free for customers. Of course, no PLB bus, no OPB bus, and no SDRAM controller etc. are inside the FPGA.

# Chapter 3
# Security Standard

In this chapter, the fundamental of AES and GCM algorithms will be presented. Section 3.1 introduces the Advanced Encryption Standard (AES). Section 3.2 introduces the Galois/Counter Mode (GCM).

## 3.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) was published by NIST in 2001. AES is a symmetric block cipher that operates on 128-bit block as input and output data. The algorithm can encrypt and decrypt blocks using a secret key which has a key size of 256-bit, 192-bit, or 128-bit. One of the main features of AES is simplicity that is achieved by repeatedly combining substitution and permutation computations at different rounds. That is, AES encrypts/decrypts a 128-bit plaintext/ciphertext by repeatedly applying the same round transformation a number of times depending on the key size, see Table 4.

**Table 4. Key-Block-Round Combinations [3]**

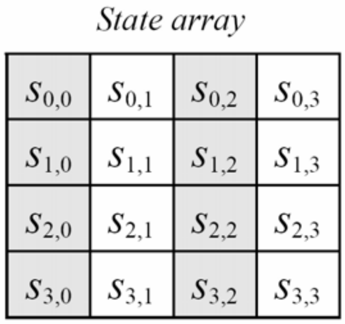
| | Key Length<br>(32-bit word) | Block Size<br>(32-bit word) | Number of Rounds |
|---|---|---|---|
| AES-128 | 4 | 4 | 10 |
| AES-192 | 6 | 4 | 12 |
| AES-256 | 8 | 4 | 14 |

The actual key size depends on the desired security level. Today, AES-128 is predominant and supported by most hardware implementations. In this section, the AES forward cipher operation (i.e. AES encryption) with 128-bit key is mainly discussed since it is invoked in the GCTR module of the AES-GCM standard to provide confidentiality.

### 3.1.1 AES Cipher

For 128-bit key size, there are 10 rounds substitutions and permutations that have to be executed in AES cipher (see Table 4). The input 128-bit plaintext is presented in a 4x4 matrix of bytes. Thus, there are 32 bits each row and each column in the matrix. This matrix is also called State array which is illustrated in Figure 11. In Figure 11, $S_{i,j}$ indicates a byte, where $0 \leq i,j \leq 3$. The state array is altered in each round. The input key is expanded into an array of forty four 32-bit words, and each 4 words of the expanded key will be used in each round. The key expansion should be done before the cipher operation. Each round transformation consists of four phases as follows:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey



*State array*

| | | | |
|---|---|---|---|
| $s_{0,0}$ | $s_{0,1}$ | $s_{0,2}$ | $s_{0,3}$ |
| $s_{1,0}$ | $s_{1,1}$ | $s_{1,2}$ | $s_{1,3}$ |
| $s_{2,0}$ | $s_{2,1}$ | $s_{2,2}$ | $s_{2,3}$ |
| $s_{3,0}$ | $s_{3,1}$ | $s_{3,2}$ | $s_{3,3}$ |

**Figure 11. Illustration of State Array [3]**

**SubBytes** The function SubBytes is the only non-linear function in AES. It substitutes all bytes of the state array using a LUT which is a 16x16 matrix of bytes, often called S-box. The S-box is used for SubBytes operation that contains the results of substitution and permutation of all possible 8-bit values. The content of the LUT can be computed by a finite-field inversion followed by an affine transformation over GF($2^8$). Each byte of state is mapped into a byte from the S-box; The 4 leftmost bits are used as the row index while the 4 rightmost bits are used as the column index. Figure 12 illustrates the effect of the SubBytes transformation on the State array. The S-box is designed to be resistance to known cryptanalytic attacks [18]. SubBytes function has a property that the output cannot be described as a simple mathematical function of the input.

In this thesis, two schemes to implement S-box are discussed, one based on Block SelectRAM+, and one based on Distributed SelectRAM+.

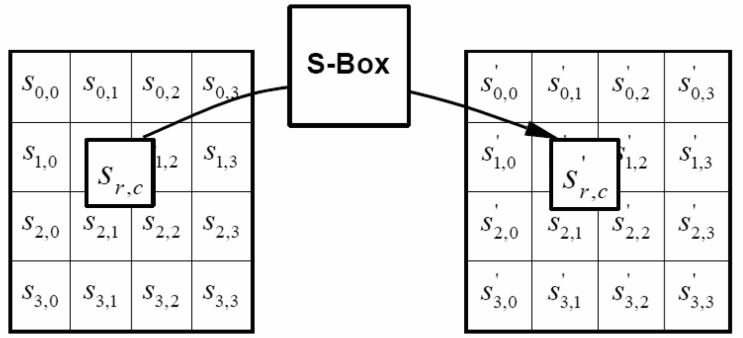
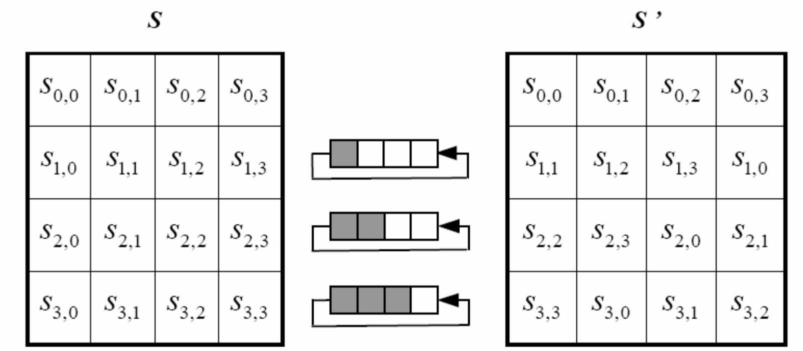

S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$S_{r,c}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$
$S'_{r,c}$

**Figure 12. Illustration of SubBytes Operation [3]**

**ShiftRows** In the ShiftRows transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (see Figure 13). The first row is not shifted. The second row is left-shifted circularly one byte. For the third row, a 2-byte circular left shift is performed. For the forth row, a 3-byte circular left shift is performed. Since the MixColumns and AddRoundKey operations are done column by column, ShiftRows ensures that 4 bytes of one column are spread out to four different columns. Figure 13 illustrates the effect of the ShiftRows transformation on the State array.



| | | | |
|---|---|---|---|
| $s_{0,0}$ | $s_{0,1}$ | $s_{0,2}$ | $s_{0,3}$ |
| $s_{1,0}$ | $s_{1,1}$ | $s_{1,2}$ | $s_{1,3}$ |
| $s_{2,0}$ | $s_{2,1}$ | $s_{2,2}$ | $s_{2,3}$ |
| $s_{3,0}$ | $s_{3,1}$ | $s_{3,2}$ | $s_{3,3}$ |

| | | | |
|---|---|---|---|
| $s_{0,0}$ | $s_{0,1}$ | $s_{0,2}$ | $s_{0,3}$ |
| $s_{1,1}$ | $s_{1,2}$ | $s_{1,3}$ | $s_{1,0}$ |
| $s_{2,2}$ | $s_{2,3}$ | $s_{2,0}$ | $s_{2,1}$ |
| $s_{3,3}$ | $s_{3,0}$ | $s_{3,1}$ | $s_{3,2}$ |

**Figure 13. Illustration of ShiftRows Transformation [3]**

**MixColumns** MixColumns function operates on the state column by column. Each byte of a column in state array is mapped into a new value that is a function of all the four bytes in that column as follows:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s0,0 & s0,1 & s0,2 & s0,3 \\ s1,0 & s1,1 & s1,2 & s1,3 \\ s2,0 & s2,1 & s2,2 & s2,3 \\ s3,0 & s3,1 & s3,2 & s3,3 \end{bmatrix} = \begin{bmatrix} s'0,0 & s'0,1 & s'0,2 & s'0,3 \\ s'1,0 & s'1,1 & s'1,2 & s'1,3 \\ s'2,0 & s'2,1 & s'2,2 & s'2,3 \\ s'3,0 & s'3,1 & s'3,2 & s'3,3 \end{bmatrix}$$

MixColumns operation ensures a good mixing among the bytes of each column [18]. ShiftRows and MixColumns together ensure that after executing the rounds all output bits depend on all input bits.

**AddRoundKey** AddRoundKey operation is designed as a stream cipher; all the 128 bits of state are XORed with 4 32-bit words of expanded key resulting from key expansion. AddRoundKey is the only operation that involves using the key to ensure security.

The AES with 128-bit key size forward cipher operation is shown in Figure 14. In Figure 14, w[i,i+3] indicates 4 words of expanded key resulting from key expansion, where 0≤ i ≤ 40 (see section 3.1.2).

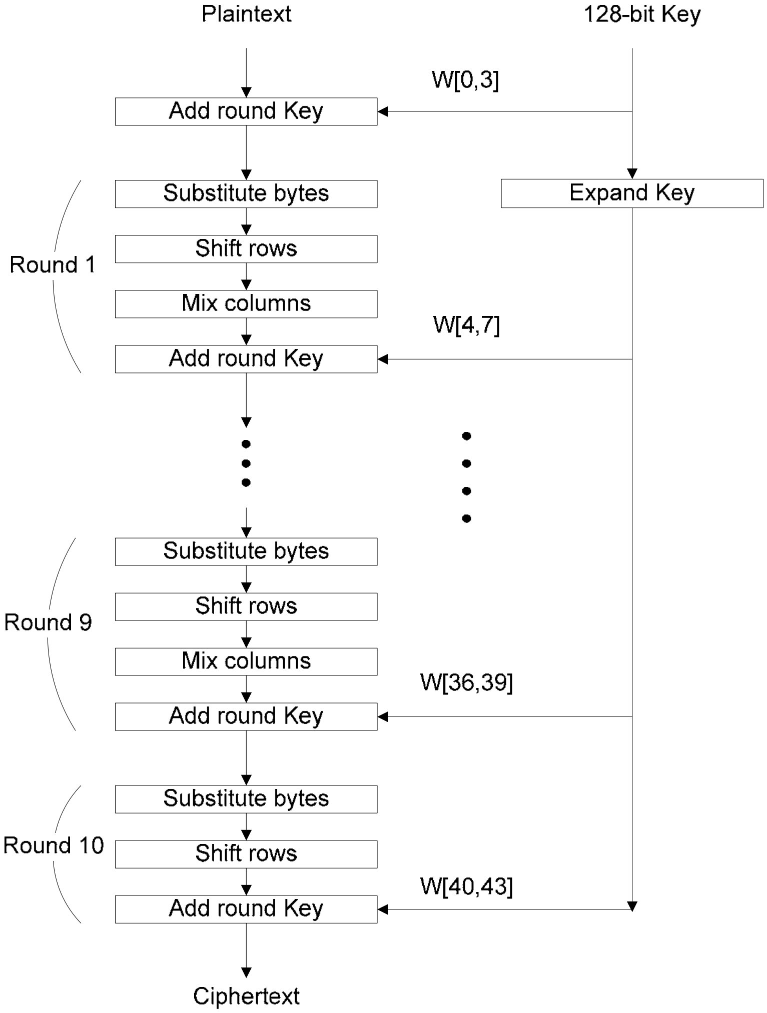


**Figure 14. AES Forward Cipher Operation (Pipelining Data Path)**

Decryption is a reverse of encryption which inverse round transformations to computes out the original plaintext of an encrypted ciphertext in reverse order. The round transformation of decryption uses the functions AddRoundKey, InvMixColumns, InvShiftRows, and InvSubBytes successively. AddRoundKey is its own inverse function because the XOR function is its own inverse. The round keys have to be selected in reverse order. InvMixColumns needs a different constant polynomial than Mix-Columns does. InvShiftRows rotates the bytes to the right instead of to the left. InvSubBytes reverses the S-

Box look-up table by an inverse affine transformation followed by the same inversion over GF($2^8$) which is used for encryption. About more details information of AES Decryption, refer to [3] and [18].

### 3.1.2 Key Expansion

The Key expansion operation takes the 128-bit key as the input for each session and yields a 44 32-bit words expanded key array as its output. In each round, AES cipher uses 4 words of the 44-word expanded key in AddRoundKey transformation, like shown in Figures 14. Figure 15 is the illustration how to expand the Key. The first 4 words of the output array is nothing but the 16-byte input secret key. Except the words whose indexes are multiple of four, the other words are simply made by XORing the preceding word with the word four positions back. The words whose indexes are multiple of four go through a more complex function, called function g before XORing with the word four positions back.

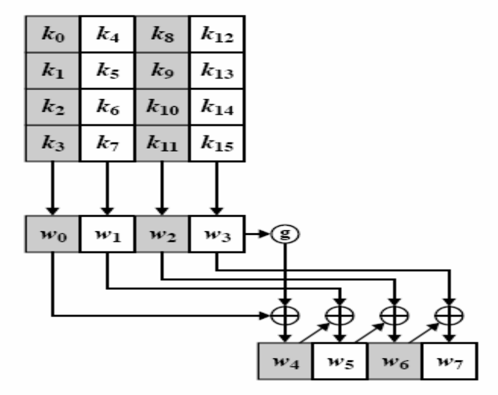


**Figure 15. Key Expansion [18]**

The function g takes the preceding word performs a one-byte circular left shift, then it performs SubBytes operation on each byte of the shifted result. In the last step it takes the substituted word and XORs it with a round constant hexadecimal word array “RC(i), 0, 0, 0”, where, 1≤ i ≤10. RC(i) is given in Table 5 in hexadecimal for each round. The purpose of using round constants is to eliminate symmetries and similarities in making the 4-word expanded key for each round.

**Table 5. Round Constant Bytes, RC in Hexadecimal [18]**

| I (round number) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| RC(i) | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 |

## 3.2 Galois/Counter Mode (GCM)

The elements of GCM and the associated notation and requirements are introduced in the three sections below. The block cipher and key are discussed in Sec. 3.2.1. The data elements of the authenticated encryption and authenticated decryption functions of GCM are discussed in Sec. 3.2.2. The types of application of GCM supposed in [2] are summarized in Sec. 3.2.3. The GHASH function, GCTR function and GCM specification are described in section 3.2.4, 3.2.5 and 3.2.6, respectively.

### 3.2.1 Block Cipher

The AES-GCM standard depends on the symmetric key block cipher AES. The AES-GCM key is the block cipher key. The key shall be generated uniformly at random, or close to uniformly at random. The key should be established secretly among the parties to communicate. AES-GCM designates the encryption function of the block cipher AES as the forward cipher function denoted $CIPH_K$ which actually is AES in ECB mode (see Figure 21). GCM does not employ the inverse cipher function.

### 3.2.2 Input and Output Data

GCM consists of the two functions that are called authenticated encryption and authenticated decryption. The requirements and notation for the input and output data of these functions are introduced in Section 3.2.2.1 and 3.2.2.2.

#### 3.2.2.1 Authenticated Encryption

There are three input bit streams to the authenticated encryption operation:

- A plaintext, denoted as P that can have up to$2^{39}$ bits;
- Additional authenticated data (AAD), denoted as A that can have up to $2^{64}$ bits;
- An initialization vector denoted, as IV that can have up to $2^{64}$ bits.

In this thesis, a 96-bit IV is adopted for efficiency following the suggestion in [12].

GCM verifies the authenticity of both P and AAD; GCM also protects the confidentiality of P, while the AAD is transmitted in the clear. The IV is a nonce that is associated with the data to be against related attack.

The following two bit strings comprise the output data of the authenticated encryption function:

- A ciphertext, denoted C, with the same bit length as that of the plaintext.
- An authentication tag, denoted T that have up to 128 bits. The T's bit length is denoted as t.

#### 3.2.2.2 Authenticated Decryption

The inputs to the authenticated decryption function are values for IV, A, C, and T, as described in Sec. 3.2.2.1 above. The output is one of the following:

- The plaintext P that corresponds to the ciphertext C, or

- An indication that the inputs are not authentic, denoted as FAIL.

GCM authenticated decryption computes the authentication tag T' based on received data, and compares it with the received authentication tag T. If the two tags T and T' are equal, then P will be the output of the authenticated decryption function. Otherwise, FAIL will be the output.

### 3.2.3 Types of Applications of GCM

There are four types of applications of GCM that are recommended in SP800-380D. They are

1) GCM with an arbitrary length IV,

2) GCM with the default IV, i.e. the length of the IV is restricted to exactly 96 bits.

3) GMAC, i.e. the algorithm generates a stand-alone authentication tag T on the AAD with the arbitrary length IV. The plaintext P is the empty string.

4) GMAC with the default IV.

In the thesis, GCM with the default IV is chosen and will be discussed in Sec. 4.

### 3.2.4 GHASH Function

The authentication mechanism within GCM is based on the hash function, GHASH, that features multiplication by a fixed hash subkey, over a binary Galois field GF($2^{128}$). The hash subkey, denoted as H, is generated by applying the block cipher to the 128-bit "0" string. GHASH is a keyed hash function. Algorithm 3 below specifies the function that will be invoked within the AES-GCM authenticated encryption and authenticated decryption functions:

**Algorithm 3**: $\text{GHASH}_H$ (X)

Input:

1. Bit string X with length len(X) = 128·m for some integer m.

2. The hash subkey H.

Output:

Block $Y_m$.

