KARACHI INSTITUTE OF ECONOMICS AND TECHNOLOGY

MS THESIS

Implementation of Galois Counter Mode Encryption and Authentication for 100Gbps High Speed Ethernet on FPGA

Author:
Ali Umair

Supervisor:
Dr. Hussain Parvez

A thesis submitted in fulfillment of the requirements

for the degree of Science

in the

Embedded System Research Group

Graduate School of Science and Engineering

December, 2016

Declaration of Authorship

I, ALI UMAIR, declare that this thesis titled, ’Implementation of Galois Counter Mode Encryption and Authentication for 100Gbps High Speed Ethernet on FPGA’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: ______________________________

Date: ______________________________

Abstract

Embedded System Research Group

Graduate School of Science and Engineering

Masters of Science

Implementation of Galois Counter Mode Encryption and Authentication for 100Gbps High Speed Ethernet on FPGA

By ALI UMAIR

Security is one of the most challenging subject of today's era of Cloud Computing. Cisco® claims that Annual Global IP traffic will surpass 1 zettabyte (1000 exabytes) by end of 2016[1]. With the increase in internet traffic, there is a subsequent increase in demand of high performance internet security hardware. In this thesis we present high performance hardware architecture of Advanced Encryption Standard(AES) with Galois Counter Mode. This encryption and authentication algorithm is part of IEEE 802.3ba Ethernet standard that provides throughput of 100Gbits/sec. We selected FPGAs as platform of our choice due to its growing demand in reconfigurable communication systems. In previous implementations, the same objective is achieved with four parallel AES-GCM cores. We experimented with different variants of Karatsuba multiplier to optimize for Area and speed. Also, we explored a unique way to pipeline the Galois multiplier while maintaining the functionality. Our improved hardware architecture will achieve the same performance with two parallel cores, saving ~40% of FPGA resources.

Acknowledgements

I would like to dedicate this work to my mom. Being a working woman, she had a very tough life. She invested most part of her income to my education. Back in 2000, I was in eight standard when I had my first computer. My mom paid the installments for the duration of 24 months. I believe that behind every successful man, there is a woman called mother.

I would like to sincerely thank my supervisor Dr. Hussain Parvez, for guiding and supporting me while I was working on this thesis. During my last semester, I got an opportunity in Intel Mexico and I wanted to relocate and continue my work remotely. It was a difficult decision for me but more than that, it was tough for Dr. Hussain to collaborate and guide me thru emails, skype and phone calls. He really motivated me to continue my Thesis and it would be impossible without his supervision.

Table of Contents

List of Figures

List of Tables

Abbreviations

FPGA	Field Programmable Gate Array
AES	Advanced Encryption Standard
NIST	National Institute of Standards and Technology
DES	Data Encryption Standard
FIPS	Federal Information Processing Standards
CBC	Cipher Block Chaining
CF	Cipher Feedback
GCM	Galois Counter Mode
AAD	Additional Authenticated Data
IV	Initialization Vector
OFB	Output Feedback
DOS	Denial of Service
IOT	Internet of Things

1 Introduction

With the increase in demand of Online gaming, online social networking, Digital TV, Online Music, Online Video, Video on demand, cloud computing and 5G connected home applications(such as home automation, home security and video surveillance) and other device, the amount of internet traffic has increased enormously. Cisco®, the largest networking equipment manufacturer, predicts that by the end of 2016, the annual global IP traffic will hit zettabyte which will be 2.3 Zettabyte by the end of this decade[1]. Privacy and Security remains always a major concern for many Technology companies and groups.

1.1 Motivation

The end user wants online experience to be always available and always secure. Security risks includes: Privacy breach, financial hazards, security for Intellectual property and trade secrets and online money transactions.

In Dec, 2014, Sony admitted a major security crack in their systems. Hackers hacked the system and stole peoples' private information and pre-released movies maintained on Sony's Playstation server.

According to Cisco®:

- There has been an increase of 2.5 times the Denial of Service(DOS) attack since last three years [1]
- 458% increase in the number of times, hackers tried to hack Internet of thing(IOT) connections [1]

- 55% increase in spear phishing attack targeted to different organizations to obtain confidential information thru emails [1].
- Malware attacks nearly doubled with Android devices the prime target
- 221% in increase of successfully hacked websites by WordPress attack
- 89% increase of cyber-attacks with financial motivations.

Security is maintained at different levels and layers during the transfer of information. For the Media Access(MAC) level, IEEE standard 802.3ba, AES GCM is the standard choice. This package of encryption and authentication, offers robust security against WordPress or Brute Force attacks.

Performance is a major challenge while implementing AES GCM in hardware or software. AES GCM is computationally expensive and requires 2x the area of Ethernet hardware implementation.

AES GCM contains two separate layers: the encryption and authentication. The encryption is block cipher and has multiple rounds. There are many contributions to improve performance per unit area for AES. The authentication is based on GHASH algorithm.

GHASH requires GF 2^{128} multiplication with a feedback path. The orientation of Multiplier and feedback path, limits the overall performance of AES GCM. There are different algorithms to perform multiplication with Area-Speed tradeoffs. Complexity of multiplication and some proposed methods are discussed in details in later sections.

1.2 Contribution

AES-GCM encryption and authentication presented in 802.3ba directly couples with standard Ethernet interfaces. 100Gbps Ethernet standard, 802.3bm, is the fastest standard interface available for communication. In previous works, we have implementation for AES-GCM that supports 100Gbps of throughput using four parallel AES-GCM engines. We experimented with different proposed algorithms and their variants for GF multiplication and drive an optimal way to implement it on FPGA. In this thesis, we proposed complete structure for AES-GCM encryption and authentication that can support throughput of 100Gbps using only two parallel AES-GCM engines. Thus our solution saves around 40% of FPGA area. We generated results for Xilinx and Altera FPGAs.

1.3 Thesis organization

The remainder of the thesis is organized as follows:

- Chapter 2: In chapter two we discuss the AES and GCM encryption and authentication algorithms in details
- Chapter 3: In this chapter we explain the details of the problem addressed in this thesis i.e. solution for 100Gbps AES-GCM. We also discuss, the proposed solutions for the similar work
- Chapter 4: In chapter 4, we presented our proposed architecture for AES-GCM that can support 100Gbps throughput using two parallel cores

- Chapter 5: In this chapter, we presented simulation results and synthesis results based on Altera's FPGA. We also shared synthesis results for 100Gbps Etherenet solution provided by Altera.
- Chapter 6: We discussed the summary of the thesis and the future work

2 Introduction to Advanced Encryption Standard and GHASH Algorithm

AES(Advanced Encryption Standard) is one of the most famous block encryption standard after DES(Data Encryption Standard). The standard is effective and functional since May, 2002. AES was established and standardized by NIST (National Institute of Standards and Technology) as the base standard for many encryption domains. Considering its popularity, giant processor manufacturing companies like Intel and ARM has included a dedicated hardware for AES as part of their processor die and they support AES as part of instruction set.