Steps:

1. Let $X_1$, $X_2$, ... , $X_{m-1}$, $X_m$ represents the unique sequence of blocks such that X= $X_1$ || $X_2$ || ... || $X_{m-1}$ || $X_m$.
2. Let $Y_0$ be the "zero block," which means $Y_0$ is a bit string comprised by 128 binary 0.
3. For i = 1, ..., m, let $Y_i = (Y_{i-1} \oplus X_i) \cdot H$. where "·" indicates multiplication over finite field as discussed in chapter 2.
4. Return $Y_m$.

The GHASH function is illustrated in Figure 16 below.

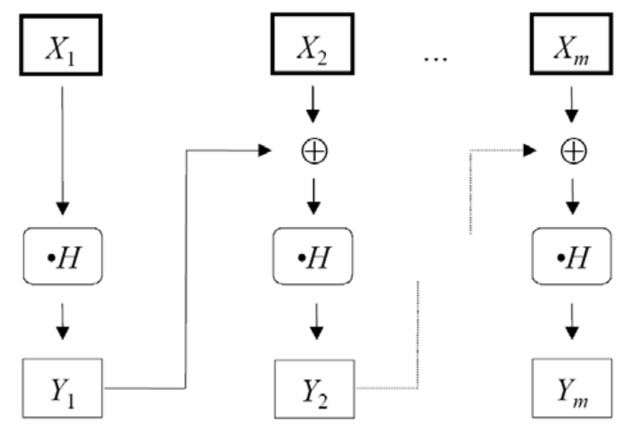


**Figure 16. GHASH$_H$ (X1 || X2 || ... || Xm) = Ym.[2]**

### 3.2.5 GCTR Function

The mechanism for the confidentiality of GCM is a variation of the CTR mode (see section 2.2.2.), called GCTR, with a particular incrementing function, denoted inc, for generating the necessary sequence of counter blocks. The first counter block for the plaintext encryption is generated by incrementing a block that is derived from IV.

Algorithm 4 below specifies the GCTR function that will be invoked within the algorithms for the GCM authenticated encryption and authenticated decryption functions:

**Algorithm 4**: $\mathrm{GCTR}_K$ (ICB, X)

Input:

1. Bit string X, of arbitrary length;
2. Initial counter block ICB, i.e. IV or some value generated from IV;
3. Approved block cipher CIPH (such as AES) with a 128-bit block size;
4. Key K;

Output:

Bit string Y of bit length len(X).

Steps:

1. Let $n=\lceil \mathrm{Len}(X)/128 \rceil$.
2. Let $X_1, X_2, \ldots, X_{n-1}, X^*_n$ denote the unique sequence of bit strings such that $X = X_1 \| X_2 \| \ldots \| X_{n-1} \| X^*_n$.
3. Let $CB_1 = ICB$.
4. For i=2 to n, let $CB_i = \mathrm{inc}(CB_{i-1})$.
5. For i= 1 to n-1, let $Y_i=X_i \oplus \mathrm{CIPH}_K(CB_i)$.
6. Let $Y^*_n=X^*_n \oplus \mathrm{MSB}_{\mathrm{len}(X^*n)}(\mathrm{CIPH}_K(CB_n))$.

7. Let $Y = Y_1 \| Y_2 \|... \| Y_{n-1} \| Y^*_n$.
8. Return Y.

Note:

1. Len(X) indicates the bit length of the bit string X.
2. $X_i \| X_{i+1}$ indicates the concatenation of two bit strings $X_i$ and $X_{i+1}$.
3. LSBs (X) indicates the bit string consisting of the s right-most bits of the bit string X.
4. MSBs (X) indicates the bit string consisting of the s left-most bits of the bit string X.
5. Int(X) indicates the integer for which the bit string X is a binary representation.
6. Inc(X) indicates the output of the GCM incrementing function applied to the block X, the more specifically, $\text{inc}(X)=\text{MSB}_{96}(X) \|[(\text{int}(\text{LSB}_{32}(X))+1) \bmod 2^{32}]_{32}$.

Figure 17 below illustrates the GCTR function.

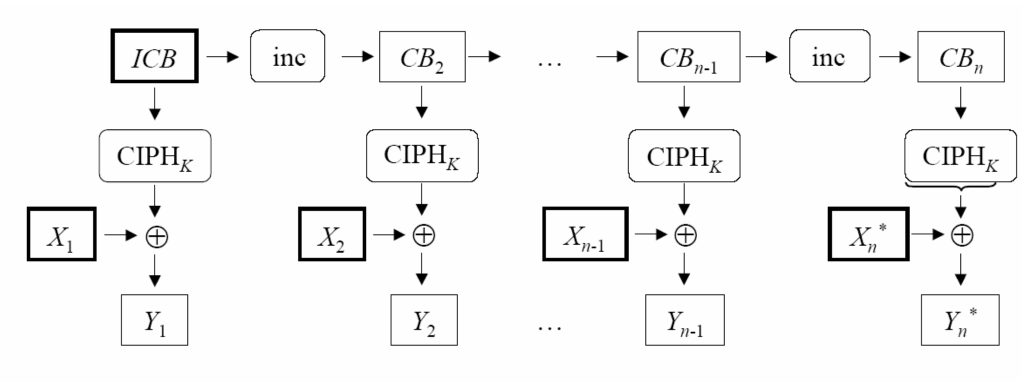


**Figure 17. $\text{GCTR}_K$ (ICB, X1 || X2 || ... || X*n) = Y1 || Y2 || ... || Y*n.[2]**

### 3.2.6 GCM Specification

Algorithms for the authenticated encryption and authenticated decryption functions of GCM are specified in Section 3.2.6.1, and 3.2.6.2 below. The block cipher is AES (see section 3.1).

#### 3.2.6.1 Authenticated Encryption

Algorithm 5 below performs the authenticated encryption function.

**Algorithm 5**: AES-GCM-$AE_K$ (IV, P, A)

Input:

1. Block cipher CIPH (i.e. AES) with a 128-bit block size;
2. Key K;
3. Tag length t.
4. Initialization vector IV;
5. Plaintext P;
6. Additional authenticated data A.

Output:

1. Cipher text C;
2. Authentication tag T.

Steps:

1. Let H = $CIPH_K(0^{128})$
2. Define a block, $J_0$, as follows: $J_0$=IV||$0^{31}$l, i.e. $J_0$ is a 128-bit string consisted of 96-bit IV, 31 ‘0’ bits, and1 ‘1’ bit.
3. Let C = $GCTR_K(inc(J_0), P)$.

4. Let u = 128·⌈len(C)/128⌉-len(C), and let v =128·⌈len(A)/128⌉-len(A)
5. Define a block, S, as follows: S = $\text{GHASH}_H$ (A||$0^v$||C||$0^u$||[len(A)]$_{64}$||[len(C)]$_{64}$)
6. Let T = $\text{MSB}_t(\text{GCTR}_K(J_0,S))$.
7. Return (C, T).

Notes:

1. $[x]_s$ indicates the binary representation of the non-negative integer x as a string of s bits, where x<2s.
2. $0^s$ denotes the string that consists of s ‘0’ bits. For example, $0^5$ = B00000.

The authenticated encryption function is illustrated in Figure 18 below.

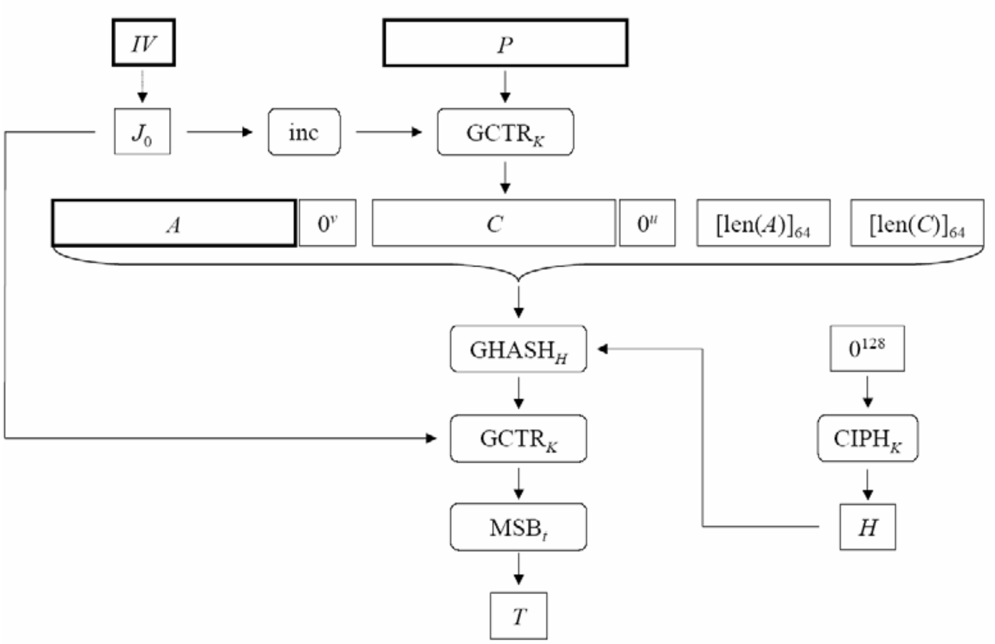


**Figure 18. AES-GCM-AE$_K$ (IV, P, A) = (C, T).[2]**

### 3.2.6.2 Authenticated Decryption

Algorithm 6 below performs the authenticated decryption function.

**Algorithm 6**: AES-GCM-AD$_K$ (IV, C, A, T)

Input:

1. Block cipher CIPH (i.e. AES) with a 128-bit block size;
2. Key K;
3. Tag length t.
4. Initialization vector IV;
5. Cipher text C;
6. Additional authenticated data A.
7. Authentication tag T.

Output:

Plaintext P or indication of inauthenticity FAIL;

Steps:

1. Let H = CIPH$_K$($0^{128}$)
2. Define a block, $J_0$, as follows: $J_0$=IV||$0^{31}$1. i.e. $J_0$ is a 128-bit string consisted of 96-bit IV, 31 '0' bits, and1 '1' bit.
3. Let P = GCTR$_K$(inc($J_0$),C).
4. Let u = 128·$\lceil$len(C)/128$\rceil$-len(C), and let v =128·$\lceil$len(A)/128$\rceil$-len(A)
5. Define a block, S, as follows: S = GHASH$_H$(A||$0^v$||C||$0^u$||[len(A)]$_{64}$||[len(C)]$_{64}$)
6. Let T'= MSB$_t$(GCTR$_K$($J_0$,S)).
7. If T = T', then return P; else return FAIL.

The authenticated decryption function is illustrated in Figure 19 below.

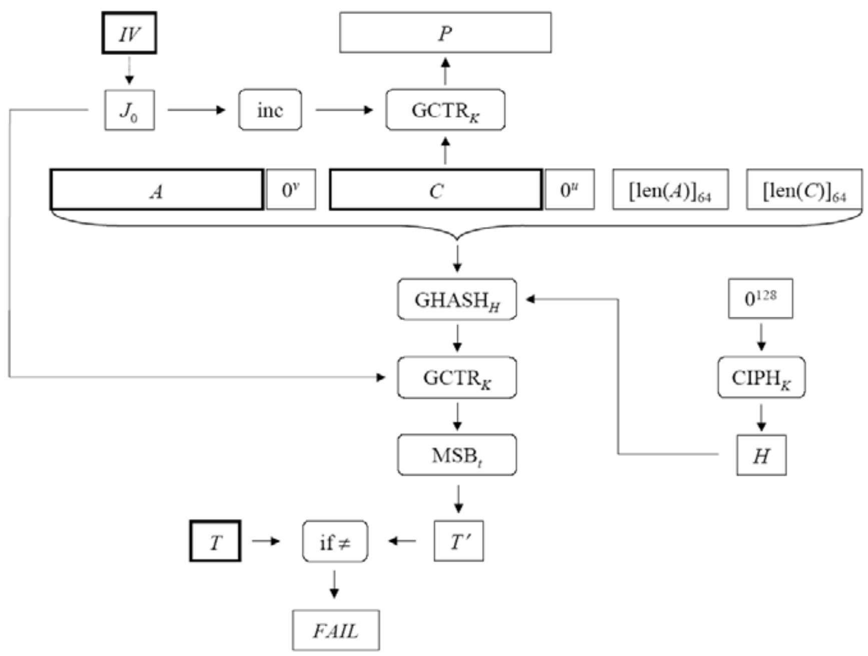


**Figure 19. AES-GCM-AD$_K$ (IV, C, A, T) = P or FAIL.[2]**

# Chapter 4
# The Architecture of AES-GCM

This chapter describes the AES-GCM implementation on the CMC-Prototype-Platform. Section 4.1 discusses the architectures of the modules of AES and GHASH, section 4.2 discusses the architectures of the AES-GCM, including IPsec data packet, and GCM data flow. Section 4.3 discusses how to verify the AES-GCM functionality.

## 4.1 Modules Design

In AES-GCM encryption and AES-GCM decryption, AES and GHASH are the basic modules which are responsible for confidentiality and authentication, respectively. In section 4.1.1, an iterative AES and fully pipelined AES are presented; in section 4.1.2, a bit serial GHASH and a bit parallel GHASH are presented. The pipelined AES and parallel-bit GHASH modules are selected for designing a high speed AES-GCM architecture discussed in section 4.2.

### 4.1.1 AES Module

For the 128-bit key size, the AES algorithm requires calculating 10 round transformations, and each round contains four phases: SubBytes, ShiftRows, MixColumns, and AddRoundKey (see section 3.1). This allows implementing AES algorithm in either iterative method or pipelined method. In an iterative AES design, the round transformation is instantiated only once (see Figure 20). This round transformation block of hardware is used 10 times in 10 computation clock cycles while the intermediate value is stored in a register Ciphertext and used as input for the next time. A pipelined AES design can calculate all 10 rounds transformations in one clock cycle by duplicated a single round 10 times (see Figure 14). A pipelined AES architecture can be achieved by placing 128-bit registers between each round. In large FPGAs, registers are almost free; a pipelined structure can take advantage of this feature of FPGAs.

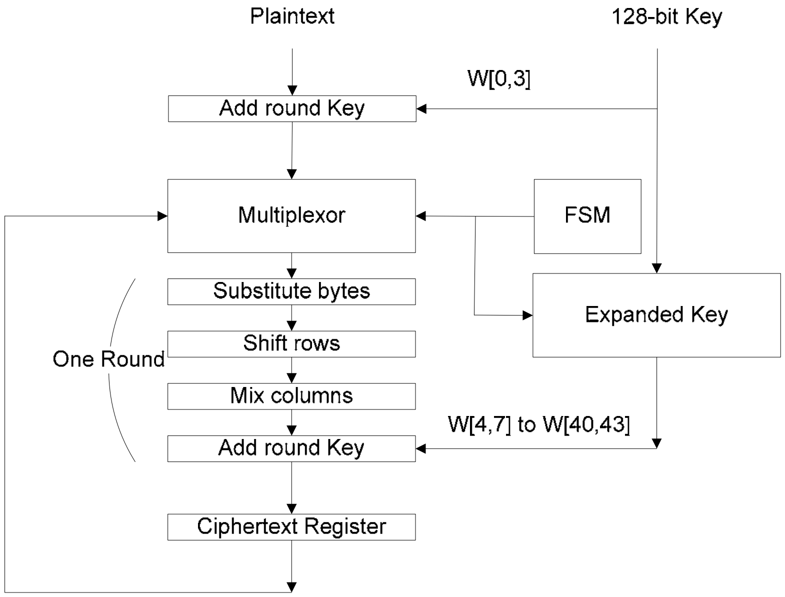


**Figure 20. AES Iterative Data Path**

**Table 6. Comparison between Iterative and Pipelined AES**

| Architectures (expanded keys stored in CLBs) | Num of unrolled rounds in hardware | Num of ciphertext per 10 clock cycles | Cost in Virtex-II pro | |
|---|---|---|---|---|
| | | | With BRAM | Without BRAM |
| Iterative AES | 1 | 1 | 418slices+16BRAMs | 1678slices |
| Pipelined AES | 10 | 10 | 1613slices+160BRAMs | 11693slices |

The control logic of both the iterative and pipelined AES architectures is implemented by using a finite state machine (FSM). Table 6 shows a rough comparison between these two approaches on throughput and cost.

In AES-GCM algorithm, the AES block is implemented in a pipelined architecture. This AES block works as a core in a hybrid from ECB and CTR mode. This hybrid actually is

GCTR(see section 3.2.5). Figure 21 shows the structure how the AES block is embedded into an ECB module and how the ECB module is embedded into a CTR module. In ECB module, AES block encrypts an input which actually is a continuously increasing counter value in CTR module, and produces the output as keystream. In CTR module, the keystream XORs a plaintext to produce output, i.e. ciphertext. After the first 10 clock cycles, the pipeline is fully filled so that the AES module can output a new 128-bit keystream every clock cycle.

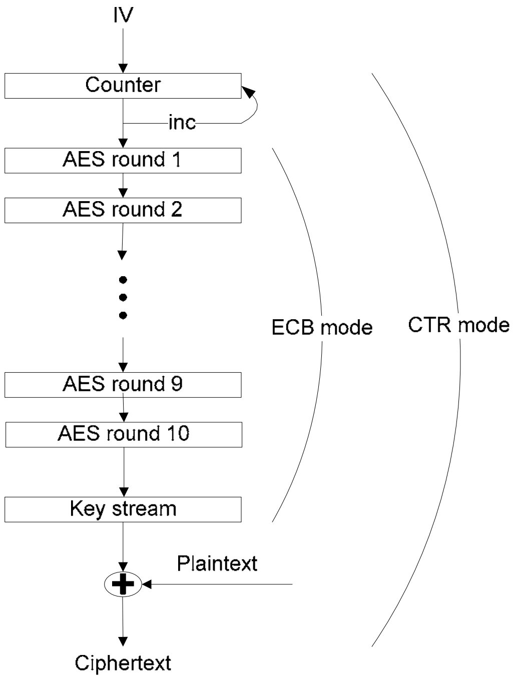


**Figure 21. AES CTR over ECB Mode Cipher Structure**

The iterative AES can also be adopted in the AES-GCM algorithm, specifically for generating the hash subkey H. This calculation can be done in advance if the 128-bit key is known because H is nothing but the output of the iterative AES module. Therefore, it needs 10 clock cycles to generate H after inputting 128 bits '0' string into the iterative AES module.

As briefly mentioned in section 3.1.1, except for the SubBytes operation in round transformation, the ShiftRows, MixColumns, and AddRoundKey are all directly designed using CLBs in FPGA. SubBytes which actually is a LUT operation can be designed either using CLBs (i.e single-port distributed SelectRAM+, see section 2.3.2.3) or signal-port block selectRAM+ (see section 2.3.2.4). Table 8 in section 4.2.3 shows the differences in performance and cost between AES-GCM implementations which are purely using CLBs and those using both CLBs and Block RAMs.

### 4.1.2 GHASH Module

A 128-bit multiplier over GF($2^{128}$) is the core of the GHASH architecture. In AES-GCM, the GF($2^{128}$) multiplier multiplies two 128-bit operands modulo the field polynomial $F(x) = 1 + x + x^2 + x^7 + x^{128}$ to generate a 128-bit output. The GHASH architecture is shown in Figure 22. One operand of the GF multiplier is the hash subkey H which can be treated as a fixed 128-bit constant for it will not change if the 128-bit key does not change. The Register Y whose initial value is zero holds the intermediate hash value for next step authentication computation.

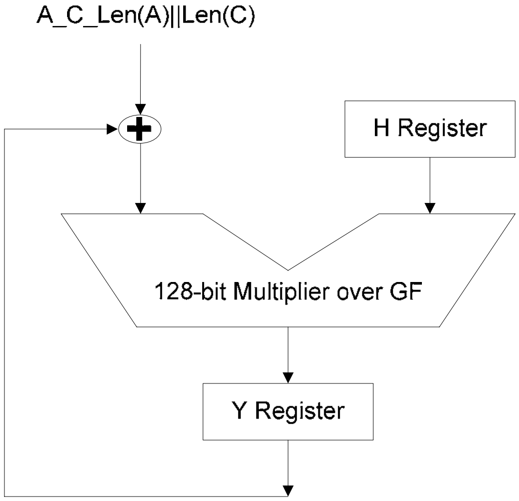


**Figure 22. GHASH Hardware Architecture**

The architecture shown in Figure 22 is based on an iteration operation. Suppose all input data and output data are satisfied with the definitions in section 3.2.4. In the first m clock cycles, the 128-bit additional authenticated data block sequence (AAD) $A_1$, $A_2$, …, $A_m$ are hashed to the GHASH through one of two inputs of XOR gates as described by algorithm 3. In the next n clock cycles, the 128-bit ciphertext block sequence $C_1$, $C_2$, …, $C_{n-1}$, $C_n$ are hashed to the same input of XOR gates following AAD. In the last clock cycle, 128-bit word length (A)||length(C) is hashed. Meanwhile, the intermediate hash value $Y_i$ (see Figure 16) is fed back to another input of XOR gates to generate the another operand for the GF multiplier.

It takes m + n + 1 cycles to compute the hash value for bit parallel multiplier, and 128∗(m + n + 1) cycles for Bit Serial multiplier. There is a rough comparison listed in Table 7 between GHASH architectures using these two kinds of multipliers.

**Table 7. Comparison between Different GHASH Architectures**

| GHASH architecture | Latency (clock cycle) | Hardware complexity (k=128) | Num of Slices |
|---|---|---|---|
| Using Bit Serial Multiplier | 128 * (m + n + 1) | O(k) | 282 |
| Using Mastrovito Bit Parallel Multiplier | m + n +1 | $O(k^2)$ | 8297 |

In order to match pipelined AES module, the GHASH module is implemented using a Mastrovito Bit Parallel GF multiplier. From a whole Mastrovito Bit Parallel $GF(2^{128})$ multiplier point of view, $128^2$ two-input AND gates and $O(128^2)$ two-input XOR gates were used for implementation. The delay from this architecture is one AND gate and 7 XOR gates. This is the critical path in the entire AES-GCM circuit design. Although a Bit Parallel multiplier over GF can be pipelined for high data rate [19], this is not the case for GHASH because the GHASH is a kind of feedback mode as mentioned in section 1.1.

## 4.2 High Speed Hardware Implementation of AES-GCM

This section describes the AES-GCM implementation. It begins by a brief introduction on the data packet structure of IPsec ESP [9] in section 4.2.1, then follows with a top level data flow description of the pipelined AES and bit parallel GHASH modules in section 4.2.2. Finally the details of the AES-GCM implementation are presented in section 4.2.3.

### 4.2.1 Format of Data Packet of IPsec ESP

The IPsec Encapsulating Security Payload (ESP) Packet Format is shown in Figure 23. Each field in Figure 23 is explained as follows:

**Security Parameters Index (SPI)** is an arbitrary 32-bit value that uniquely identifies the Security Association for this datagram, in combination with the destination IP address and security protocol. The SPI field is mandatory.

**Sequence Number** is a 32-bit monotonically increasing counter value (sequence number). It is mandatory and is always present.

**Salt** field is also a 32-bit value that is assigned at the beginning of the security association, and then remains constant for the life of the security association. The salt should be unpredictable [10].

**ESP Payload Data** is comprised of a 64-bit initialization vector (IV of ESP), followed by the data area, along with an authentication tag T (also known as ICV) associated with the payload data.

The document in [10] clearly explains how to use AES-GCM as an IPsec ESP mechanism to provide confidentiality and data origin authentication. More specifically, the additional authentication data (AAD of Figure 23) in AES-GCM includes the SPI and Sequence Number fields; the IV of GCM includes Salt and IV of ESP fields.

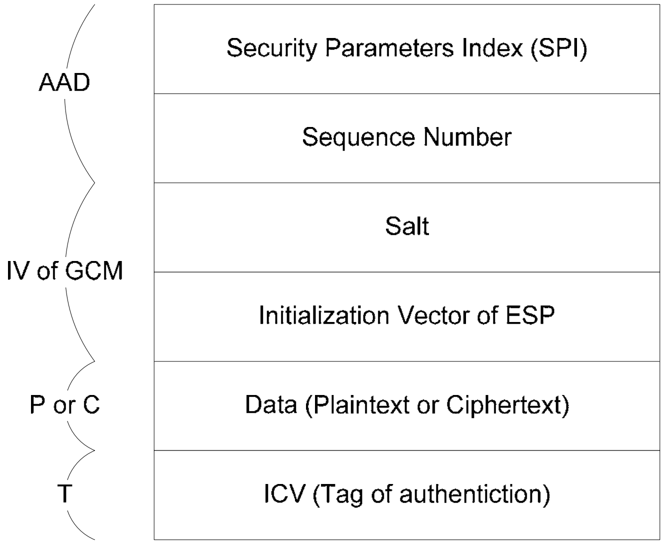

AAD
IV of GCM
P or C
T
Security Parameters Index (SPI)
Sequence Number
Salt
Initialization Vector of ESP
Data (Plaintext or Ciphertext)
ICV (Tag of authentiction)

**Figure 23. Format of Data Packet of IPsec ESP**

Information with respect to the format of data packet of IPsec ESP is provided in RFC2406[9] and RFC4106[10]. The Use of Galois-Counter Mode in IPsec ESP is shown in Figure 24.

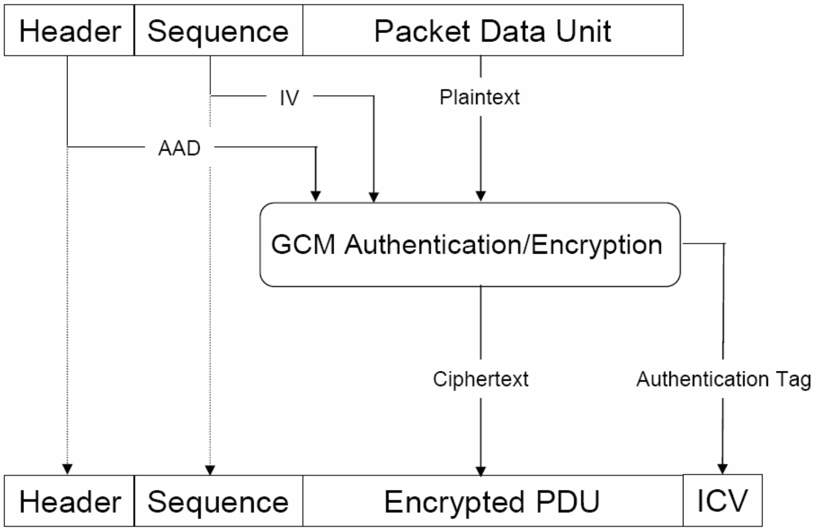


**Figure 24. The Use of GCM in IPsec ESP[11]**

### 4.2.2 Data Flow in GCM

If the AES module is implemented in the pipelined architecture, the GHASH module is implemented by choosing a parallel-bit multiplier as its core, and the hash subkey H can be calculated out ahead in an iterative AES module based on a known key by each communication party. The data flow in GCM Encryption is shown in Figure 25(a); and the data flow in GCM Decryption is shown in Figure 25(b). For GCM encryption, AES-GCM starts to compute intermediate hash value $Y_i$ when it receives additional authenticated data. It takes m clock cycles to generate $Y_m$. Then the GHASH has to be idle for 11 clock cycles until the first ciphertext block $C_1$ is generated by the GCTR which is created by using a pipelined AES module. For GCM with default IV, the IV is always 96 bits long, and $J_0$ can be created instantly by concatenation of bit strings.

The key streams for GCM encryption are created after the 10th clock cycle when $J_0$ is input into the pipeline of GCTR. At the 11th clock cycle, cipher block $C_1$ is generated and input to GHASH. GHASH begins to hash data again. At the m+11+n+1 clock cycle, $Y_{m+n+1}$ is generated and XORed with $K_0$ (i.e. $CIPH_K(J_0)$) to create authentication tag T.

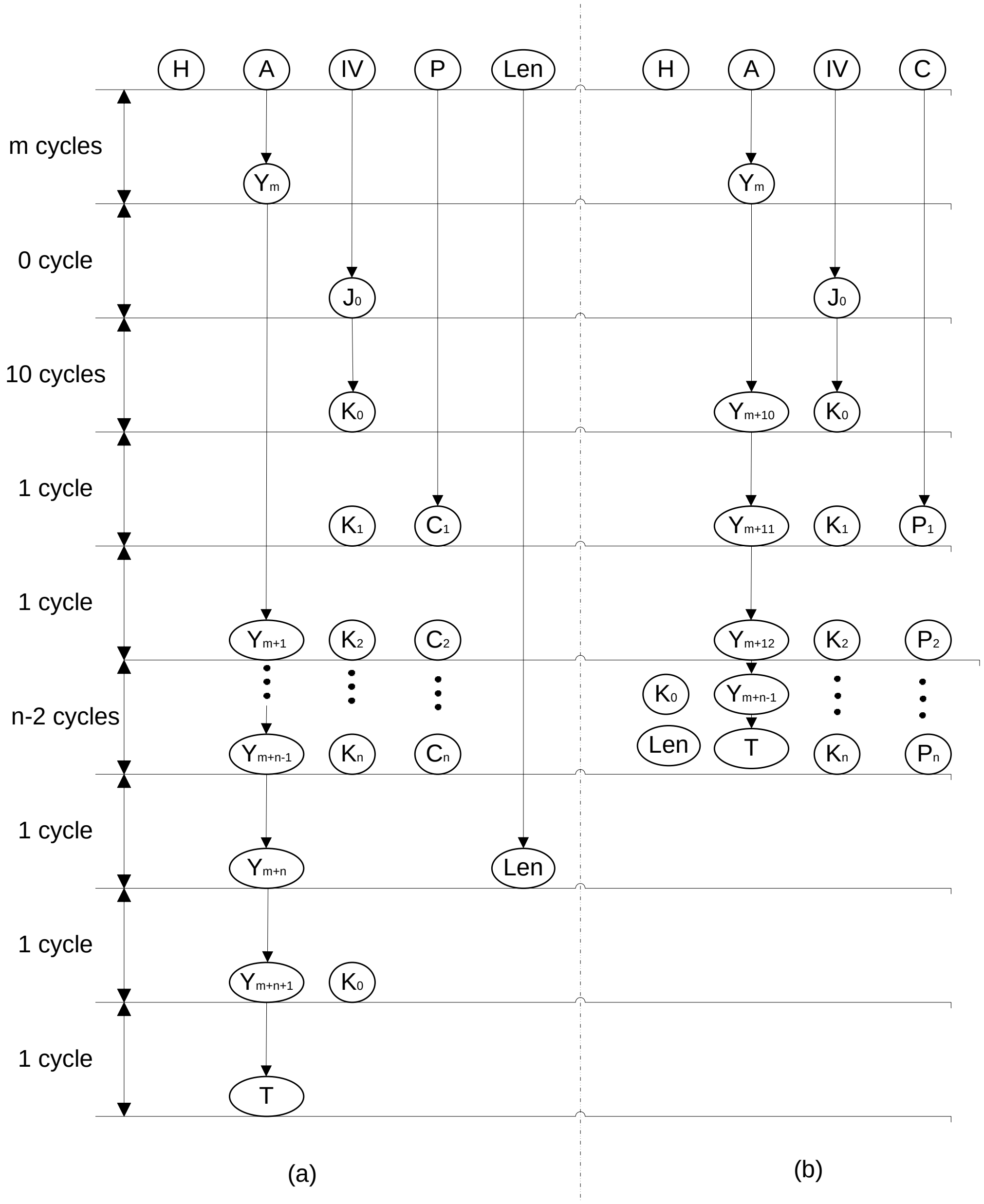


**Figure 25. (a) The Data Flow of GCM Encryption (b) The Data Flow of GCM Decryption**

For GCM decryption, GHASH can directly compute the authentication tag T’ based on AAD and ciphertext C from the input of GCM Decryption. Therefore, the max 11 clock cycles are saved compared with data flow in GCM Encryption.

### 4.2.3 Hardware Implementation Bidirectional GCM

Based on the data flow analysis in section 4.2.2, a bidirectional AES-GCM hardware module is built. The "bidirectional" means: the AES-GCM module can work not only as GCM encryption but also as GCM decryption depending on the logic value of the control signal Encryption. If the Encryption signal is high, then AES-GCM works in GCM encryption mode as shown in Figure 26, otherwise, it works in GCM decryption mode.

If the Encryption signal is high, then AES-GCM works in GCM encryption mode, i.e. AES-GCM-AE (see section 3.2.6.1). In Figure 26, the data paths are 128-bit wide. The control signals do not show up except signal Encryption. They all are driven by a finite sate machine (FSM) module which is designed according to IPsec ESP packet format. The 44 32-bit round-key words are stored in a look up table instead of generated in real time. The hash subkey H is generated by an iterative AES module from the key K in advance.

A 3-to-1 multiplexer MUX-I is used whose output connects to one of the input ports of XOR gates in GHASH module. The three inputs of MUX-I are additional authenticated data AAD, ciphertext C and length information length (A)||length(P). As discussed in section 3.2, in the first m clock cycles, the output of MUX-I is the additional authenticated data A. After 11 clock cycles, in the next n clock cycles, the output of MUX-I switches to the ciphertext C. The Final output of MUX-I is length (A)||length(P). The first 128-bit key stream which is produced by GCTR, from the initial value IV of GCM, is stored in the $AES_K(J_0)$ Register. This $AES_K(J_0)$ Register is later used to generate the authentication tag T. Since the IV of GCM is followed by plaintext P, and the first 128-bit keystream is generated by GCTR after a delay of 10 clock cycles. Therefore, the plaintext P is delayed by 11 clock cycles in order to be encrypted by the corresponding key streams. A 11* 128-bit FIFO meets this requirement. In the first 11 clock cycles, the data flow AAD, IV and payload data P are input to the FIFO. From the 12th clock cycle onwards, the FIFO remains in a dynamic full status by reading data out and writing new data in simultaneously until reaching the end of the IPsec ESP packet. Suffering 11 clock cycles delay through the FIFO, AAD and IV connect directly to one of the inputs of the 3-to-1 MUX-II; delayed payload data P exclusive-ORs with GCTR output, key stream, to produce ciphertext which is connected to one input of MUX-I and MUX-II. The left input of MUX-II is the authentication tag T which is the result of GHASH final output $Y_{m+n+1}$ XORing value in $AES_K(J_0)$ Register. MUX-II output connects to register Output. The final output of AES-GCM-AE from register Output is data flow A_IV_C_T corresponding to the input data flow A_IV_P.