AES can support different key size combinations including 64-bit key, 128-bit key and 256-bit key. The wider the key size the more stronger is the encryption. AES is commonly used with other block cipher mode of operations like CBC (Cipher Block Chaining), CFB (Cipher Feedback), OFB (Output Feedback) and GCM (Galois Counter Mode)

2.1 Functional Description of AES

In this section we will discuss the function used in AES algorithm in details. As we can see in algorithm mentioned earlier, that each round has an input called *state.* A *state* is a 128-bit vector.

Here is the algorithm that taken from FIPS-197 AES specification:

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr–1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for
    SubBytes(state) ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    out = state
end
```

Figure 2.1: AES algorithm taken from FIPS-197, AES Standard

A round is a combination of four transformations: SubBytes(), ShiftRows(),MixColumns() and AddRoundKey()s. Number of rounds in an AES is decided by the key size. Following tables shows the relation between key size, number of rounds and block size

	AES-128	**AES-192**	**AES-256**
Key Length	128-bit	192-bits	256-bits
Block Size	128-bits	128-bits	128-bits
Number of Rounds	10	12	14

Table 2.1: AES rounds depending on key size

You can noticed that block size doesn't vary with the key size. Also, another important fact is that in each round a different key is used. These N-different keys are extracted from the base key used as input to the AES encryption.

2.1.1 AddRoundKey Transformation

In AddRoundKey transformation, round key is xored with the state.

$$\text{State}_{N+1} = \text{State}_{N} \oplus \text{Key}_{\text{Round}} \qquad (2.1.1)$$

2.1.2 SubBytes Transformation

In SubBytes, a non-linear transformation is applied on each byte of State. Byte substitution applied on each byte of state are independent of each other. Following diagram is taken from FIPS-197 document:

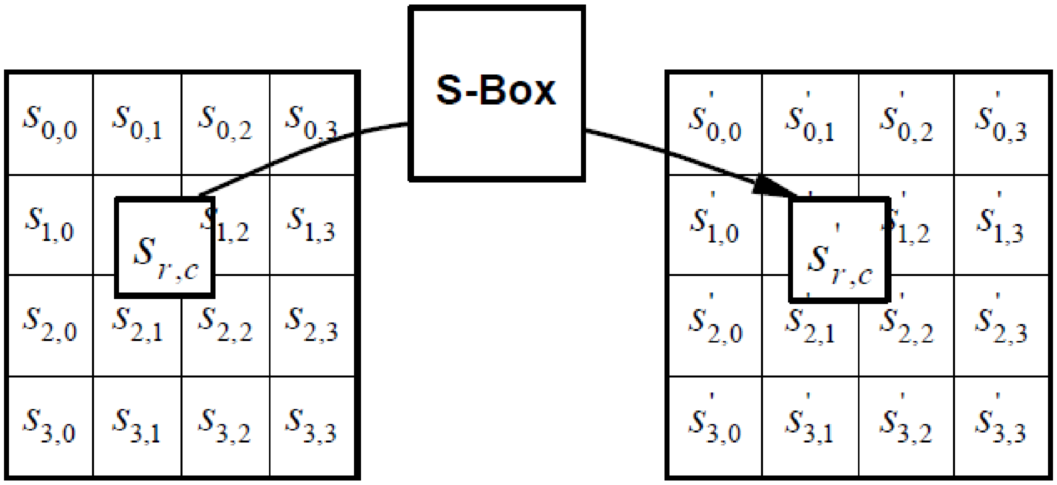

S-Box
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{r,c}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S'_{0,0}$ $S'_{0,1}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{r,c}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{2,0}$ $S'_{2,1}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$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

Figure 2.2: Byte Substitution Operation

This complex transformation is usually performed with the help of a table stored in FPGA memory. Here is table sourced from FIPS-197 AES standard:

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

Figure 2.3: Bytes substitution lookup table

This table when stored in FPGA LUT based Memory, occupies 40-ALMs in both Xilinx and Altera's FPGA.

An example of Byte Substitution:

Suppose you have a 128-bit state and byte-0 is ‘8’h63’so for byte substitution we will take ‘6’ for row search and ‘3’ for column search and the intersect of row-6 and column-3 is ‘fb’. So after byteSub, ‘8h63’ will become ‘8’hfb’

2.1.3 ShiftRows Transformation

Shift rows transformation is the lightest transformation to implement on FPGA and actually requires no logic at all. In Shift Rows, state vector is supposed to be arranged in a matrix form. Then each row of the matrix is left shifted, circularly. The amount of left shift is based on the row number. So the Row-0 is not shifted at all, Row-1 is shifted once, Row-2 and Row-3 are shifted twice and thrice respectively.

Following diagram is taken from FIPS-197 document and it explains row shift operation very well:

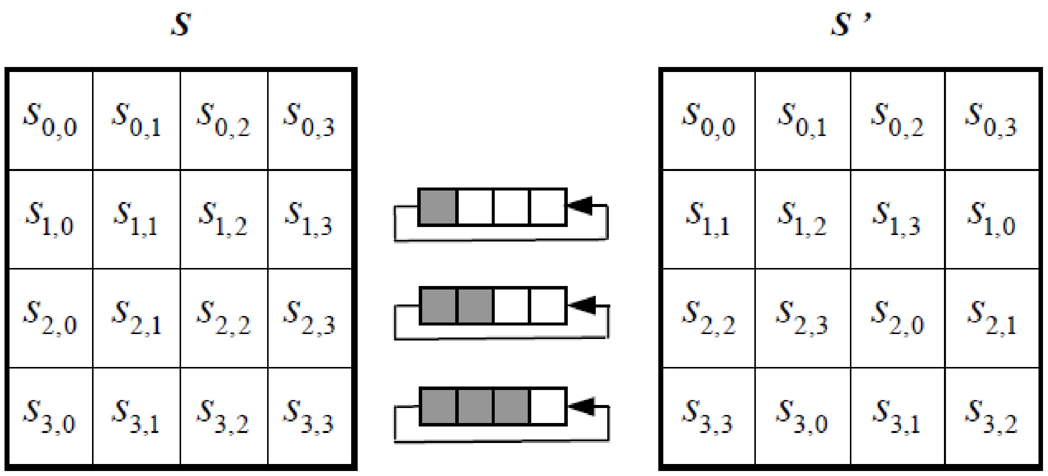

S
S ’
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 2.4: Shift Row Operation

2.1.4 MixColumns Transformation

Mixcolumn transformation is performed on each column. Each column in the state matrix is multiplied by a fixed matrix defined in AES standard. Following diagram is taken from FIPS-197 document:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figure 2.5: Mix Columns transformation equation

Here, $S_{0,c}$ is the first state element of nth column and similarly, $S_{3,c}$ is the last element of nth column.

So, basically, first element of output column will be computed as:

$$S'_{0,c} = 02.\ S_{0,c} \oplus 03.\ S_{1,c} \oplus S_{2,c} \oplus S_{3,c} \qquad (2.1.4)$$

To multiply with an element 02, we can simply do a conditional bitwise xor with 11011. The condition is, if the MSB of the state element is set.

For example, lets assume that $S_{0,c}$ = 8'hE4, then the multiplied output with 02 will be:

E4 = 1110 0100

E4<<1 = 1100 1000 (dropping MSB as galois multiplication is carry less)

Since the MSB is set, then

Output = {1100 1000} $\oplus$ {0001 1011}

Output = 1101 0011 = D3

Since, multiplication with three is simply defined as:

$\text{Output} = \text{StateElement} . (10 \oplus 01)$

$\text{Output} = \text{StateElement} .10 \oplus \text{StateElement}$

So we use the same 02 multiplier to multiply an element with 03 and then add the state element in the result.

For simplicity, We made this Verilog function:

```
function [7:0] mul_02;
  input  [7:0]  in_byte;
    begin
      mul_02 = { in_byte[6],
                 in_byte[5],
                 in_byte[4],
                 (in_byte[3] ^ in_byte[7]) ,
                 (in_byte[2] ^ in_byte[7]),
                 in_byte[1],
                 (in_byte[0] ^ in_byte[7]),
                 in_byte[7] } ;
    end
endfunction
```

Figure 2.6: Simplified x2 Galois multiplication number

2.1.5 Key Expansion

A new key is used in each AES round. For that, key expansion algorithm runs in parallel to generate round's key. Here is the algorithm taken from FIPS-197 document:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word temp
   i = 0
   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while
   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if

      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 2.7: Key expansion algorithm

Here in the algorithm:

- Rcon = [01 02 04 08 10 20 40 80 1b 36]
- SubWord is same as sub-bytes.
- RotWord is column wise rotation

In simple words, the first column of output key is produced by rotating the column, performing byte substitution on each byte and then adding Rcon. The subsequent columns are produced by simply xoring the output column with the current column.

2.2 Functional Description of GHASH

Galois counter mode (GCM), is a block cipher mode of operation to encrypt and authenticate data. It can be implemented on a hardware to achieve performance upto hundreds of gigabits per second as well as its software performance is also excellent.

GCM can use any block cipher mode of encryption but here we are focusing on AES counter mode as the block cipher of our choice. The reason is, algorithm for AES counter mode of encryption and decryption is same i.e. the same hardware/software implementation can be used to encrypt or decrypt information.

GCM can be simply defined as:

$$(T,C) = GCM(K, IV, P, AAD) \qquad (2.2)$$

K = A secret key used by block cipher to encrypt data

IV = Initialization vector. It should be unique per key. For current implementation, we are using 96-bits of data

P = Plaint text. This is the data to be encrypted and authenticated. It can have maximum 239 number of bits

AAD = Additional Authentication Data. This data is to authenticated but not encrypted by GCM function.

T = Authenticated Tag that is used by the decryption block to verify authenticity of data. It can have length of 64bits to 128bits. As per GCM spec, Tag should be 128-bits wherever possible as it provides the best security[3].

C = Encrypted information whose length is same as of P

2.2.1 Details of GCM

Following equations are taken from GCM specification [3] and define the comprehensive algorithm of GCM encryption and authentication:

Initialization steps:

Hashkey or H $= E(K,0^{128})$ (2.2.1.1)

$Y0 = IV[95:0] \,||\, 0^{31}1$ (2.2.1.2)

Encryption and authentication equations:

$Y_{i+1} = incr(Y_i)$ for $i = 1,2....n$ (2.2.1.3)

$C_i = P_i + E(K,Y_i)$ for $i = 1,2....n$ (2.2.1.4)

$T = MSB_t(GHASH(H,A,C) + E(K,Y_0))$ (2.2.1.5)

In above equations:

- IV is initialization vector
- 0^{31} means append 31 zeros
- || indicates concatenation operation
- Incr() is increment function
- '+' indicates Galois addition which is logical Xor
- E() indicates the ecrpytion function which is AES counter mode
- GHASH is the hashing algorithm used for authentication
- MSB_t emphasize on the fact that we need to take most significant t-bits from hash function

In simple words, we initialize the system whenever there is a new key and new IV. Firs, we need to prepare hashkey which is simply the output of AES encryption. In this encryption, input key to the system is used as key and data is all zeros.

Second part of initialization is preparation of encryption vector. If IV is 96-bits then we simply append 31 zeros and a 1. In Verilog syntax, it can described as:

$$Y_0 = \{IV, \quad 31'd0, \quad 1'b1\} \qquad (2.2.1.6)$$

Once the initialization steps are done, system can accept input for encryption or AAD. As per the algorithm, AAD should preceded by encryption data. From above equation [ADD REFERENCE] its clear that actual input to the encryption block is encryption vector Y and user data is xored with the encryption result. This is the heart of GCM encryption and that's the reason that same hardware can be used for encryption and decryption.

The result of encryption is succeeded by authentication. The details of authentication is discussed in the following section.

2.2.2 Details of GHASH algorithm

GHASH algorithm is simply described by following equation:

$$X_{i+1} = (Input + X_i).H \qquad (2.2.2.1)$$

To explain further, the above equation (2.2.2.1) is depicted below:

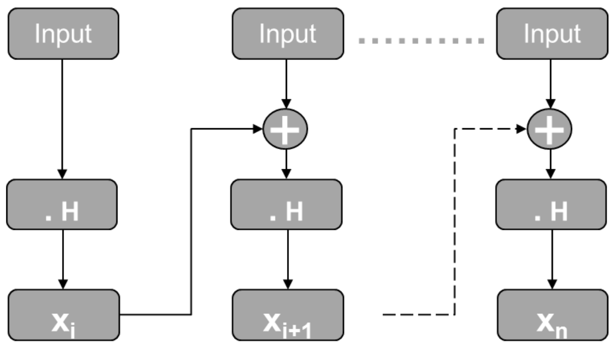


Figure 2.8: GHASH Data flow

From above equation (2.2.2.1) and diagram *Figure 2.8: GHASH Data flow*, it is clear that the 128-bit Xor operation and 128-bit Galois multiplication are part of feedback loop and limits the performance of GHASH algorithm.

Details for Galois Field multiplication are explained in details in the following section.

2.2.3 GF(2128) Multiplication

From above mentioned tag equation (2.2.2.1), its clear that Galois multiplication is an essential part of Hashing algorithm. Referring from GCM specification [3] that multiplication and addition possess algebraic properties like Commutativity, associativity and distributivity.

Unlike base-10 addition and multiplication, Galois multiplication and addition are carry less and doesn't contribute in bit growth. That's why addition and subtraction of two polynomials shows same behavior. Both are easily implementable by a xor gate. Note that, due to carry less operation, Galois filed operations are more suitable for hardware.

The algorithm for Galois multiplication is similar to base-10 multiplication i.e. the primary element is an 'and' gate. To avoid bit growth, we always divide the product by a particular polynomial. In GCM spec, the polynomial used for GF 2^{128} is mentioned below:

$$R = (11100001 \ || \ 0^{120})^2$$

Above equation [ADD REFERENCE] is degree 128 polynomial in which most significant 8 bits are 11100001 and least significant 120-bits are all zero.

The algorithm for GF 2^{128} is mentioned in GCM spec[3]. Here is the pseudocode for the algorithm:

Suppose, Z = X.Y where X,Y and Z belongs to GF 2128