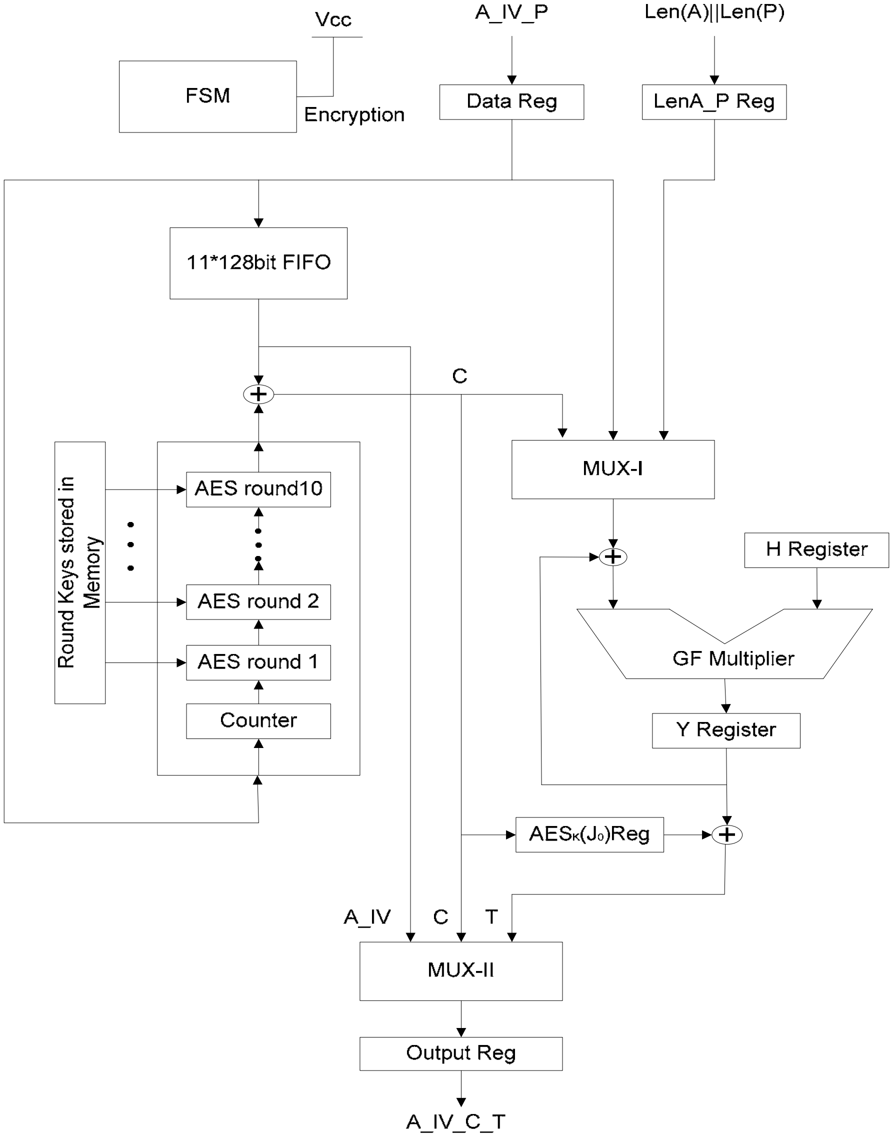


**Figure 26. AES-GCM Encryption Architecture**

As mentioned in section 4.1.2, the critical path of this design is determined by the GHASH module. The delay of all other paths in Figure 26 is smaller than the delay produced by GHASH module.

If Encryption is low, then AES-GCM works as GCM decryption. i.e. AES-GCM-AD (see section 3.2.6.2). AES-GCM-AD is similar to AES-GCM-AE. Compared with Figure 26, one difference is that two 2-to-1 multiplexers, also named MUX-I and MUX-II are used instead of two 3-to-1 multiplexers in the Figure 26. The reason that 2-to-1 multiplexers are used is that the authentication tag *T'* is computed directly from A and C of the original input A_IV_C_T and it does not need to be input into register Output either. Another difference is that a 128-bit comparator is used to generate the FAIL signal depending on the comparison between T and T'. The delay of the comparator is 1 XOR gate plus 7 OR gates which is still smaller than the delay of the 128-bit bit parallel multiplier over GF($2^{128}$) which is 1 AND gate plus 7 XOR gates.

Like the S-box of AES module in section 4.1.1, the 11*128-bit FIFO can also be implemented by using dual-port Block SelectRAM+ or dual-port Distributed SelectRAM+. Therefore, AES-GCM can be implemented either by purely using CLBs or using CLBs and block RAM. Table 8 lists the performance and cost comparison of these designs. Table 1 shows that there are totally 44096 slices and 444 Blocks SelectRAM+ in Virtex-II pro xc2vp100. Therefore, for the scheme (1), 48.5% (21409/44096=0.485) slices are used to implement AES-GCM; for the scheme (2), 24.4% (10797/44096=0.244) slices and 36.9% (164/444=0.369) BRAM blocks are used.

**Table 8. AES-GCM Implementations on Block RAM and Distributed RAM**

| | Scheme(1) Purely using CLBs | | | Scheme(2) Using CLBs + BRAMs | | |
|---|---|---|---|---|---|---|
| | Clock Rate | Throughput | Area Cost | Clock Rate | Throughput | Area Cost |
| AES-GCM | 75.7MHz* | 9.68 Gbps | 21409slices | 75.8 MHz* | 9.70 Gbps | 10797slices<br>+164BRAMs |

Note *: Maximum clock frequency from simulation results

## 4.3 Verification of AES-GCM Functionality

This section describes how the modules were verified in the realistic environment CMC-prototype-platform. All of them including AES, GHASH, AES-GCM-AE, and AES-GCM-AD were verified on this platform. They were also although designed in VHDL and timing simulated using Modelsim, respectively. The results are compared with other researches on hardware implementations of AES-GCM.

### 4.3.1 IPsec Signal Generator

In order to perform verification, an IPsec ESP signal Generator had to be built based on the IPsec ESP data packet format discussed in section 4.2.1. Figure 27 shows a 16-bit LFSR which generates $2^{16}$-1 bit stream sequence periodically based on a primitive polynomial f(x)= $1+x+x^3+x^{12}+X^{16}$ for building the IPsec signal generator which consisted of 8 16-bit LFSRs. The primitive polynomial with degree 16 was chosen since the maximum length of payload data of IPsec data packet is $2^{16}$ bit long. At the beginning, the control signal start_LFSR asserts for m clock cycles, the signal generator generates m blocks of parallel 128-bit data as AAD; at the next clock cycle, start_LFSR desserts for generating IV-GCM; sequentially, start_LFSR asserts again for n clock cycles in order to generate n blocks of parallel 128-bit data as payload data P. The values of m and n are controlled by one input of the signal generator, in other words, it is adjustable to meet the test requirement.

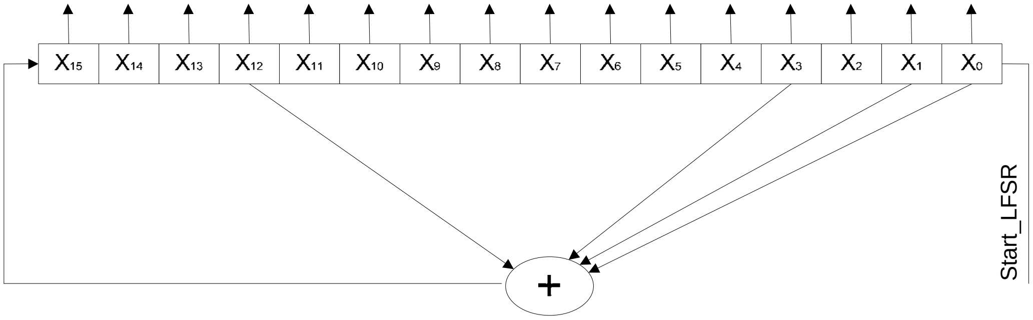


**Figure 27. 16-bit LFSR for IPsec ESP Signal Generator**

### 4.3.2 Verifying Both AES-GCM-AE and AES-GCM-AD on FPGA

In Figure 28, two AES-GCM modules are used, one working as AES-GCM-AE by connecting Encryption to the power, one working as AES-GCM-AD by connecting Encryption to the ground. The mimic IPsec data packets A_IV_P from the signal generator go through the AES-GCM-AE and the AES-GCM-AD consecutively, and then go to the comparison module in which there is another identical IPsec signal generator for checking the recovered data P validity. If each node in Figure 28 works correctly, then the plaintext P will be recovered from the AES-GCM-AD without any bit-errors, the signal Verifying_GCM will go to high to indicate the AES-GCM-AE and the AES-GCM-AD have been verified successfully. The comparison between T and T' is handled in the AES-GCM-AD module. If T is not equal to T', then the signal T_Verification (corresponding the output Fail in algorithm 6) will be set to high to indicate that the inputs are not authentic. The signal Verifying_GCM and T_Verification physically are connected to two LEDs on the CMC-prototype-platform in order to observe the verification results.

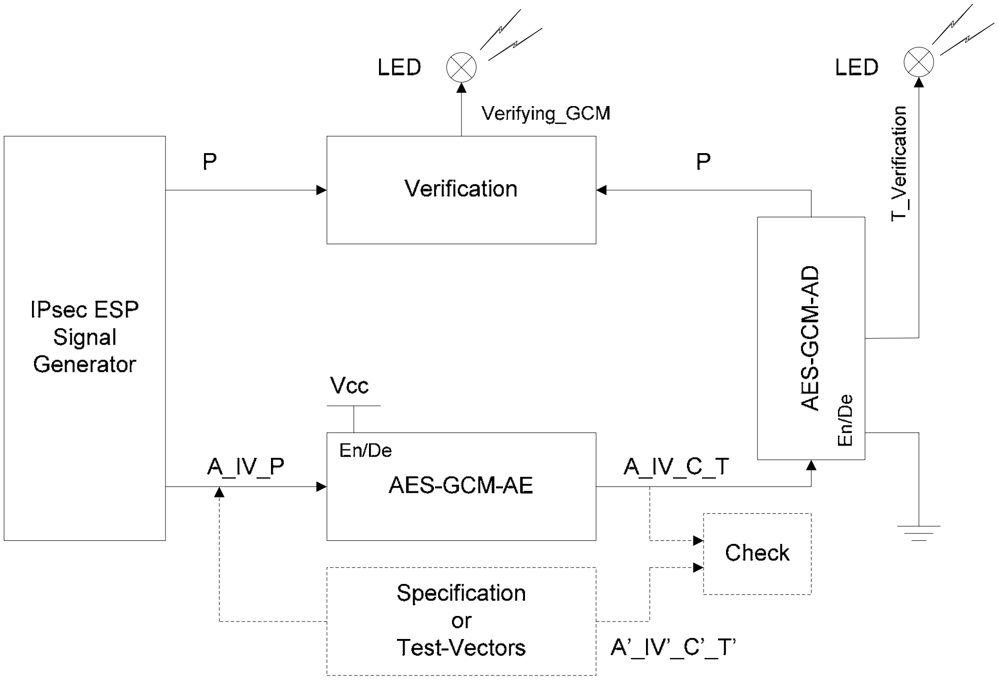


**Figure 28. AES-GCM Verification System**

After choosing 40MHz clock input as the global clock of FPGA, downloading the bitstream of the described architecture in Figure 28 to the CMC-prototype-platform, and configuring the Virtex-II pro xc2vp100, the LED Verifying_GCM turns on and LED T_Verification remains off. Hence the module with AES-GCM functionality is implemented successfully on FPGA platform.

In addition, in the appendix B of [12], the designers of GCM provides several cases of test vectors for testing AES-GCM implementation designs with different AES key sizes. The Test Case 3 and Test Case 4 are chosen to verify the work in this thesis. More specifically, first, using the 128-bit secret key K provided in Test Case 3 or Test Case 4 generates not only 44 32-bit expanded key words for AES round-transformations but also hash subkey H for GHASH hash operations; second, using the additional authentication data A, the initial vector IV, and the plaintext data P provided in Test Case 3 or Test Case 4 as parameters builds a test-bench which works as a stimulus to output data flow A_IV_P into AES-GCM module for timing simulation; Finally, comparing the results A_IV_C_T of the timing simulation of

AES-GCM with the A'_IV'_C'_T' provided in Test Case 3 and Test Case 4 and make sure they are identical (see the dash-line part of Figure 28).

All the VHDL codes for generating AES-GCM, test benches, and test vectors are printed out and listed in Appendixes. The hierarchical HDL code designs are shown in Figure 29.

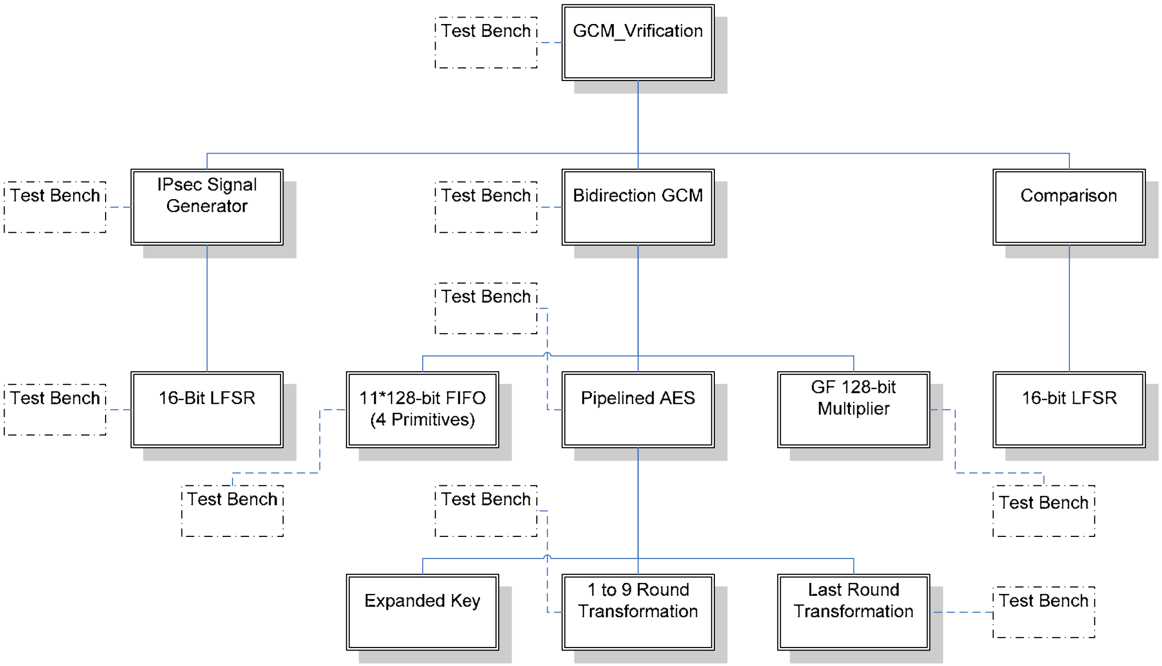


**Figure 29. AES-GCM Hierarchical HDL Codes Design**

## 4.4 Comparison with Other Research

Research on hardware architectures or implementations of GCM is fairly small. This is likely due to the new mode of operation based on symmetric block cipher. However, some precious research involving GCM Encryption and Decryption based on AES is given in Table 9 for a comparison. In Table 9, except the design (3) implemented in FPGA, i.e. the work in this thesis, both design (1) and (2) complete AES-GCM implementations on CMOS technology.

The design (1) is a commercialized IP core named CLP-24 which is built by the company Elliptic Semiconductor Inc. CLP-24 supports for three key sizes – 256, 192, or 128 bit keys. Its input data bit-width is configurable.

**Table 9. AES-GCM Implementation Results**

| GCM En/Decryption Designs | Technology | Areas | Clock Rates | Throughputs | Time |
|---|---|---|---|---|---|
| (1) Elliptic Semiconductor Inc. [11] | ASIC | 97k gates | 300 MHz | 7 Gbps | 2006 |
| (2) Bo Yang, Sambit Mishra, et al. [20] | ASIC | 498k gates | 271 MHz * | 34.7 Gbps | 2005 |
| (3) Work in this thesis | FPGA | 21409 slices | 76 MHz * | 9.7 Gbps | 2006 |

Notes *: based on simulation results.

The design (2) demonstrates AES-GCM architecture which is similar to the architecture discussed in this thesis, using a 0.18 um CMOS standard cell library. In design (2), the proposed architecture is modeled in Verilog HDL and simulated using Modelsim. The Verilog models were synthesized using Synopsys Design Compiler. The cost and clock rate were reported after the netlist is generated by Synopsys Design Compiler and placed and routed by Cadence Silicon Ensemble.

Comparing design (1) and design (2), the area cost of design (1) is one-fifth of that of design (2); the throughput of design (1) is one-fifth of that of design (2) also. Design (2) is a 128-bit bus width architecture, thus, design (1) is likely an architecture based on inner 32-bit bus width.

In [23], authors claim that the ratio of critical path delay, from FPGA to ASIC is roughly 3 to 4, with less influence from block memory and hard multipliers. Comparing ASIC design

(2) and FPGA design (3) because both of them have similar 128-bit bus width AES-GCM architectures, the ratio is 3.6 (271MHz/76MHz = 3.6) which confirms this result.

# Chapter 5
# Concluding Remarks

## 5.1 Summary and Contributions

In this thesis, the development of a complete architecture for the AES-GCM security standard has been discussed. The AES-GCM takes advantage of the new block cipher algorithm AES and hash function GHASH which is based on the multiplication algorithm over finite field GF ($2^{128}$). Field addition and multiplication are represented with respect to the polynomial basis while the non-feedback security operation modes ECB and CTR are introduced for they are the fundamental of the confidentiality of GCM. AES implementation schemes are discussed in both iterative and pipelined architectures while GHASH implementation schemes are discussed also in both Serial-bit and Parallel-bit methods. The AES-GCM architecture was designed to support IPsec ESP application or others. Compared with previous researches which were only implemented in ASICs, this FPGA architecture is robust and achieves a good throughput from the pipelining design. The feasibility of the AES-GCM architecture has been verified through a prototype implementation on the CMC-Prototype FPGA platform which worked in the Stand-alone mode (see section 2.4).

The Contributions achieved in this work are as follows:

- This is the first time that the author is aware of an implementation of the AES-GCM security standard has been performed in a FPGA platform.
- AES-GCM module can work in bidirection, either GCM encryption or GCM decryption.
- AES-GCM module can be implemented by using either block selectRAM+ or distributed selectRAM+.

## 5.2 Future Work

Although the proposed AES-GCM modules were optimized by following approaches:

- Remove Reset signal from data flow,
- Replicate registers,
- Flat the hierarchical code structure,
- Use parentheses on arithmetic equations to suggest parallelism,
- Perform arithmetic on the minimum length of bits needed to reduce operation strength,
- Pipeline architecture,
- Increase fan-out ability,
- Insert buffer,

There may still be improvement possible with respect to the performance based on following issues:

- **Bad Packing**, bad packing means two LUTs that are packed together and nets go to different locations.
- **Bad Placement**, bad placement means logic placed far away from each other.
- **Poor IO Timing**, poor IO timing means an IO signal uses a flip-flop in CLB instead of the one in IOB.
- **High Fanout Net**, high fanout net meant a net connecting to too many different loads in the FPGA chip.
- **Too many levels of logic**, this kind of circuit is usually critical path (see Appendix F).

Furthermore, AES-GCM may also work as an IP cores in the Base-line mode as mentioned in section 2.4. Section 4.2.3 shows that AES-GCM design implemented in this thesis occupies 48.5% or 24.4% resources of the FPGA without or with using BRAM. Since approximate 93% FPGA resources are free in the Base-line mode, the AES-GCM IP core can hang on the bus PLB inside the FPGA to encrypt and authenticate the data buffered in DDR-SDRAM. Figure 30 illustrates how the AES-GCM IP core can work in the Base-line mode. Data in DDR-SDRAM go through the 64-bit 80MHz PLB bus to the 128-bit bandwidth AES-GCM module which works in 40MHz to be authenticated and encrypted, then output to the MGT IP cores (Multi-Gigabit Transciever) to be sent out from the high speed serial ports. Or, the authenticated and encrypted input data can be received from the MTG, then verified and decrypted by AES-GCM and buffered in DDR-SDRAM through PLB bus.

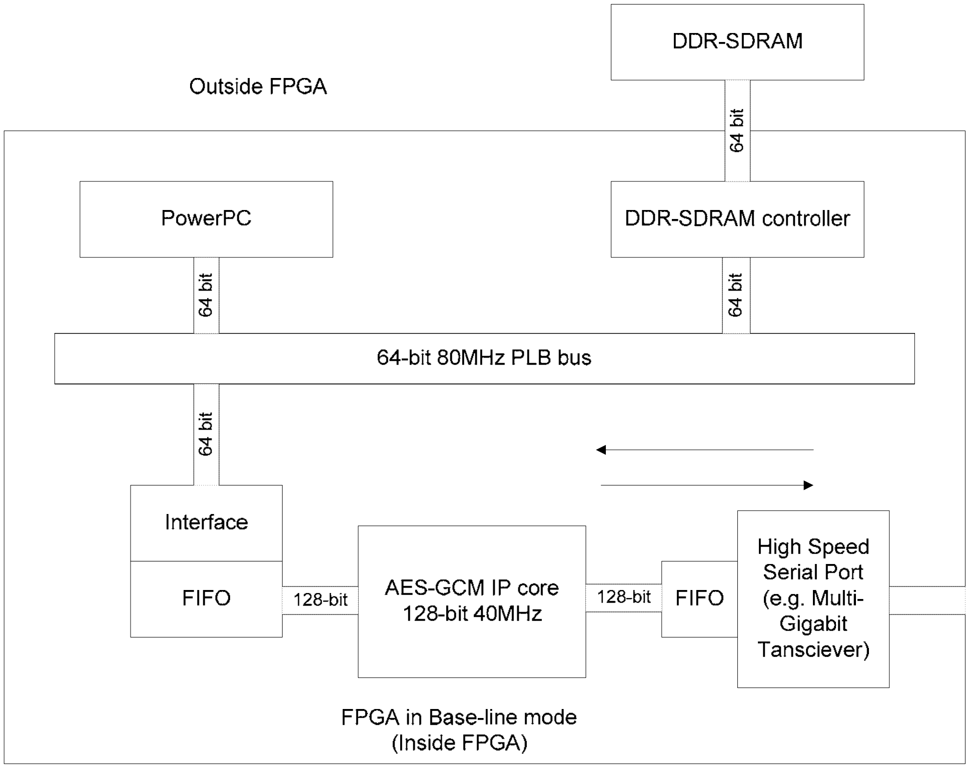


**Figure 30. Illustration How AES-GCM Works in Base-line Mode**

# Appendix A

## Test-Vectors for AES-GCM [12]

```
GCM Test Case #03 (AES-128)
Variable          Value
-------------------------------------------------
K               : feffe9928665731c6d6a8f9467308308
P               : d9313225f88406e5a55909c5aff5269a
                : 86a7a9531534f7da2e4c303d8a318a72
                : 1c3c0c95956809532fcf0e2449a6b525
                : b16aedf5aa0de657ba637b391aafd255
IV              : cafebabefacedbaddecaf888
H               : b83b533708bf535d0aa6e52980d53b78
Y_0             : cafebabefacedbaddecaf88800000001
E(K,Y_0)        : 3247184b3c4f69a44dbcd22887bbb418
Y_1             : cafebabefacedbaddecaf88800000002
E(K,Y_1)        : 9bb22ce7d9f372c1ee2b28722b25f206
Y_2             : cafebabefacedbaddecaf88800000003
E(K,Y_2)        : 650d887c3936533a1b8d4e1ea39d2b5c
Y_3             : cafebabefacedbaddecaf88800000004
E(K,Y_3)        : 3de91827c10e9a4f5240647ee5221f20
Y_4             : cafebabefacedbaddecaf88800000005
E(K,Y_4)        : aac9e6ccc0074ac0873b9ba85d908bd0
X_1             : 59ed3f2bb1a0aaa07c9f56c6a504647b
X_2             : b714c9048389afd9f9bc5c1d4378e052
X_3             : 47400c6577b1ee8d8f40b2721e86ff10
X_4             : 4796cf49464704b5dd91f159bb1b7f95
len(A)||len(C)  : 00000000000000000000000000000200
GHASH(H,A,C)    : 7f1b32b81b820d02614f8895ac1d4eac
C               : 42831ec2217774244b7221b784d0d49c
                : e3aa212f2c02a4e035c17e2329aca12e
                : 21d514b25466931c7d8f6a5aac84aa05
                : 1ba30b396a0aac973d58e091473f5985
T               : 4d5c2af327cd64a62cf35abd2ba6fab4
```

```
GCM Test Case #04 (AES-128)
Variable          Value
-------------------------------------------------
K               : feffe9928665731c6d6a8f9467308308
P               : d9313225f88406e5a55909c5aff5269a
                : 86a7a9531534f7da2e4c303d8a318a72
                : 1c3c0c95956809532fcf0e2449a6b525
                : b16aedf5aa0de657ba637b39
A               : feedfacedeadbeeffeedfacedeadbeef
                : abaddad2
IV              : cafebabefacedbaddecaf888
H               : b83b533708bf535d0aa6e52980d53b78
Y_0             : cafebabefacedbaddecaf88800000001
E(K,Y_0)        : 3247184b3c4f69a44dbcd22887bbb418
X_1             : ed56aaf8a72d67049fdb9228edba1322
X_2             : cd47221ccef0554ee4bb044c88150352
Y_1             : cafebabefacedbaddecaf88800000002
E(K,Y_1)        : 9bb22ce7d9f372c1ee2b28722b25f206
Y_2             : cafebabefacedbaddecaf88800000003
E(K,Y_2)        : 650d887c3936533a1b8d4e1ea39d2b5c
Y_3             : cafebabefacedbaddecaf88800000004
E(K,Y_3)        : 3de91827c10e9a4f5240647ee5221f20
Y_4             : cafebabefacedbaddecaf88800000005
E(K,Y_4)        : aac9e6ccc0074ac0873b9ba85d908bd0
X_3             : 54f5e1b2b5a8f9525c23924751a3ca51
X_4             : 324f585c6ffc1359ab371565d6c45f93
X_5             : ca7dd446af4aa70cc3c0cd5abba6aa1c
X_6             : 1590df9b2eb6768289e57d56274c8570
len(A)||len(C)  : 00000000000000a000000000000001e0
GHASH(H,A,C)    : 698e57f70e6ecc7fd9463b7260a9ae5f
C               : 42831ec2217774244b7221b784d0d49c
                : e3aa212f2c02a4e035c17e2329aca12e
                : 21d514b25466931c7d8f6a5aac84aa05
                : 1ba30b396a0aac973d58e091
T               : 5bc94fbc3221a5db94fae95ae7121a47
```

# Appendix B

# Package for AES-GCM VHDL Codes

```
--===========================
DATATYPE.VHD
--===========================
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package datatypes is                                                    ----Package definition
type key_arr is array (0 to Nr) of std_logic_vector(0 to 127);
type RCbox is array (0 to 10) of std_logic_vector (7 downto 0);
type MatrixU is array (0 to 127) of std_logic_vector(0 to 127);
type shiftN_B is array (0 to 3) of integer range 0 to 3;
type rom_type is array (0 to 255) of std_logic_vector (7 downto 0);
constant Nb: integer:=4;
constant Nk: integer:=4;
constant Nr: integer:=10;
constant R: std_logic_vector(0 to 7):=B"11100001";      --"11100001 + 0000.....00"
constant  shiftNb: shiftN_B:=(0, 3, 2, 1);
constant ROM: rom_type :=(
--0
 X"63", X"7c", X"77", X"7b",  X"f2", X"6b", X"6f", X"c5",  X"30", X"01", X"67", X"2b", X"fe", X"d7", X"ab", X"76",
--1
 X"ca", X"82", X"c9", X"7d",  X"fa", X"59", X"47", X"f0",  X"ad", X"d4", X"a2", X"af", X"9c", X"a4", X"72", X"c0",
--2
 X"b7", X"fd", X"93", X"26",  X"36", X"3f", X"f7", X"cc",  X"34", X"a5", X"e5", X"f1", X"71", X"d8", X"31", X"15",
--3
 X"04", X"c7", X"23", X"c3",  X"18", X"96", X"05", X"9a",  X"07", X"12", X"80", X"e2", X"eb", X"27", X"b2", X"75",
--4
 X"09", X"83", X"2c", X"1a",  X"1b", X"6e", X"5a", X"a0",  X"52", X"3b", X"d6", X"b3", X"29", X"e3", X"2f", X"84",
--5
 X"53", X"d1", X"00", X"ed",  X"20", X"fc", X"b1", X"5b",  X"6a", X"cb", X"be", X"39", X"4a", X"4c", X"58", X"cf",
--6
 X"d0", X"ef", X"aa", X"fb",  X"43", X"4d", X"33", X"85",  X"45", X"f9", X"02", X"7f", X"50", X"3c", X"9f", X"a8",
--7
 X"51", X"a3", X"40", X"8f",  X"92", X"9d", X"38", X"f5",  X"bc", X"b6", X"da", X"21", X"10", X"ff", X"f3", X"d2",
--8
 X"cd", X"0c", X"13", X"ec",  X"5f", X"97", X"44", X"17",  X"c4", X"a7", X"7e", X"3d", X"64", X"5d", X"19", X"73",
--9
 X"60", X"81", X"4f", X"dc",  X"22", X"2a", X"90", X"88",  X"46", X"ee", X"b8", X"14", X"de", X"5e", X"0b", X"db",
--10
 X"e0", X"32", X"3a", X"0a",  X"49", X"06", X"24", X"5c",  X"c2", X"d3", X"ac", X"62", X"91", X"95", X"e4", X"79",
--11
 X"e7", X"c8", X"37", X"6d",  X"8d", X"d5", X"4e", X"a9",  X"6c", X"56", X"f4", X"ea", X"65", X"7a", X"ae", X"08",
--12
 X"ba", X"78", X"25", X"2e",  X"1c", X"a6", X"b4", X"c6",  X"e8", X"dd", X"74", X"1f", X"4b", X"bd", X"8b", X"8a",
--13
 X"70", X"3e", X"b5", X"66",  X"48", X"03", X"f6", X"0e",  X"61", X"35", X"57", X"b9", X"86", X"c1", X"1d", X"9e",
--14
 X"e1", X"f8", X"98", X"11",  X"69", X"d9", X"8e", X"94",  X"9b", X"1e", X"87", X"e9", X"ce", X"55", X"28", X"df",
--15
 X"8c", X"a1", X"89", X"0d",  X"bf", X"e6", X"42", X"68",  X"41", X"99", X"2d", X"0f", X"b0", X"54", X"bb", X"16"
);

--=================================================================
constant RoundKeySet: Key_arr:= (           ---for GCM Test-Case 4 only
                                            (X"feffe9928665731c6d6a8f9467308308"),   --for Round 0
                                            (X"fb13d9177d76aa0b101c259f772ca697"),  --for Round 1
                                            (X"883751e2f541fbe9e55dde76927178e1"),
                                            (X"2f8ba9addaca52443f978c32ade6f4d3"),
                                            (X"a934cf3873fe9d7c4c69114ee18fe59d"),
                                            (X"caed91c0b9130cbcf57a1df214f5f86f"),
                                            (X"0cac393ab5bf358640c528745430d01b"),
                                            (X"48dc961afd63a39cbda68be8e9965bf3"),
                                            (X"58e59b04a58638981820b370f1b6e883"),
                                            (X"0d7e77a5a8f84f3db0d8fc4d416e14ce"),
                                            (X"a484fc260c7cb31bbca44f56fdca5b98")
                                  );


--constant RoundKeySet: Key_arr:= (         ---for AES spec only
--                                          (X"2b7e151628aed2a6abf7158809cf4f3c"),
--                                          (X"a0fafe1788542cb123a339392a6c7605"),
--                                          (X"f2c295f27a96b9435935807a7359f67f"),
--                                          (X"3D80477D4716FE3E1E237E446D7A883B"),
--                                          (X"ef44a541a8525b7fb671253bdb0bad00"),
--                                          (X"d4d1c6f87c839d87caf2b8bc11f915bc"),
--                                          (X"6d88a37a110b3efddbf98641ca0093fd"),
--                              (X"4e54f70e5f5fc9f384a64fb24ea6dc4f"),
--                              (X"ead27321b58dbad2312bf5607f8d292f"),
--                              (X"ac7766f319fadc2128d12941575c006e"),
--                              (X"d014f9a8c9ee2589e13f0cc8b6630ca6")
--                              );

--==================================================================
constant Rcon: RCbox:= (
(X"00", X"01", X"02", X"04",
 X"08", X"10", X"20", X"40",
 X"80", X"1b", X"36")
);
--===================================================================
function multiply_by_02(multiplicand: in std_logic_vector(7 downto 0)) return std_logic_vector;
function multiply_by_03(multiplicand: in std_logic_vector(7 downto 0)) return std_logic_vector;
function rightshift(X: in std_logic_vector(0 to 127)) return std_logic_vector;
end datatypes;

package body datatypes is

function multiply_by_02(multiplicand: in std_logic_vector(7 downto 0)) return std_logic_vector is
variable temp: std_logic_vector(7 downto 0);
begin
temp:=multiplicand;
if (temp(7)='1') then
temp:=(temp(6 downto 0)&'0') xor X"1b";               --"temp" for loading multiply by 02;
else
temp:=(temp(6 downto 0)&'0');                         --"temp" for loading multiply by 02;
end if;
return temp;
end function multiply_by_02;

function multiply_by_03(multiplicand: in std_logic_vector(7 downto 0)) return std_logic_vector is
variable temp: std_logic_vector(7 downto 0);
begin
temp:=multiplicand;
if (temp(7)='1') then
temp:=(temp(6 downto 0)&'0') xor X"1b";                        --"temp" for loading multiply by 02;
else
temp:=(temp(6 downto 0)&'0');                                  --"temp" for loading multiply by 02;
```

```
end if;
temp:=multiplicand xor temp;                                                    --after *2, do addition (+1)
return temp;
end function multiply_by_03;

function rightshift(X: in std_logic_vector(0 to 127)) return std_logic_vector is
variable temp: std_logic_vector(0 to 127);
begin
for i in 0 to 126 loop
temp(i+1) := X(i);
end loop;
temp(0) := '0';
return temp;
end function rightshift;

end package body datatypes;                                ----end of Package definition
```