```
Z ← 0, V ← X
for i = 0 to 127 do
   if $Y_i = 1$ then
      Z ← Z $\oplus$ V
   end if

   if $V_{127} = 0$ then
      V ← rightshift(V)
   else
      V ← rightshift(V) $\oplus$ R
   end if
end for

return Z
```

Figure 2.9: 128-bit Galois Multiplication function taken from GCM Specifications

2.2.4 Interface Diagram of GCM Cipher

Following is the block diagram for GCM Cipher. The GCM cipher is capable to encrypt and authenticate data at a data rate of 128-bits/cycle. Here are some information about the i/os:

1. Clk: is the only clock input. It decides the overall performance of the block and the frequency depends on device and technology used for hardware synthesis
2. Reset: GCM cipher controller works on asynchronous negative edge triggered reset
3. Data: data is 128-bit input which is supposed to be authenticated and/or encrypted
4. Key: the cipher block works using a 128-bit key which is direct input to AES GCM Encryptor
5. New_key: is a one shot key valid signal which notify the GCM controller about presence of new key.
6. Last block: this signal should be asserted by the signal only when the last block is being input to the GCM cipher. This helps in marking the data words present input pipeline.
7. Data_valid: The designed cipher block is capable to support data throttle by the driver. This signal specifies the validity of data. User can control this signal to throttle input data words
8. Type_selector: This is a 3-bit one hot encoded signal which determines the type of data being driven by the user. Following is the encoding used in GCM cipher block:
    a. 3’b001: payload
    b. 3’b010: IV (initialization vector)

c. 3'b100: AAD (Additional Authentication data) the data which should only be authenticated and not encrypted by the design

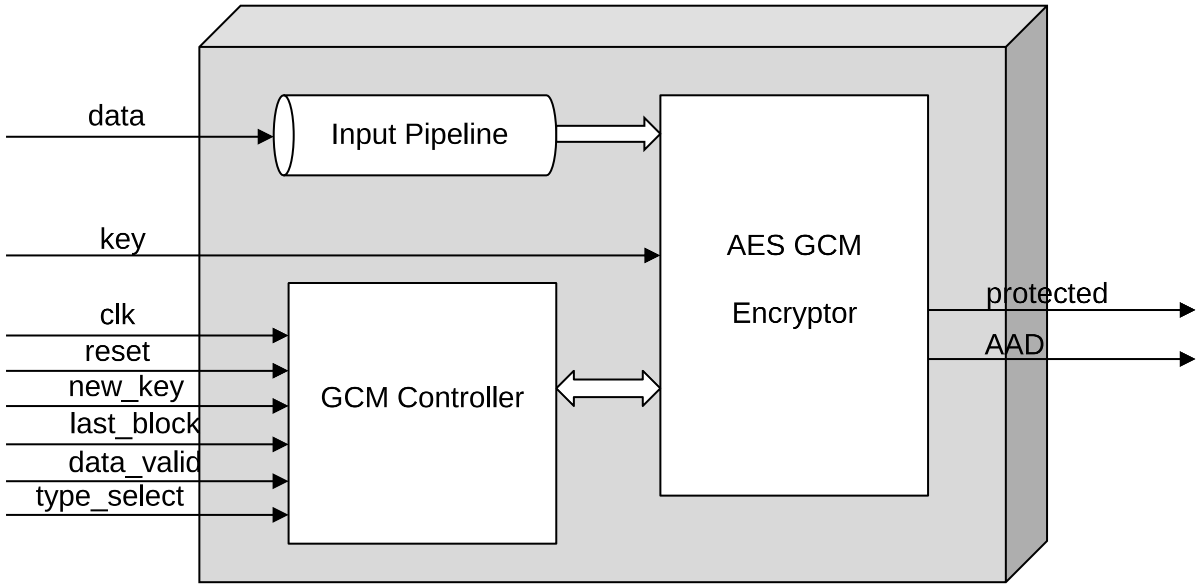


Figure 2.10: AES-GCM I/O Diagram

9. Protected Data: This the output protected data, 128-bits wide

10. AAD marker: Tell the next connecting block about the type of output data. It can an encrypted and authenticated data or only an authenticated data

3 Problem Description

As per the algorithm of AES-GCM encryption and authentication, the authentication part appears to be the one that limits the overall performance. In contrast to Authentication part, AES Encryption can be deep pipelined and achieves the performance matching to what is required in high performance applications like Ethernet and fiber optic communications.

For the authentication, GHASH algorithm is suggested in AES-GCM Specifications [3]. GHASH algorithm can be explained by the following simple equation:

$$X_{i+1} = (\text{Input} + X_i).H \qquad (3)$$

Here X_i is the previous authenticated tag and H is the hash key. So basically, GHASH is a Galois addition-multiplication. In digital design, its simplified architecture is a digital X-Or and Galois multiplication. The straight forward implementation of AES-GCM requires excessive amount of FPGA area and the performance is well below par. Our experimental results showed that AES-GCM could consume more than 1.7 times (rough estimation) the area of Ethernet core.

Since it's a carry less multiplication, the overall gate delay is much less as compare to binary multiplication, still, there is a vital need for pipelining the multiplier. As reader can see the fact from the previous equation that the equation used as-it-is cannot be pipelined due to the feedback path in the equation i.e. each previously computed tag is used as an input to the X-Or operation used in the following tag computation.

3.1 Previous Work

Previous authors has attempted to work in the following dimensions to improve the performance of AES-GHASH algorithm

1. Authors has suggested different techniques to split the feedback path in GHASH algorithm that improves the performance of overall algorithm
2. Authors has suggested different multiplication algorithms to improve the performance for multipliers that certainly improves the throughput. The known methods include, Kartsuba algorithm, bit-parallel multiplier, Masterovito and Fan-Hasan Subquadratic multiplier.
3. Improving platform specific architectural bottlenecks. One example is, the adding the redundant parallel registers to reduce clock tree.

In [4], the authors has presented an FPGA based implementation of AES-GCM using a bit-parallel multiplier. Authors added four pipeline stages in the multiplier to maximize throughput. They also introduced redundant register technique to address the high fan-out issue.

In another paper, [5], authors addressed the complexity of multiplication with Karatsuba algorithm that results in area efficient implementation of GHASH algorithm. They also mentioned the tradeoff between speed and area of multipliers by comparing karatsuba algorithm with straight-forward implementation of multipliers. Their implementation and results are based on Xilinx Virtex-4 FPGA. In the same paper, authors presented composite field AES encryption approach to match the throughput of GHASH with AES-encryption

In a thesis with title Parallel Multiplier Designs for the Galois/Counter Mode of Operation[7], author has presented an excellent comparison of different Galois Field multiplication algorithms. He presented comparison between, Karatsuba, Fan-Hasan and Mastervito multipliers and suggested that Karatsuba algorithm results in best area implementation while Fan-Hasan multiplier gives best throughput at the significant cost of FPGA LUTs. Here, We am pasting the summary of his thesis:

Multiplier	Delay (ns)	4 input LUT	Slices	Utilization (%)
FH [19]	6.125	6616	3781	8.96
KA [20]	6.637	6136	3518	8.34
Mastrovito [21]	4.583	14199	8163	19/35

Table 3.1: Area-Delay Comparison between different Galois Field Multipliers (taken from [7])

Another author presented a 130nm ASIC based implementation of AES-GCM in [High speed hardware architecture for authenticated Encryption mode GCM]. In this paper, author presented an area and performance constrained solutions for AES-GCM. The performance based solution yields 42Gbps throughput with 297K gates. In this paper, author presented a unique way to break the feedback path in GHASH algorithm that allows deep pipelining of Galois multiplication. The algorithm uses precomputed powers of hash key that helps in pipelining removing the feedback path. The only bottleneck of the design is, algorithm needs an advance information about the size of Ethernet packet that actually decides which GHAH key power to use to authenticate the data chunk.

In another research work[12], author presented a different approach for multiplication. Authors used polynomial based multiplication approach and then performed modular reduction on multiplied polynomials to yield reduced number of overall multiplications.

Authors also presented an ISA instruction set based architecture of AES implemented on Intel's platform.

A less relevant but interesting research is done on using the distributed key concept across parallel cores of AES-GCM algorithm is presented in [10]. This approach saves area but is only suitable for slow key changing networks like VPN(virtual private networks). Still in this approach, Karatsuba algorithm is used across all parallel multipliers.

The closest match and the motivation of this thesis is the work done in [6]. Authors presented an area and performance based implementation of AES-GCM targeting 100Gbps Ethernet. In this paper, authors presented an architecture based on four parallel AES-GCM cores each supporting a 128-bit interface for AES encryption. Each core provides an approximate throughput of 25Gbps and thus resulting an overall throughput of 100Gbps. In this paper, authors used, Karatsuba multiplier for GHASH algorithm implementations.

Following table gives comparison between previous implementations and our implementation:

	Slices	Equivalent LUTS	Throughput (Gbps)	Hardware efficiency(Mbps)	Implementation type
Ours	----	4,852	49.3	10.16	
[4]	9481	18,962	32	3.38	Ghash_4
[4]	10,756	21,512	40	3.72	Ghash_f
[11]	----	----	36.48		
[14]			15.36		
[5]			20.61		
[16]		9,405	15.38	1.63	
[8]		12,558	34.6	2.7	Sequential
[6]	8,223	16,446	25	3.29	4-parallel GCM cores

Table 3.2 Previous works in contrasts with our results

4 Implementation Details

Our focus is on improving the overall area and overall delay of GHASH. We implemented AES counter mode with the simplest implementation by unrolling all the rounds in AES to provide maximum throughput. Even that simplest implementation proved to be enough to challenge the GHASH performance. We didn’t focus on AES further. We used Altera’s Stratix IV FPGA as a platform for experiments. First we implemented the multiplication algorithm as-it-is as mentioned in AES-GCM specifications.

Here is the multiplication algorithm taken from GCM specifications[3]:

```
Z ← 0, V ← X
for i = 0 to 127 do
   if $Y_i$ = 1 then
      Z ← Z ⊕ V
   end if

   if $V_{127}$ = 0 then
      V ← rightshift(V)
   else
      V ← rightshift(V) ⊕ R
   end if
end for

return Z
```

Figure 4.1: Multiplication approach taken from GCM spec

4.1 Implementation of GHASH Multiplier

We first tried to implement the multiplier present in GCM specs, without modifying it at all. To measure the direct performance(without pipeline stages), We registered the input of 128-bit multiplicand and multiplier and the output of the multiplier to measure register to register delays.

Following are the synthesis results:

Device	ALUTs	FFs	Fmax
Stratix IV	8,523	384	52MHz

Table 4.1: Synthesis results for direct implementation of GF Multiplier

4.1.1 Karatsuba algorithm

From Wikipedia, Karatsuba algorithm was invented by Anatoly Karatsuba in 1962. It's a method that results in fast multiplication while keeping the overall area, minimum.

Consider a 128-bit binary multiplier. It requires 128-bit x 128-bit single bit multiplier i.e. 16,384 single bit multipliers. Now suppose A and B are two 128-bit numbers to be multiplied. We can represent A and B as follows:

$A = A[127:64].2^{64} + A[63:0]$ (4.1.1.1)

$B = B[127:64].2^{64} + B[63:0]$ (4.1.1.2)

$Z = A \times B$

$Z = (A[127:64].B[127:64].2^{128} + A[63:0].B[127:64].2^{64} + A[127:64].B[63:0].2^{64} + A[63:0].B[63:0])$ (4.1.1.3)

The above equation (4.1.1.3) requires 4*64*64 = 16,384 single bit multipliers. Let's assume:

$Z_0 = A[63:0].B[63:0]$ (4.1.1.4)

$z_1 = A[63:0].B[127:64].2^{64} + A[127:64].B[63:0].2^{64}$ (4.1.1.5)

$z_2 = A[127:64].B[127:64].2^{128}$ (4.1.1.6)

Karatsuba realized that above multiplication can be done with 3 multipliers at the cost of few extra additions. The term z_1 can be computed as:

$z_1 = (A[127:64] + A[63:0]) . (B[127:64] + B[63:0]) – z_2 – z_1$

Hence, number of Single bit multipliers required by z_0, z_1 and z_2 are:

z_0 requires 64x64 = 4096 single bit multipliers

z_1 requires 65x65 = 4225 single bit multipliers

z_2 requires 64x64 = 4096 single bit multipliers

Hence we need a total of 12,417 single bit multipliers instead of 16,384 single bit multipliers at the cost of few extra additions.

Above equation represent level-1 Karatsuba multiplier. Each multiplication component can be further discretized to improve delay and area until the extra additions required by Karatsuba starts dominating the overall area. After that Karatsuba level, we will start seeing increase in overall area.

Also, note that now critical delay is a function of 65x65 bit multiplication instead of 128x128 bit multiplication.

We tried multiple levels of Karatsuba algorithm. In the following table, We am summarizing the results:

Device	ALUTs	ALUTs difference	FFs	Fmax(MHz)	Method	KA level
Stratix IV	8523	-	384	52	Direct Mult	-
Stratix IV	6592	1931	384	93	KA	1
Stratix IV	5322	1270	384	148	KA	2
Stratix IV	4628	694	384	157	KA	3
Stratix IV	4574	54	384	160	KA	4
Stratix IV	5201	-627	384	172	KA	5
Stratix IV	5293	-92	384	165	KA	6
Stratix IV	5588	-295	384	163	KA	7

Table 4.2: Implementation results for variants of KA multiplier

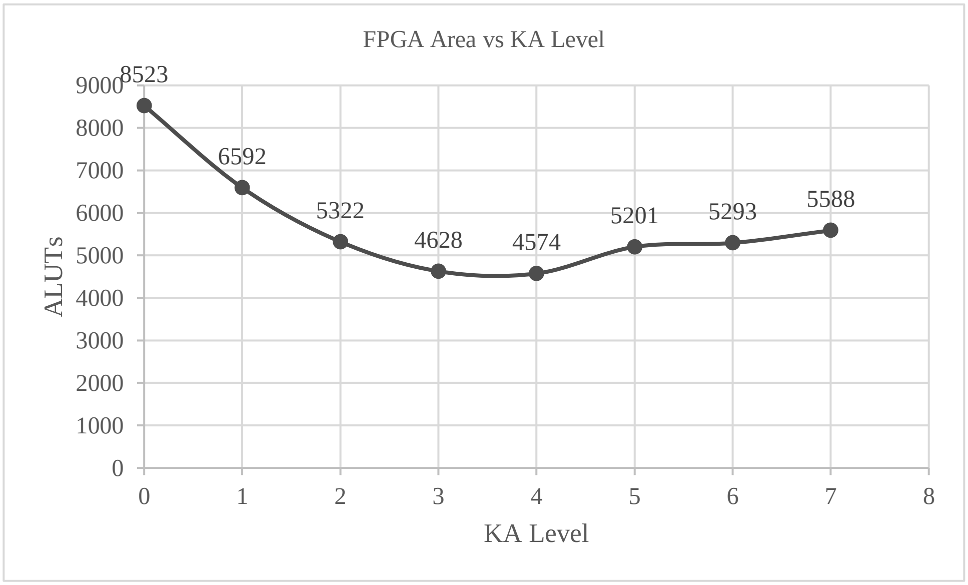