# Appendix C
# VHDL Codes for AES-GCM modules

Notes:

1. Even some modules are seemingly “Behavioral” architectures, but they are actually “Structural” architectures.
2. Refer to Figure 29. AES-GCM Hierarchical HDL Codes Design.
3. All these codes were synthesized, simulated and ran based on the tools as follows:

   a) Xilinx ISE7.0i

   b) Monter Graphics ModelSim SE 6.1a

   c) Xilinx FPGA Virtex-II pro xc2vp100

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    GCM_Verification - Behavioral
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity GCM_Verification is
port(
                    clk: in std_logic;
                    Reset_n: in std_logic;
                    --A_P_Len: std_logic_vector(0 to 31);
                    LFSR_16bit_out_prime, P_16bit_out: out std_logic_vector(0 to 15);
                    Sec_LED: out std_logic;
                    T_Verification: out std_logic;
                    Verifying_GCM: out std_logic;
                    floating: out std_logic;
                    clk_out: out std_logic;
                    data_valid: out std_logic
        );
end GCM_Verification;
```

```
architecture Behavioral of GCM_Verification is

component LFSR_16bit
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_LFSR: in std_logic;
                    LFSR_IV: in std_logic_vector(0 to 15);
                    LFSR_parallel_out: out std_logic_vector(0 to 15)
          );
end component;

component IPsec_Signal_Source
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_GCM: out std_logic;
                    H: out std_logic_vector(0 to 127);
                    A_IV_P: out std_logic_vector(0 to 127);
                    A_P_Len: in std_logic_vector(31 downto 0)
          );
end component;

component GCM
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_GCM: in std_logic;
                    Encryption: in std_logic;
                    H: in std_logic_vector(0 to 127);
                    A_IV_P: in std_logic_vector(0 to 127);
                    A_P_Len: in std_logic_vector(31 downto 0);
                    A_IV_C_T: out std_logic_vector(0 to 127);
                    T_Verification: out std_logic;
                    data_valid: out std_logic
          );
end component;

constant Low : std_logic :='0';
constant High : std_logic :='1';
```

```
signal start_GCM, start_GCM_decryption, start_compr, Verification, Sec_output, reset: std_logic;
signal H: std_logic_vector(0 to 127);
signal A_IV_P,A_IV_C_T, P, LFSR_128bit_prime: std_logic_vector(0 to 127);
signal A_P_Len: std_logic_vector(0 to 31);
signal LFSR_IV, LFSR_16bit_out: std_logic_vector(0 to 15);
signal Sec_counter: integer range 0 to 40000000;

begin

reset <= not Reset_n;
clk_out <= clk;
A_P_Len <=X"01800380";

Signal_Source: IPsec_Signal_Source
port map (
                clk=>clk,
                reset=>reset,
                start_GCM=>start_GCM,
                H=>H,
                A_IV_P=>A_IV_P,
                A_P_Len=>A_P_Len
        );

GCM_encryption: GCM
port map(
                clk=>clk,
                reset=>reset,
                start_GCM=>start_GCM,
                Encryption=>High,
                H=>H,
                A_IV_P=>A_IV_P,
                A_P_Len=>A_P_Len,
                A_IV_C_T=>A_IV_C_T,
                T_Verification=> floating,
                data_valid=>start_GCM_decryption
        );

GCM_decryption: GCM
port map(
                clk=>clk,
                reset=>reset,
                start_GCM=>start_GCM_decryption,
                Encryption=>Low,
                H=>H,
                A_IV_P=>A_IV_C_T,
                A_P_Len=>A_P_Len,
                A_IV_C_T=>P,
                T_Verification=>T_Verification,
                data_valid=>start_compr
        );

LFSR_16b_parallel_out: LFSR_16bit
port map(
                clk=>clk,
                reset=>reset,
                start_LFSR=>start_compr,
                LFSR_IV=>LFSR_IV,
                LFSR_parallel_out=>LFSR_16bit_out
        );

LFSR_128bit_prime <= LFSR_16bit_out & LFSR_16bit_out & LFSR_16bit_out & LFSR_16bit_out
                & LFSR_16bit_out &  LFSR_16bit_out & LFSR_16bit_out &  LFSR_16bit_out;
LFSR_16bit_out_prime <= LFSR_16bit_out;
P_16bit_out <= P(0 to 15);

process(clk)
begin
if clk'event and clk='1' then
if reset='1' or start_compr='1' then
if reset='1'then
Verification <='0';
else
if LFSR_128bit_prime = P(0 to 127) then
Verification <='0';
else
Verification <='1';
end if;
end if;
end if;
end if;
end process;

process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
Verifying_GCM <='0';
else
if Verification = '1' then
Verifying_GCM <='1';
end if;
end if;
end if;
end process;

process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
Sec_counter <=0;
Sec_output<='0';
else
if Sec_counter = 20000000 then    ---20000000
Sec_output <= not Sec_output;
Sec_counter <=0;
else
Sec_counter <= Sec_counter + 1;
end if;
end if;
end if;
end process;

data_valid <= start_compr;
Sec_LED <= Sec_output;

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    GCM - Behavioral
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.datatypes.all;

entity GCM is
        port(
                clk: in std_logic;
                reset: in std_logic;
                start_GCM: in std_logic;
                Encryption: in std_logic;
                H: in std_logic_vector(0 to 127);
                A_IV_P: in std_logic_vector(0 to 127);
                A_P_Len: in std_logic_vector(31 downto 0);
                A_IV_C_T: out std_logic_vector(0 to 127);
                T_Verification: out std_logic;
                data_valid: out std_logic
        );
end GCM;

architecture Behavioral of GCM is

component AES_Pipelined
        port (
                encrypt: in std_logic;
                clk: in std_logic;
                reset: in std_logic;
                plaintext: in std_logic_vector(0 to 127);
                ciphertext: out std_logic_vector(0 to 127)
                --encrypt_status: out std_logic
        );
end component;

component fifo_16x128b
```

```
        port (
                clk: IN std_logic;
                sinit: IN std_logic;
                din: IN std_logic_VECTOR(127 downto 0);
                wr_en: IN std_logic;
                rd_en: IN std_logic;
                dout: OUT std_logic_VECTOR(127 downto 0);
                full: OUT std_logic;
                empty: OUT std_logic
        );
end component;

component Parallel_Multiplied_by_H
  port(
                clk: in std_logic;
                reset: in std_logic;
                H: in std_logic_vector(0 to 127);
                AC: in std_logic_vector(0 to 127);
                X: out std_logic_vector(0 to 127)
        );
end component;

signal counter, time_for_calculate_TT_prime: integer range 0 to 2047;        --because of 1023+Fifo_delay+2;
signal C, C_delay, Mux2out, Mux3out, AC, X, FIFO_out: std_logic_vector(0 to 127);
signal EKY0, IV_counter, Key_stream, FIFO_out_delay, T, TT, TT_prime: std_logic_vector(0 to 127);
signal wr_en, rd_en, full, empty: std_logic;
signal H_2, A_IV_P_2: std_logic_vector(0 to 127);
signal A_P_Len_1, A_P_Len_2, A_P_Len_3, A_P_Len_4: std_logic_vector(31 downto 0);

signal Rem_128_A, Rem_128_P: integer range 0 to 127;
signal Num_128_A, Num_128_P: integer range 0 to 511;
signal FIFO_Delay: integer range 0 to 15;
signal Num_128_IV: integer range 0 to 1;
signal Reg_Delay: integer range 0 to 1;
signal Num_128_T: integer range 0 to 1;

begin

------Calculate Control parameter--------
```

```
Rem_128_A <= conv_integer(A_P_Len_1(22 downto 16));        -- indicate how many bits for last segment of A;
Rem_128_P <= conv_integer(A_P_Len_1(6 downto 0));          -- indicate how many bits for last segment of P;

process(A_P_Len_3, A_P_Len_4, Rem_128_A, Rem_128_P)
begin
if Rem_128_A =0 then
Num_128_A <= conv_integer(A_P_Len_3(31 downto 23));
else
Num_128_A <= (conv_integer(A_P_Len_3(31 downto 23)) + 1);
end if;
if Rem_128_P =0 then
Num_128_P <= conv_integer(A_P_Len_4(15 downto 7));
else
Num_128_p <= (conv_integer(A_P_Len_4(15 downto 7)) + 1);
end if;
end process;

FIFO_Delay <= 11;                        -- 11 cycles delay for deleted bulb between A_IV and C;
Num_128_IV <= 1;                         -- at here, We only choose IV is 96-bit long, so Num_128_IV = 1;
Reg_Delay <= 1;                          -- 1 cycles delay for deleted bulb between C and T;
Num_128_T <= 1;                          -- the number of segments of the 128-bit of T or the number of 128-bit of A_P_Len;

------------------------------------------

process(clk)
begin
if clk'event and clk='1' then
 if reset='0' and start_GCM='1' then

                if counter = 0 then
                        wr_en <= '1';
                end if;
                if counter =10 then
                        rd_en <='1';
                end if;
                if counter = (Num_128_A+Num_128_IV+Num_128_P) then
                        wr_en <= '0';
                end if;
                if counter = (FIFO_Delay+Num_128_A)+(Num_128_IV+Num_128_P) then
                        rd_en <= '0';
                end if;
                counter <= counter + 1;

 else
                counter <=0;
                wr_en <= '0';
                rd_en <= '0';
 end if;
end if;
end process;

process(clk)
begin
if clk'event and clk='1' then
if counter = 0 then
--      H_2 <= H;
        A_P_Len_1 <= A_P_Len;
        A_P_Len_2 <= A_P_Len;
        A_P_Len_3 <= A_P_Len;
        A_P_Len_4 <= A_P_Len;
end if;
end if;
end process;

A_IV_P_2 <= A_IV_P;

-----------------------------------------
--MUX_I:
-----------------------------------------
MUX2:process(A_P_Len_2, A_IV_P_2, C, counter, Encryption, Num_128_A, Num_128_IV, Num_128_P)
begin
if Encryption ='1' then
if 0 < counter and counter <= Num_128_A then
Mux2out <= A_IV_P_2;
elsif (FIFO_Delay+Num_128_A+Num_128_IV) < counter and counter <= (FIFO_Delay+Num_128_A)+(Num_128_IV+Num_128_P) then
Mux2out <= C;
else
Mux2out <= (X"000000000000")&A_P_Len_2(31 downto 16)&(X"000000000000")&A_P_Len_2(15 downto 0);
end if;
```

```
else

if 0 < counter and counter <= Num_128_A then
Mux2out <= A_IV_P_2;
elsif (Num_128_A+Num_128_IV < counter and counter <=Num_128_A+Num_128_IV+Num_128_P) then
Mux2out <= A_IV_P_2;
else
Mux2out <= (X"000000000000")&A_P_Len_2(31 downto 16)&(X"000000000000")&A_P_Len_2(15 downto 0);
end if;
end if;
end process;
----------------------------------------
MUX3:process(FIFO_out_delay, C_delay, T, counter, Encryption, Num_128_A, Num_128_IV, Num_128_P)

begin
if Encryption ='1' then
if (FIFO_Delay+Reg_Delay) < counter and counter <=(FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV) then
Mux3out <= FIFO_out_delay;
elsif (FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV) < counter
          and counter <= (FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV)+Num_128_P then
Mux3out <= C_delay;
else
Mux3out <= T;
end if;
else
if (FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV) < counter
          and counter <= (FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV)+Num_128_P then
Mux3out <= C_delay;
else
Mux3out <= FIFO_out_delay;
end if;
end if;
end process;
----------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
AC <=(others => '0');   --in order to keep 'X' initial value is 0, we should assign AC to 0;
else
```

```
if Encryption ='1' then
if (0 < counter and counter <= Num_128_A) or ((FIFO_Delay+Num_128_A+Num_128_IV) < counter
        and counter <= (FIFO_Delay+Num_128_A)+(Num_128_IV+Num_128_P)+Num_128_T) then
AC <= Mux2out xor X;
end if;
else
if (0 < counter and counter <= Num_128_A) or (Num_128_A+Num_128_IV < counter
        and counter <= (Num_128_A+Num_128_IV)+(Num_128_P+Num_128_T)) then
AC <= Mux2out xor X;
end if;
end if;
end if;
end if;
end process;
-----------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if counter=(Num_128_A+Num_128_IV) then
IV_counter <= A_IV_P_2;
elsif counter>(Num_128_A+Num_128_IV) then
IV_counter(119 to 127)<=IV_counter(119 to 127)+1;
end if;
end if;
end process;
-----------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if counter=(FIFO_Delay+Num_128_A+Num_128_IV) then
EKY0 <= Key_stream;
end if;
end if;
end process;
------------------------------------------
process(clk)
begin
if clk'event and clk='1' then
C_delay <= C;
end if;
end process;
------------------------------------------
process(clk)
begin
if clk'event and clk='1' then
FIFO_out_delay <= FIFO_out;
end if;
end process;
------------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if Encryption ='1' then
if (FIFO_Delay+Reg_Delay) < counter
        and counter <=(FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV)+(Num_128_P+Num_128_T) then

A_IV_C_T <= Mux3out;
end if;
else
if (FIFO_Delay+Reg_Delay) < counter and counter <= (FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV)+Num_128_P then
A_IV_C_T <= Mux3out;
end if;
end if;
end if;
end process;
------------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
data_valid <='0';
else
if Encryption ='1' then
if (FIFO_Delay+Reg_Delay)-1 < counter
        and counter <=(FIFO_Delay+FIFO_Delay+Reg_Delay)+(Num_128_A+Num_128_IV)+(Num_128_P+Num_128_T+1) then
data_valid <= '1';
else
data_valid <='0';
end if;
else
```

```
if FIFO_Delay+Num_128_A+Num_128_IV+Reg_Delay < counter
        and counter <= (FIFO_Delay+Num_128_A)+(Num_128_IV+Num_128_P)+Reg_Delay then
data_valid <= '1';
else
data_valid <='0';
end if;
end if;
end if;
end if;
end process;
-----------------------------------------
FIFO_12of128b: fifo_16x128b
        port map (
                clk => clk,
                sinit => reset,
                din => A_IV_P_2,
                wr_en => wr_en,
                rd_en => rd_en,
                dout =>FIFO_out,
                full => full,
                empty => empty
        );

Pipelining_AES: AES_Pipelined
        port map(
                encrypt=>start_GCM,
                clk => clk,
                reset=>reset,
                plaintext=> IV_counter,
                ciphertext=> Key_stream
                --encrypt_status=>encrypt_status
        );

Multiplier128x128: Parallel_Multiplied_by_H
        port map(
                clk=>clk,
                reset=>reset,
                H=>H,
                AC=>AC,
                X=>X
        );
-----------------------------------------
process(FIFO_out, Key_stream, counter, Rem_128_P, Num_128_A, Num_128_IV, Num_128_P)
begin
if counter = (FIFO_Delay+Num_128_A)+(Num_128_IV+Num_128_P) then

if Rem_128_P = 0 then

C <= FIFO_out xor Key_stream;
else
C <= FIFO_out xor Key_stream;
C(Rem_128_P to 127) <= (others =>'0');
end if;


else
C <= FIFO_out xor Key_stream;
end if;
end process;
-----------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
TT <=(others => '1');
else
--if Encryption ='0' then
if counter = (Num_128_A+Num_128_IV)+(Num_128_P+Num_128_T) then
TT <= A_IV_P_2;
end if;
--end if;
end if;
end if;
end process;
------------------------------------------

process(Num_128_A, Num_128_IV, Num_128_P, FIFO_Delay)
begin
if (FIFO_Delay) < (Num_128_P+1) then
time_for_calculate_TT_prime <= (Num_128_A+Num_128_IV)+(Num_128_P+2);
else
```

```
time_for_calculate_TT_prime <= (FIFO_Delay+Num_128_A)+(Num_128_IV+1);
end if;
end process;

process(clk)
begin
if clk'event and clk='1' then
if reset='1'then
TT_prime <=(others => '0');
else
--if Encryption ='0' then
if counter= time_for_calculate_TT_prime then
TT_prime <= T;
end if;
--end if;
end if;
end if;
end process;
------------------------------------------
process(clk)
begin
if clk'event and clk='1' then
if reset='1' then
T_Verification <='0';
else
--if Encryption ='0' then
if counter=(time_for_calculate_TT_prime+1) then
if TT_prime /= TT then
T_Verification <= '1';
end if;
end if;
--end if;
end if;
end if;
end process;
------------------------------------------

T <= X xor EKY0;                                            --NOTE: EKY0 IS AES_K(J0)

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    AES_Pipelined - structural
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;                                   --use IEEE.NUMERIC_STD.ALL
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.datatypes.all;

entity AES_Pipelined is
port (
                    encrypt: in std_logic;
                     clk: in std_logic;
                     reset: in std_logic;
                     plaintext: in std_logic_vector(0 to 127);
                     ciphertext: out std_logic_vector(0 to 127)
                     );
end AES_Pipelined;

architecture structural of AES_Pipelined is

component AES_1to9_round
port(
          encrypt: in std_logic;
          clk: in std_logic;
          reset: in std_logic;
          round_input: in std_logic_vector(0 to 127);
          roundkey: in std_logic_vector(0 to 127);
          round_output: out std_logic_vector(0 to 127)
          );
end component;

component AES_last_round
port(
          encrypt: in std_logic;
          clk: in std_logic;
          reset: in std_logic;
          last_round_input: in std_logic_vector(0 to 127);
          roundkey: in std_logic_vector(0 to 127);
```

```
        last_round_output: out std_logic_vector(0 to 127)
        );
end component;

signal first_round_output: std_logic_vector(0 to 127);
signal round_1_output: std_logic_vector(0 to 127);
signal round_2_output: std_logic_vector(0 to 127);
signal round_3_output: std_logic_vector(0 to 127);
signal round_4_output: std_logic_vector(0 to 127);
signal round_5_output: std_logic_vector(0 to 127);
signal round_6_output: std_logic_vector(0 to 127);
signal round_7_output: std_logic_vector(0 to 127);
signal round_8_output: std_logic_vector(0 to 127);
signal round_9_output: std_logic_vector(0 to 127);
signal last_round_output: std_logic_vector(0 to 127);

begin

ciphertext <= last_round_output;
first_round_output <= RoundKeySet(0) xor plaintext;

round_1:
AES_1to9_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        round_input => first_round_output,
        roundkey => RoundKeySet(1),
        round_output => round_1_output
        );

round_2:
AES_1to9_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        round_input => round_1_output,
        roundkey => RoundKeySet(2),
        round_output => round_2_output
        );
```

```
round_3:
AES_1to9_round PORT MAP(
            encrypt => encrypt,
            clk => clk,
            reset => reset,
            round_input => round_2_output,
            roundkey => RoundKeySet(3),
            round_output => round_3_output
            );

round_4:
AES_1to9_round PORT MAP(
            encrypt => encrypt,
            clk => clk,
            reset => reset,
            round_input => round_3_output,
            roundkey => RoundKeySet(4),
            round_output => round_4_output
            );

round_5:
AES_1to9_round PORT MAP(
            encrypt => encrypt,
            clk => clk,
            reset => reset,
            round_input => round_4_output,
            roundkey => RoundKeySet(5),
            round_output => round_5_output
            );

round_6:
AES_1to9_round PORT MAP(
            encrypt => encrypt,
            clk => clk,
            reset => reset,
            round_input => round_5_output,
            roundkey => RoundKeySet(6),
            round_output => round_6_output
            );
```

```
round_7:
AES_1to9_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        round_input => round_6_output,
        roundkey => RoundKeySet(7),
        round_output => round_7_output
        );

round_8:
AES_1to9_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        round_input => round_7_output,
        roundkey => RoundKeySet(8),
        round_output => round_8_output
        );

round_9:
AES_1to9_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        round_input => round_8_output,
        roundkey => RoundKeySet(9),
        round_output => round_9_output
        );


--for last round ( round 10): compared with AES_1to9_round, no MixColumn operation.

last_round:
AES_last_round PORT MAP(
        encrypt => encrypt,
        clk => clk,
        reset => reset,
        last_round_input => round_9_output,
        roundkey => RoundKeySet(10),
        last_round_output => last_round_output
        );
end structural;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    AES_1to9_Round - structural
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;                      --use IEEE.NUMERIC_STD.ALL
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.datatypes.all;

entity AES_1to9_Round is
port(
        encrypt: in std_logic;
        clk: in std_logic;
        reset: in std_logic;
        round_input: in std_logic_vector(0 to 127);
        roundkey: in std_logic_vector(0 to 127);
        round_output: out std_logic_vector(0 to 127)
  );
end AES_1to9_Round;

architecture structural of AES_1to9_Round is

signal SubBytes_input, SubBytes_output: std_logic_vector(0 to 127);
signal ShiftRows_input, ShiftRows_output: std_logic_vector(0 to 127);
signal MixColumn_input, MixColumn_output: std_logic_vector(0 to 127);
signal AddRoundKey_input, AddRoundKey_output: std_logic_vector(0 to 127);
--signal row_index, column_index: index;

begin

SubBytes_input <= round_input;
ShiftRows_input <= SubBytes_output;
MixColumn_input <= ShiftRows_output;
AddRoundKey_input <= MixColumn_output;
round_output <= AddRoundKey_output;

--for SubBytes--------------using 16 Sboxes----------

Read_Sbox: process(clk)
```

```
begin
if clk='1' and clk'event then
for r in 0 to 15 loop
SubBytes_output((0+r*8) to (7+r*8))<=ROM(conv_integer(SubBytes_input((0+r*8) to (7+r*8))));
end loop;
end if;
end process;

--for ShiftRows--------------------------------------

ShiftRows: for r in 0 to 3 generate
For_Column: for c in 0 to 3 generate
ShiftRows_output(((r+4*((c+shiftNb(r))mod Nb))*8) to ((r+4*((c+shiftNb(r))mod Nb))*8+7))
                                                            <=ShiftRows_input(((r+4*c)*8) to ((r+4*c)*8+7));
end generate For_Column;
end generate ShiftRows;

----for MixColumn--------------------------------------

MixColumn: for c in 0 to 3 generate

MixColumn_output((0+4*c*8) to (7+4*c*8)) <= multiply_by_02(MixColumn_input((0+4*c*8) to (7+4*c*8)))

         xor multiply_by_03(MixColumn_input((8+4*c*8) to (8+7+4*c*8)))

         xor MixColumn_input((2*8+4*c*8) to (2*8+7+4*c*8))

         xor MixColumn_input((3*8+4*c*8) to (3*8+7+4*c*8));

MixColumn_output((8+4*c*8) to (8+7+4*c*8)) <= MixColumn_input((0+4*c*8) to (7+4*c*8))

                 xor multiply_by_02(MixColumn_input((8+4*c*8) to (8+7+4*c*8)))

                 xor multiply_by_03(MixColumn_input((2*8+4*c*8) to (2*8+7+4*c*8)))

                 xor MixColumn_input((3*8+4*c*8) to (3*8+7+4*c*8));

MixColumn_output((2*8+4*c*8) to (2*8+7+4*c*8)) <= MixColumn_input((0+4*c*8) to (7+4*c*8))

                  xor MixColumn_input((8+4*c*8) to (8+7+4*c*8))

                  xor multiply_by_02(MixColumn_input((2*8+4*c*8) to (2*8+7+4*c*8)))

                  xor multiply_by_03(MixColumn_input((3*8+4*c*8) to (3*8+7+4*c*8)));
```

```
MixColumn_output((3*8+4*c*8) to (3*8+7+4*c*8)) <= multiply_by_03(MixColumn_input((0+4*c*8) to (7+4*c*8)))

                              xor MixColumn_input((8+4*c*8) to (8+7+4*c*8))

                              xor MixColumn_input((2*8+4*c*8) to (2*8+7+4*c*8))

                              xor multiply_by_02(MixColumn_input((3*8+4*c*8) to (3*8+7+4*c*8)));

end generate MixColumn;

--for AddRoundKey----------------------------------

AddRoundKey_output <= roundkey xor AddRoundKey_input;

end structural;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    AES_last_round - structural
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;                         --use IEEE.NUMERIC_STD.ALL
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.datatypes.all;

entity AES_last_round is
port(
        encrypt: in std_logic;
        clk: in std_logic;
        reset: in std_logic;
        last_round_input: in std_logic_vector(0 to 127);
        roundkey: in std_logic_vector(0 to 127);
        last_round_output: out std_logic_vector(0 to 127)
   );
end AES_last_round;

architecture structural of AES_last_round is

signal SubBytes_input, SubBytes_output: std_logic_vector(0 to 127);
signal ShiftRows_input, ShiftRows_output: std_logic_vector(0 to 127);
signal AddRoundKey_input, AddRoundKey_output: std_logic_vector(0 to 127);

begin

SubBytes_input <= last_round_input;
ShiftRows_input <= SubBytes_output;
AddRoundKey_input <= ShiftRows_output;
last_round_output <= AddRoundKey_output;

--for SubBytes--------------using 16 Sboxes----------

Read_Sbox: process(clk)
begin
if clk='1' and clk'event then
for r in 0 to 15 loop
```

```
SubBytes_output((0+r*8) to (7+r*8))<=ROM(conv_integer(SubBytes_input((0+r*8) to (7+r*8))));
end loop;
end if;
end process;

--for ShiftRows--------------------------------------

ShiftRows: for r in 0 to 3 generate
For_Column: for c in 0 to 3 generate
ShiftRows_output(((r+4*((c+shiftNb(r))mod Nb))*8) to ((r+4*((c+shiftNb(r))mod Nb))*8+7))
                                                                            <=ShiftRows_input(((r+4*c)*8) to ((r+4*c)*8+7));
end generate For_Column;
end generate ShiftRows;

--for AddRoundKey---------------------------------

AddRoundKey_output <= roundkey xor AddRoundKey_input;

end structural;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    Bit_Parallel_Multiplied_by_H - structural
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.datatypes.all;

entity Parallel_Multiplied_by_H is
port(
        clk: in std_logic;
        reset: in std_logic;
        H: in std_logic_vector(0 to 127);
        AC: in std_logic_vector(0 to 127);
        X: out std_logic_vector(0 to 127)
        );
end Parallel_Multiplied_by_H;

architecture structural of Parallel_Multiplied_by_H is

--signal temp_out: std_logic_vector(0 to 127);

begin

process(H, AC)
variable temp: std_logic_vector(0 to 127);
variable MatrixU: MatrixU;
begin
temp:=(others=>'0');
MatrixU(0):=H;

for i in 0 to 126 loop
if MatrixU(i)(127)='0' then
MatrixU(i+1) := rightshift(MatrixU(i));
else
MatrixU(i+1):= (rightshift(MatrixU(i))(0 to 7) xor R) & rightshift(MatrixU(i))(8 to 127);
end if;
end loop;
```

```
for k in 0 to 127 loop
for l in 0 to 127 loop
temp(k):=temp(k) xor (MatrixU(l)(k) and AC(l));
end loop;
end loop;
X<= temp;
end process;

--process(clk)
--begin
--if clk'event and clk='1' then
--if reset='1' then
--X<=(others=>'0');
--else
--X<=temp_out;
--end if;
--end if;
--end process;

end structural;
```

# Appendix D

# Test-bench for the GCM module

```
-------------------------------------------------------------------------------
-- Design Name:   AES-GCM
-- Module Name:   test_GCM.vhd                (TEST BENTCH FOR GCM.VHD)
-------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY test_GCM_vhd IS
END test_GCM_vhd;

ARCHITECTURE behavior OF test_GCM_vhd IS

        -- Component Declaration for the Unit Under Test (UUT)
        COMPONENT gcm
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                start_GCM : IN std_logic;
                Encryption : IN std_logic;
                H : IN std_logic_vector(0 to 127);
                A_IV_P : IN std_logic_vector(0 to 127);
                A_P_Len : IN std_logic_vector(31 downto 0);
                T_Verification: out std_logic;
                A_IV_C_T : OUT std_logic_vector(0 to 127);
                data_valid : OUT std_logic
                );
        END COMPONENT;

        --Inputs
        SIGNAL clk :  std_logic := '0';
        SIGNAL reset :  std_logic := '0';
        SIGNAL start_GCM :  std_logic := '0';
        SIGNAL Encryption :  std_logic := '0';
        signal T_Verification: std_logic;
```

```
    SIGNAL H :  std_logic_vector(0 to 127) := (others=>'0');
    SIGNAL A_IV_P :  std_logic_vector(0 to 127) := (others=>'0');
    SIGNAL A_P_Len :  std_logic_vector(31 downto 0) := (others=>'0');

    --Outputs
    SIGNAL A_IV_C_T :  std_logic_vector(0 to 127);
    SIGNAL data_valid :  std_logic;
signal count_b: integer range 0 to 31;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: GCM PORT MAP(
        clk => clk,
        start_GCM => start_GCM,
        Encryption => Encryption,
        reset => reset,
        H => H,
        A_IV_P => A_IV_P,
        A_P_Len => A_P_Len,
        T_Verification=>T_Verification,
        A_IV_C_T => A_IV_C_T,
        data_valid => data_valid
    );

    clk <= not clk after 12 ns;
    reset <= '1' after 3 ns, '0' after 43 ns;
    start_GCM <= '1' after 56 ns;
    Encryption <='0';
    A_P_Len <= X"00a001e0";    --X"00000200" for test case 3 ; --X"00a001e0" for test case 4
    H <= X"B83B533708BF535D0AA6E52980D53B78";



    BT : PROCESS(clk)
    BEGIN
    if clk'event and clk='1' then
    if reset='1' then
    A_IV_P <= (others =>'0');
    count_b <=0;
    --start_GCM <= '0';
```

```
            else

            if Encryption ='1' then

--          if count_b=0 then
--          A_IV_P <= X"cafebabefacedbaddecaf88800000001";                    --use test case 3
--          elsif count_b=1 then
--          A_IV_P <= X"d9313225f88406e5a55909c5aff5269a";
--          elsif count_b=2 then
--          A_IV_P <= X"86a7a9531534f7da2e4c303d8a318a72";
--          elsif count_b=3 then
--          A_IV_P <= X"1c3c0c95956809532fcf0e2449a6b525";
--          else
--          A_IV_P <= X"b16aedf5aa0de657ba637b391aafd255";
--          end if;

            if count_b=0 then
            A_IV_P <= X"feedfacedeadbeeffeedfacedeadbeef";                    --use test case 4
            elsif count_b=1 then
            A_IV_P <= X"abaddad2000000000000000000000000";
            elsif count_b=2 then
            A_IV_P <= X"cafebabefacedbaddecaf88800000001";
            elsif count_b=3 then
            A_IV_P <= X"d9313225f88406e5a55909c5aff5269a";
            elsif count_b=4 then
            A_IV_P <= X"86a7a9531534f7da2e4c303d8a318a72";
            elsif count_b=5 then
            A_IV_P <= X"1c3c0c95956809532fcf0e2449a6b525";
            else
            A_IV_P <= X"b16aedf5aa0de657ba637b3900000000";
            end if;

            else

--          if count_b=0 then
--          A_IV_P <= X"cafebabefacedbaddecaf88800000001";
--          elsif count_b=1 then
--          A_IV_P <= X"42831ec2217774244b7221b784d0d49c";
--          elsif count_b=2 then
--          A_IV_P <= X"e3aa212f2c02a4e035c17e2329aca12e";
--          elsif count_b=3 then
--          A_IV_P <= X"21d514b25466931c7d8f6a5aac84aa05";
--          elsif count_b=4 then
--          A_IV_P <= X"1ba30b396a0aac973d58e091473f5985";
--          else
--          A_IV_P <= X"4d5c2af327cd64a62cf35abd2ba6fab4";
--          end if;

            if count_b=0 then
            A_IV_P <= X"feedfacedeadbeeffeedfacedeadbeef";
            elsif count_b=1 then
            A_IV_P <= X"abaddad2000000000000000000000000";
            elsif count_b=2 then
            A_IV_P <= X"cafebabefacedbaddecaf88800000001";
            elsif count_b=3 then
            A_IV_P <= X"42831ec2217774244b7221b784d0d49c";
            elsif count_b=4 then
            A_IV_P <= X"e3aa212f2c02a4e035c17e2329aca12e";
            elsif count_b=5 then
            A_IV_P <= X"21d514b25466931c7d8f6a5aac84aa05";
            elsif count_b=6 then
            A_IV_P <= X"1ba30b396a0aac973d58e09100000000";
            else
            A_IV_P <= X"5bc94fbc3221a5db94fae95ae7121a47";
            end if;
            end if;