Figure 4.2: FPGA area consumption for different levels of KA multiplier

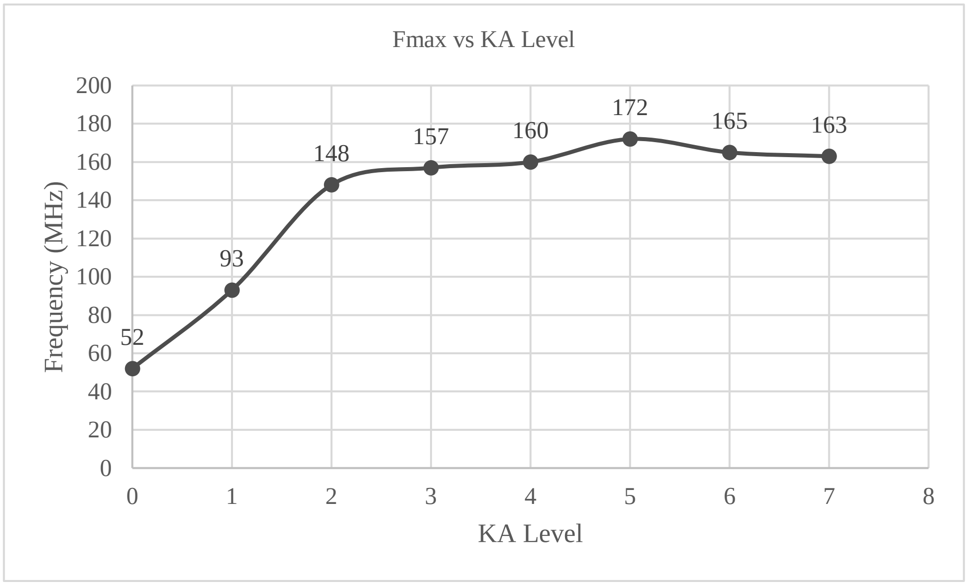


Figure 4.3: Maximum operation freq for different levels of KA multiplier

Following are some important information extracted from above graphs:

- It can observed that moving from direct multiplication to level-1 Karatsuba saved almost 2k ALUTs
- From KA-level 4 to KA level-5 we start seeing the effect of extra additions. They started dominating the area
- In this scenario, KA level-4 resulted in most optimum implementation in speed and area constrained approach.
- Maximum running frequency increased more than 3 times moving from direct multiplication to KA level-4

Hence We decided to use Karatsuba level-4 multiplication for GHASH implementation

4.1.2 Adding pipeline stages in KA multiplier

From the above section, we achieved significant improvements in performance i.e. from 52 MHz(direct implementation) to 160MHz(KA level-4). Still, 160MHz is well below par. Considering a 128-bit interface, our desired throughput is atleast 50Gbps from each core. Hence our multiplier should run at atleaset at 391MHz.

To further improve the frequency, we need to add pipeline stages in the multiplier. As discussed earlier, the bottleneck in pipelining is the feedback path present between adder and multiplier.

In [4], We found this interesting technique to break the feedback path.

Before We start explaining this technique, lets recall that, GHASH is computed by following equation:

$$X_{i+1} = (Input + X_i).H$$

This technique require us to compute multiple powers of hash keys in advance. The powers of hash key depends on the desired level of pipelining. Apart from pre-computed hash keys, the algorithm requires an advance knowledge of end of packet. The advance information, again depends on the level of pipelining. So, let's assume our multiplier has 5 pipeline stages, we need H^5, H^4, H^3, H^2 and H^1. Also, algorithm needs to know the end of packet, 5 cycles earlier. This earlier information is not difficult to implement. We delayed(added 5 additional pipe stages) the input 5 cycles before passing it to AES-GCM module.

The following table explains the data flow:

clk	Inp1	Inp2	output	Next inp
0	I_0	H^5	-	
1	I_1	H^5	-	
2	I_2	H^5	-	
3	I_3	H^5	-	
4	I_4	H^5	-	
5	I_5	H^5	$I_0 \text{ x } H^5$	$I_5 + I_0 \text{ x } H^5$
6	I_6	H^5	$I_1 \text{ x } H^5$	$I_6 + I_1 \text{ x } H^5$
7	I_7	H^5	$I_2 \text{ x } H^5$	$I_7 + I_2 \text{ x } H^5$
8	I_8	H^5	$I_3 \text{ x } H^5$	$I_8 + I_3 \text{ x } H^5$
9	I_9	H^4	$I_4 \text{ x } H^5$	$I_9 + I_4 \text{ x } H^5$
10	I_{10}	H^3	$[I_5 + I_0 \text{ x } H^5]\text{x } H^5$	$[I_5 + I_0 \text{ x } H^5]\text{x } H^5 + I_{10}$
11	I_{11}	H^2	$[I_6 + I_1 \text{ x } H^5]\text{x } H^5$	$[I_6 + I_1 \text{ x } H^5]\text{x } H^5 + I_{11}$
12	I_{12}	H^1	$[I_7 + I_2 \text{ x } H^5]\text{x } H^5$	$[I_7 + I_2 \text{ x } H^5]\text{x } H^5 + I_{12}$
13			$[I_8 + I_3 \text{ x } H^5]\text{x } H^5$	
14			$[I_9 + I_4 \text{ x } H^5]\text{x } H^4$	
15			$[[I_5 + I_0 \text{ x } H^5]\text{x } H^5 + I_{10}] \text{ x } H^3$	
16			$[[I_6 + I_1 \text{ x } H^5]\text{x } H^5 + I_{11}] \text{ x } H^2$	
17			$[[I_7 + I_2 \text{ x } H^5]\text{x } H^5 + I_{12}] \text{ x } H^1$	

Figure 4.4: Example data flow for 5-staged pipelined GHASH module

Now, we need to xor the last five outputs:

$$\text{Final Hash} = [I_8 + I_3 \text{ x } H^5]\text{x } H^5 + [I_9 + I_4 \text{ x } H^5]\text{x } H^4 + [[I_5 + I_0 \text{ x } H^5]\text{x } H^5 + I_{10}] \text{ x } H^3 + [[I_6 + I_1 \text{ x } H^5]\text{x } H^5 + I_{11}] \text{ x } H^2 + [[I_7 + I_2 \text{ x } H^5]\text{x } H^5 + I_{12}] \text{ x } H^1$$

$$= I_8 H^5 + I_3 H^{10} + I_9 H^4 + I_4 H^9 + I_5 H^8 + I_0 H^{13} + I_{10} H^3 + I_6 H^7 + I_1 H^{12} + I_{11} H^2 + I_7 H^6 + I_2 H^{11} + I_{12} H^1$$

Rearranging:

$$\text{Hash} = I_0 H^{13} + I_1 H^{12} + I_2 H^{11} + I_3 H^{10} + I_4 H^9 + I_5 H^8 + I_6 H^7 + I_7 H^6 + I_8 H^5 + I_9 H^4 + I_{10} H^3 + I_{11} H^2 + I_{12} H$$

4.1.3 5-Staged Pipelined GHASH Architecture

Finally, We combined the techniques discussed in section 4.1.1 and 4.1.2 to obtain an area and speed optimized GHASH hardware. Following diagram shows the final picture of GHASH architecture:

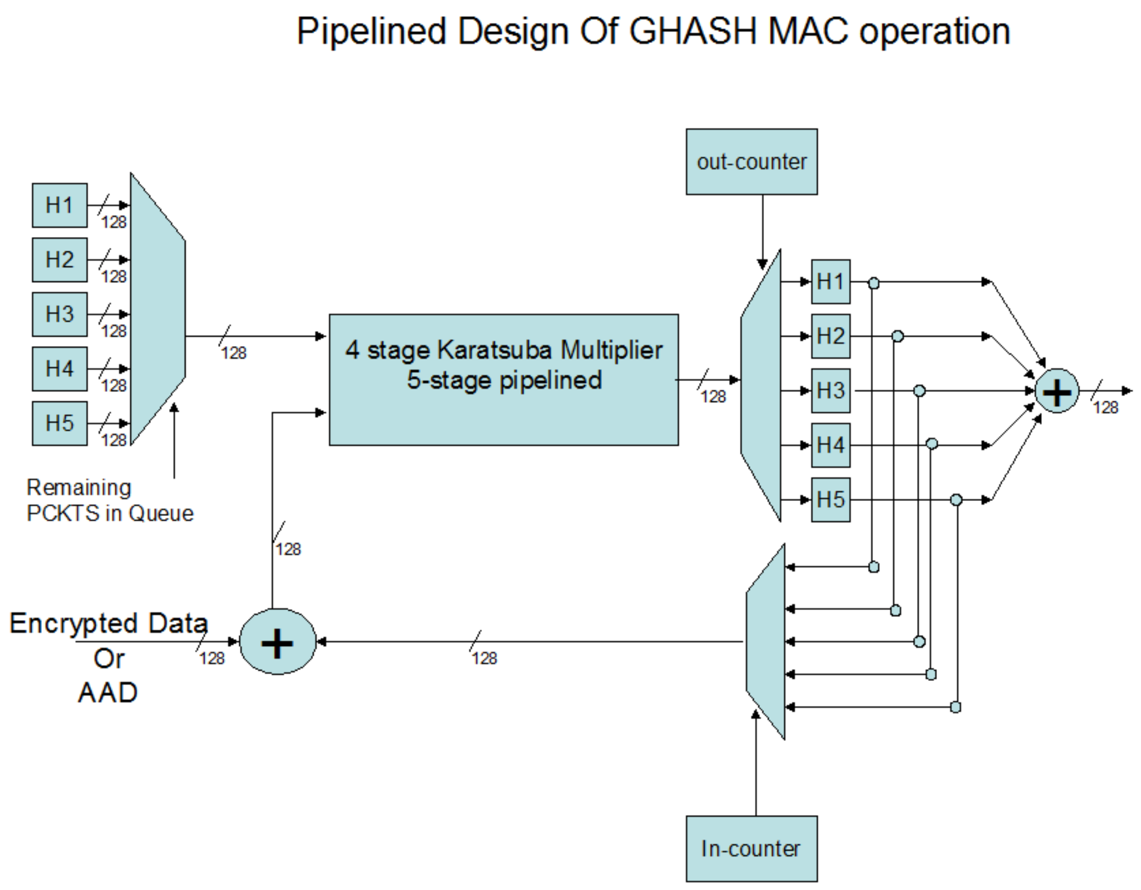

Pipelined Design Of GHASH MAC operation
H1
H2
H3
H4
H5
128
Remaining PCKTS in Queue
4 stage Karatsuba Multiplier 5-stage pipelined
out-counter
In-counter
Encrypted Data Or AAD

Figure 4.5: Block diagram for 5-staged pipelined GHASH module

The above block diagram depicts a five stage pipelined GHASH architecture. H1, H2 till H5 are the powers of hash keys. As discussed in section 4.1.2, we have to maintain the remaining words count and based on that we use the nth power of hash key. All adder operations in above diagram are xor functions.

The above architecture is good enough to be implemented directly on FPGA platform. However, during the synthesis of design, We experienced high fanout on some of the stages inside multiplier. For that, We added redundant registers and forced compiler to bypass the ‘remove redundant logic’ check on those particular registers.

5 Experimental Details

To implement the architecture shown in *Figure 4.5: Block diagram for 5-staged pipelined GHASH module*, we selected Verilog as the hardware description language. The design for KA multiplier is parameterized to support 1-5 cycles deep pipelining.

5.1 AES GCM Test Vectors

To verify our implementation, we used golden test vectors present in GCM-Specification. There are total 18 test cases present in GCM specs covering from encryption and authentication to ‘only authentication’ test scenarios. Each test case is provided with intermediate stage results that helped in debugging. Also, each test case is provided with initialization vector and a key. Thus, test vectors provided with GCM spec covers all aspects of functional validation.

5.2 Test Bench Setup

The testbench for AES GCM Encryptor is based on System Verilog’s class ranadomizer to create constrained randomized scenarios. The testbench is designed considering that the encryptor will be used in a packet based environment.

Following is the simplified block diagram of testbench:

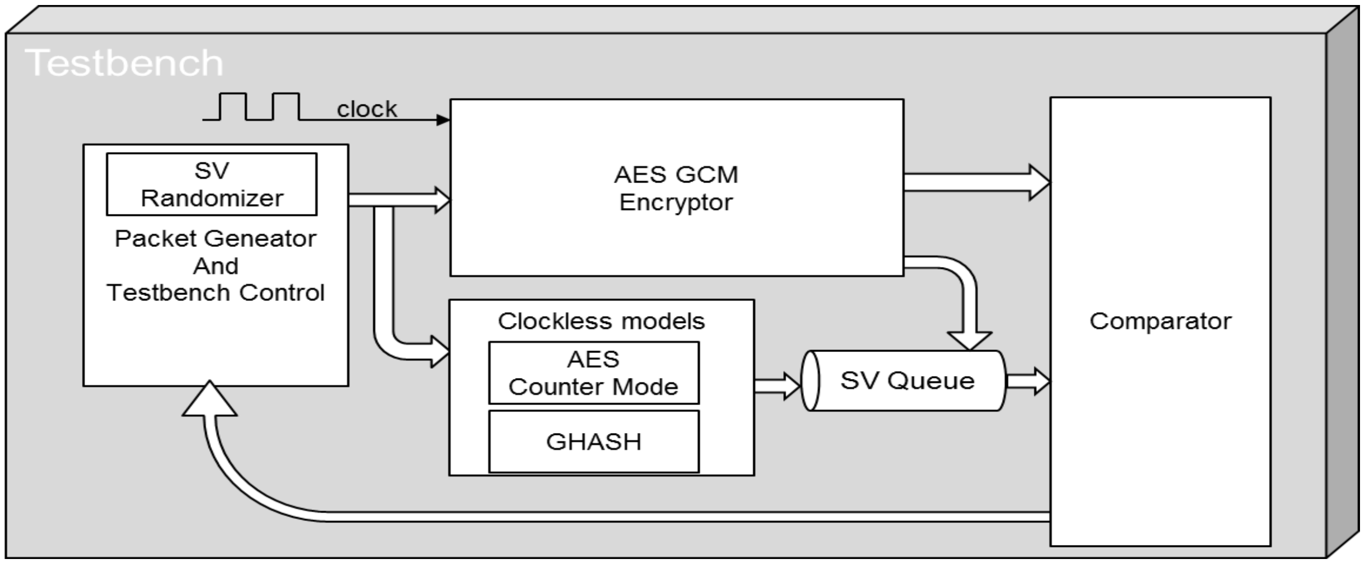


Figure 5.1: Testbench Setup