            if count_b/=31 then
            count_b<=count_b+1;
            end if;
            end if;
            end if;
            END PROCESS;
END;
```

# Appendix E
# VHDL Codes of IPsec ESP Signal Generator

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    IPsec_Signal_Source - Behavioral
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IPsec_Signal_Source is
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_GCM: out std_logic;
                    H: out std_logic_vector(0 to 127);
                    A_IV_P: out std_logic_vector(0 to 127);
                    A_P_Len: in std_logic_vector(31 downto 0)
          );
end IPsec_Signal_Source;

architecture Behavioral of IPsec_Signal_Source is

component LFSR_16bit
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_LFSR: in std_logic;
                    LFSR_IV: in std_logic_vector(0 to 15);
                    LFSR_parallel_out: out std_logic_vector(0 to 15)
          );
end component;

--signal A_P_Len_1, A_P_Len_2, A_P_Len_3, A_P_Len_4: std_logic_vector(31 downto 0);

signal Rem_128_A, Rem_128_P: integer range 0 to 127;
signal Num_128_A, Num_128_P: integer range 0 to 511;
signal FIFO_Delay: integer range 0 to 15;
signal Num_128_IV: integer range 0 to 1;
signal Reg_Delay: integer range 0 to 1;
signal Num_128_T: integer range 0 to 1;

signal counter: integer range 0 to 1023;
signal start_LFSR: std_logic;
signal GCM_IV, LFSR_128bit: std_logic_vector(0 to 127);
signal LFSR_IV: std_logic_vector(0 to 15);

begin
------Calculate Control parameter--------

Rem_128_A <= conv_integer(A_P_Len(22 downto 16));       -- indicate how many bits for last segment of A;
Rem_128_P <= conv_integer(A_P_Len(6 downto 0));         -- indicate how many bits for last segment of P;

process(A_P_Len, Rem_128_A, Rem_128_P)
begin
if Rem_128_A =0 then
Num_128_A <= conv_integer(A_P_Len(31 downto 23));
else
Num_128_A <= (conv_integer(A_P_Len(31 downto 23)) + 1);
end if;
if Rem_128_P =0 then
Num_128_P <= conv_integer(A_P_Len(15 downto 7));
else
Num_128_p <= (conv_integer(A_P_Len(15 downto 7)) + 1);
end if;
end process;

FIFO_Delay <= 11;                   -- 11 cycles delay for deleted bulb between A_IV and C;
Num_128_IV <= 1;                    -- at here, We only choose IV is 96-bit long, so Num_128_IV = 1;
Reg_Delay <= 1;                     -- 1 cycles delay for deleted bulb between C and T;
Num_128_T <= 1;                     -- the number of segments of the 128-bit of T or the number of 128-bit of A_P_Len;

------------------------------------------

process(clk)
begin
if clk'event and clk='1' then
if reset ='1' then
counter <=0;
start_GCM <= '0';
start_LFSR <='0';
else

if counter =0 or counter = (Num_128_A+1)
                    or counter = (Num_128_A+Num_128_IV+Num_128_P+Num_128_T+FIFO_Delay+FIFO_Delay+3) then
start_LFSR <='0';
else
start_LFSR <='1';
end if;

if counter = (Num_128_A+Num_128_IV+Num_128_P+Num_128_T+FIFO_Delay+FIFO_Delay+3) then
start_GCM <='0';
counter <= 0;
else
start_GCM <='1';
counter <= counter + 1;
end if;


end if;
end if;
end process;

process(start_LFSR, GCM_IV,LFSR_128bit )
begin
if start_LFSR ='0' then
A_IV_P <= GCM_IV;
else
A_IV_P <= LFSR_128bit;
end if;
end process;

LFSR_128bit0_15:LFSR_16bit
          port map(
                    clk=>clk,
                    reset=>reset,
                    start_LFSR=>start_LFSR,
                    LFSR_IV=>LFSR_IV,
                    LFSR_parallel_out=>LFSR_128bit(0 to 15)
          );

LFSR_128bit16_31:LFSR_16bit
          port map(
                    clk=>clk,
                    reset=>reset,
                    start_LFSR=>start_LFSR,
                    LFSR_IV=>LFSR_IV,
                    LFSR_parallel_out=>LFSR_128bit(16 to 31)
          );

LFSR_128bit32_47:LFSR_16bit
          port map(
                    clk=>clk,
                    reset=>reset,
                    start_LFSR=>start_LFSR,
                    LFSR_IV=>LFSR_IV,
                    LFSR_parallel_out=>LFSR_128bit(32 to 47)
          );

LFSR_128bit48_63:LFSR_16bit
          port map(
                    clk=>clk,
                    reset=>reset,
                    start_LFSR=>start_LFSR,
                    LFSR_IV=>LFSR_IV,
                    LFSR_parallel_out=>LFSR_128bit(48 to 63)
          );

LFSR_128bit64_79:LFSR_16bit
          port map(
                    clk=>clk,
                    reset=>reset,
                    start_LFSR=>start_LFSR,
                    LFSR_IV=>LFSR_IV,
                    LFSR_parallel_out=>LFSR_128bit(64 to 79)
          );

LFSR_128bit80_95:LFSR_16bit
```

```
        port map(
                clk=>clk,
                reset=>reset,
                start_LFSR=>start_LFSR,
                LFSR_IV=>LFSR_IV,
                LFSR_parallel_out=>LFSR_128bit(80 to 95)
        );

LFSR_128bit96_111:LFSR_16bit
        port map(
                clk=>clk,
                reset=>reset,
                start_LFSR=>start_LFSR,
                LFSR_IV=>LFSR_IV,
                LFSR_parallel_out=>LFSR_128bit(96 to 111)
        );

LFSR_128bit112_127:LFSR_16bit
        port map(
                clk=>clk,
                reset=>reset,
                start_LFSR=>start_LFSR,
                LFSR_IV=>LFSR_IV,
                LFSR_parallel_out=>LFSR_128bit(112 to 127)
        );


H <= X"B83B533708BF535D0AA6E52980D53B78";
GCM_IV <= X"cafebabefacedbaddecaf88800000001";

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- Design Name:    AES-GCM
-- Module Name:    16bit_LFSR - Behavioral
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity LFSR_16bit is
port(
                    clk: in std_logic;
                    reset: in std_logic;
                    start_LFSR: in std_logic;
                    LFSR_IV: in std_logic_vector(0 to 15);
                    LFSR_parallel_out: out std_logic_vector(0 to 15)
          );
end LFSR_16bit;

architecture Behavioral of LFSR_16bit is

signal LFSR: std_logic_vector(0 to 15);

begin

process(clk)
begin
if clk'event and clk='1' then
if reset='0' and start_LFSR='1'then

for i in 0 to 14 loop
LFSR(i+1) <= LFSR(i);
end loop;
LFSR(0) <= LFSR(15) xnor LFSR(14) xnor LFSR(12) xnor LFSR(3);          --from Xilinx;

--for i in 0 to 14 loop
--LFSR(i) <= LFSR(i+1);
--end loop;
--LFSR(15) <= (LFSR(12) xor LFSR(3)) xor (LFSR(1) xor LFSR(0));--from primitive polynomial 1+x+x^3+x^12+x^16;
```

```
else
LFSR <=X"1234";
end if;
end if;
end process;

LFSR_parallel_out <= LFSR;

end Behavioral;
```

# Appendix F

## The Critical Path on GCM module

```
=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 13.213ns (frequency: 75.683MHz)
  Total number of paths / destination ports: 31206775 / 2976
-------------------------------------------------------------------------
Delay:               13.213ns (Levels of Logic = 20)
  Source:            A_P_Len_1_24 (FF)
  Destination:       A_IV_C_T_9 (FF)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: A_P_Len_1_24 to A_IV_C_T_9
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDE:C->Q              7   0.374   0.690  A_P_Len_1_24 (A_P_Len_1_24)
     LUT1_L:I0->LO         1   0.313   0.000  A_P_Len_1_24_rt (A_P_Len_1_24_rt)
     MUXCY:S->O            1   0.377   0.000  gcm__n0315<1>cy (gcm__n0315<1>_cyo)
     MUXCY:CI->O           1   0.042   0.000  gcm__n0315<2>cy (gcm__n0315<2>_cyo)
     MUXCY:CI->O           1   0.042   0.000  gcm__n0315<3>cy (gcm__n0315<3>_cyo)
     MUXCY:CI->O           1   0.042   0.000  gcm__n0315<4>cy (gcm__n0315<4>_cyo)
     MUXCY:CI->O           1   0.042   0.000  gcm__n0315<5>cy (gcm__n0315<5>_cyo)
     XORCY:CI->O           4   0.868   0.514  gcm__n0315<6>_xor (_n0315<6>)
     LUT3_L:I2->LO         1   0.313   0.000  Num_128_A<6>11 (N1283)
     MUXCY:S->O            1   0.377   0.000  gcm__n1241<6>cy (gcm__n1241<6>_cyo)
     XORCY:CI->O          10   0.868   0.744  gcm__n1241<7>_xor (_n1241<7>)
     LUT1_L:I0->LO         1   0.313   0.000  _n1241<7>_rt1 (_n1241<7>_rt1)
     MUXCY:S->O            1   0.377   0.000  gcm__n0312<7>cy (gcm__n0312<7>_cyo)
     XORCY:CI->O           2   0.868   0.473  gcm__n0312<8>_xor (_n0312<8>)
     LUT4_L:I2->LO         1   0.313   0.000  gcm__n0331<8>lut (N368)
     MUXCY:S->O            1   0.377   0.000  gcm__n0331<8>cy (gcm__n0331<8>_cyo)
     XORCY:CI->O           1   0.868   0.533  gcm__n0331<9>_xor (_n0331<9>)
     LUT2_L:I0->LO         1   0.313   0.000  XNor_stagelut147 (N412)
     MUXCY:S->O            1   0.377   0.000  XNor_stagecy_rn_146
(XNor_stage_cyo144)
     MUXCY:CI->O         142   0.525   1.024  XNor_stagecy_rn_147 (_n0628)
     LUT4:I3->O           10   0.313   0.601  _n09531_13 (_n09531_12)
     FDE:CE                    0.335          A_IV_C_T_90
    ----------------------------------------
    Total                     13.213ns (8.635ns logic, 4.579ns route)
                                       (65.3% logic, 34.7% route)
```

```
=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 124379 / 1509
-------------------------------------------------------------------------
Offset:              10.231ns (Levels of Logic = 12)
  Source:            H<127> (PAD)
  Destination:       A_IV_C_T_11 (FF)
  Destination Clock: clk rising

  Data Path: H<127> to A_IV_C_T_11
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O           268   0.919   1.327  H_127_IBUF (H_127_IBUF)
     begin scope: 'Multiplier128x128'
     LUT2:I0->O           16   0.313   0.884  _n0002<120>1 (_n20873<119>)
     LUT4:I0->O            6   0.313   0.667  _n0365<120>1 (_n21979<119>)
     LUT2:I0->O            1   0.313   0.533  _n205701 (_n20777<127>)
     LUT4:I0->O            1   0.313   0.440  Mxor__n3592_inst_lut4_41271
(Mxor__n3592__net31)
     LUT4:I3->O            1   0.313   0.440  Mxor__n3592_inst_lut4_41351
(Mxor__n3592__net40)
     LUT4:I3->O            1   0.313   0.506  Mxor__n3592_inst_lut4_41371
(Mxor__n3592__net43)
     LUT2:I1->O            3   0.313   0.495  Mxor__n3592_inst_lut2_01 (X<11>)
     end scope: 'Multiplier128x128'
     LUT4:I2->O            1   0.313   0.440  _n1230<3>441_SW0_SW0 (N3537)
     LUT4_L:I3->LO         1   0.313   0.216
Mmux_Mux2out_Mux2out<112>_Mux2out<112>_rn_5111_SW0 (N3104)
     LUT4_L:I1->LO         1   0.313   0.000
Mmux_Mux2out_Mux2out<112>_Mux2out<112>_rn_5111 (Mux3out<11>)
     FDE:D                     0.234          A_IV_C_T_11
    ----------------------------------------
    Total                     10.231ns (4.283ns logic, 5.948ns route)
                                       (41.9% logic, 58.1% route)
```

# Glossary of Terms and Acronyms

The following definitions are used throughout this Thesis:

GCM The forth recommendation of symmetric block cipher mode of operation SP800-38D, *Galois/Counter Mode of Operation*

NIST The National Institute of Standards and Technology

ECB Electronic Codebook mode, the one of five confidentiality modes of operation recommended by NIST for use with an underlying symmetric key block cipher algorithm

CBC Cipher Block Chaining mode mode, the one of five confidentiality modes of operation recommended by NIST for use with an underlying symmetric key block cipher algorithm

CFB Cipher Feedback mode, the one of five confidentiality modes of operation recommended by NIST for use with an underlying symmetric key block cipher algorithm

OFB Output Feedback mode, the one of five confidentiality modes of operation recommended by NIST for use with an underlying symmetric key block cipher algorithm

CTR Counter mode, the one of five confidentiality modes of operation recommended by NIST for use with an underlying symmetric key block cipher algorithm

MAC Message Authentication Code

AES Advanced Encryption Standard

FIPS Federal Information Processing Standard

FPGA Field Programmable Gate Array

GF Galois Field

CMC Canadian Microelectronics Corporation

| | |
|---|---|
| PLB | IBM Processor Local Bus |
| OPB | IBM On-Chip Peripheral Bus |
| SoC | System on a Chip |
| ESP | IPsec Encapsulating Security Payload |
| FSM | Finite State Machine |
| MGT | Multi-Gigabit Transciever |
| IV | Initialization Vector |
| P | Plaintext |
| AAD | Additional Authenticated Data, also denoted as A. |
| C | Ciphertext |
| T | Authentication Tag |
| GHASH | The specified universal hash function in GCM is defined over a binary Galois field and is a 128-bit polynomial multiplier over GF ($2^{128}$) which can provide an authentication mechanism. |
| GCTR | For the confidentiality mechanism of GCM, the CTR mode embedded by ECB mode, is adopted using an underlying block cipher. |

# Bibliography

[1] NIST Special Publication 800-38A, *Recommendation for Block Cipher Modes of Operation—Methods and Techniques*, December 2001.

[2] NIST Special Publication 800-38D Draft, *Recommendation for Block Cipher Modes of Operation— Galois/Counter Mode (GCM) for Confidentiality and Authentication*, April 2006.

[3] FIPS Publication 197, *the Advanced Encryption Standard (AES)*, U.S. DoC/NIST, November, 2001.

[4]. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press LLC, 1997.

[5] Shu Lin, Daniel J. Costello, *Error control Coding*-Fundamentals and Applications, Englewood Cliffs, N.J. : Prentice-Hall, c1983

[6] E. D. Mastrovito, *VLSI Designs for Multiplication over Finite Fields GF($2^m$)*, in Proc. Sixth Int'l Con, Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes (AAECC-6). (Rome), pp. 297-309, July 1988.

[7] M. Anwarul Hasan. *Look-up table-based large finite field multiplication in memory constrained cryptosystems*. IEEE Transactions on Computers, 49(7), July 2000.

[8] Martin Feldhofer, Kerstin Lemke, Elisabeth Oswald, Francois-Xavier Standaert, Thomas Wollinger and Johannes Wolkerstorfer, *State of the Art in Hardware Architectures*. Information Society Technologies, September 2005.

[ 9] RFC 2406 — *IP Encapsulating Security Payload* (ESP). Network Working Group, November 1998.

[10 ] RFC 4106 — *The Use of Galois-Counter Mode (GCM) in IPsec Encapsulating Security Payload* (ESP). Network Working Group, June 2005.

[11 ] Product introduction: *CLP-24 AES-GCM Core* Elliptic Semiconductor Inc. 2006. http://www.ellipticsemi.com/products-clp-24.php

[12] D. McGrew, J. Viega, *The Galois/Counter Mode of Operation* (GCM), May 31, 2005. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/

[13] AMIRIX Systems Inc. *AP1000 FPGA Development Board Users Guide*. Document #: DOC-004017 Version 02.

[14] Platform Studio Debugging PowerPC Hardware Setup http://toolbox.xilinx.com/docsan/xilinx8/EDKHelp/platform_studio/html/ps_p_dbg_debugging_ppc_hw_setup.htm.

[15]Virtex-II Pro and Virtex-II Pro X Platform FPGAs Complete Data sheet Sheethttp://www.xilinx.com/bvdocs/publications/ds083.pdf.

[16] Embedded System Tools Reference Manual Embedded Development Kit EDK 7.1i Virtex-II Pro and Virtex-II Pro X FPGA User Guide, http://www.xilinx.com/bvdocs/userguides/ug012.pdf.

[17] Arash Reyhani-Masoleh, ,M. Anwar Hasan, *Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over* $GF(2^m)$. IEEE Transactions on Computers, vol. 53, No. 8, August 2004.

[18] William Stallings, *Cryptography and Network Security* fourth edition. Upper Saddle River, N.J. : Prentice Hall, c2006.

[19]G. Ahlquist, B. Nelson, and M. Rice, *Optimal Finite Field Multipliers for FPGAs*, International Workshop on Field Programmable Logic and Applications, pp. 51-60, August, 1999.

[20] Bo Yang, Sambit Mishra, and Ramesh Karri. *High Speed Architecture for Galois/Counter Mode of Operation (GCM)*. Cryptology ePrint Archive, Report 2005-156, May 2005. http://eprint.iacr.org/2005/146

[21] NIST Special Publication 800-38B, *Recommendation for Block Cipher Modes of Operation: the CMAC Authentication Mode*. U.S. DoC/NIST, October 2003. Available at http://csrc.nist.gov/CryptoToolkit/modes.

[22] NIST Special Publication 800-38C, *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. U.S. DoC/NIST, May 2004. Available at http://csrc.nist.gov/CryptoToolkit/modes.

[23] Ian Kuon and Jonathan Rose, *Measuring the Gap between FPGAs and ASICs*, ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA, Fourteenth ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA 2006, p 21-30, 2006.

[24]Amirix AP1000 Datasheet. http://www.amirix.com/downloads/ap1000.pdf.

125