Considering a packet (an Ethernet packet for example) which has the header part (for example source address, Destination address, length field and other header information fields). Those fields should not be encrypted so that the router can recognize the packet type and route it to its destination. However, in a secure network environment it's highly recommended that the header fields should be authenticated so that the packets known sources should be forwarded to the destination and to make sure that the packet header fields are untouched.

The testbench generates number of packets determine by the user. All the inputs to the GCM encryptor like 128-bit key and data are fully random. To cross check the output of the core, we created the following clockless models to verify the output of the encryptor:

- 128-bit base AES Encryptor
- Galois Counter mode for AES
- GHASH computer

- AES GCM encryptor

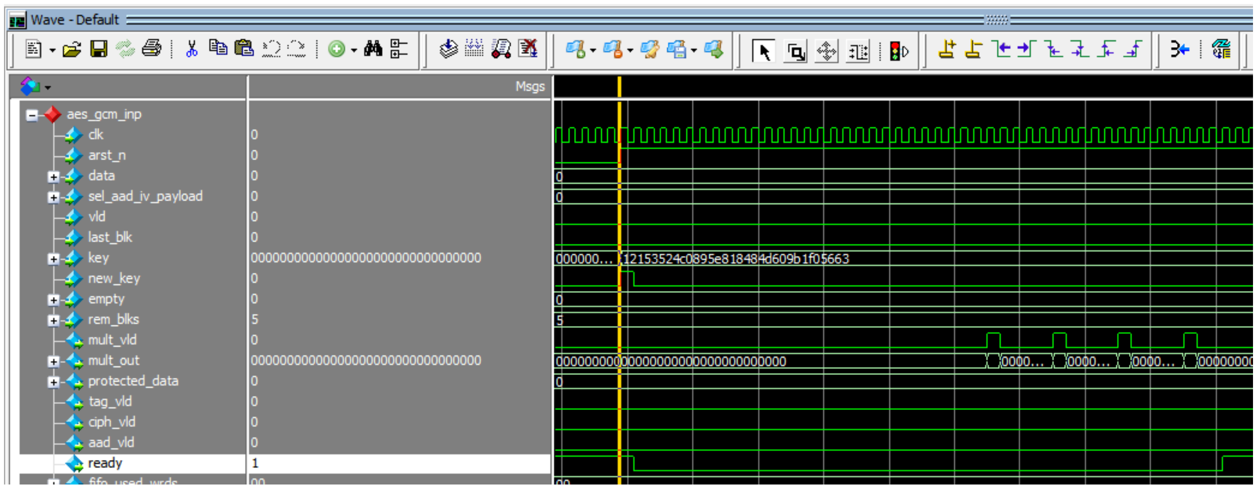
The testing procedure is as follows:

- A randomized hash key is generated
    - Calculate exponential powers of hash key
- For n number of scenarios:
    - Determine number of words in this packet using SV constraint randomizer
- First send the header words
    - Select aad_iv_payload = 3’b100
    - This data will not be encrypted
    - Send the data to clockless or behavioral models
    - Push the output of behavioral model to SV queue
- When the word count == header words
    - Send the initialization vector by selecting aad_iv_payload = 3’b010
    - This data will be encrypted and authenticated
    - Send the data to clockless or behavioral models
    - Push the output of behavioral model to SV queue
- For other words:
    - Set aad_iv_payload = 3’b001
    - This data will be encrypted and authenticated
    - Send the data to behavioral models
    - Push the output of behavioral model to SV queue

5.3 Simulation Results

For simulation, we used Mentor Graphic's Modelsim 10.1a student edition.

Following is the snapshot taken from modelsim's wave viewer. It shows the initialization state of system. Whenever a new key is input to the system, it needs few cycles to compute the powers of hash key. That's why, you can see that the ready signal goes low when the user asserts the new key input signal. The ready signal gets asserted again once all the required powers of hash key are computed successfully.



```
aes_gcm_inp
  clk                0
  arst_n             0
  data               0
  sel_aad_iv_payload 0
  vld                0
  last_blk           0
  key                00000000000000000000000000000000
  new_key            0
  empty              0
  rem_blks           5
  mult_vld           0
  mult_out           00000000000000000000000000000000
  protected_data     0
  tag_vld            0
  ciph_vld           0
  aad_vld            0
  ready              1
```

Figure 5.2: Sim results for initial state of GCM encryptor

Overall latency of system is 32 cycles @ 390MHz, which is around 81ns. This latency is quite reasonable as the latency for optical fiber transmission systems is much higher as compare to the latency of our encryptor. Following pic shows the cycles taken by our GCM encryptor to compute Tag. Also, it shows the latency between the input data and the encrypted and/or authenticated data:

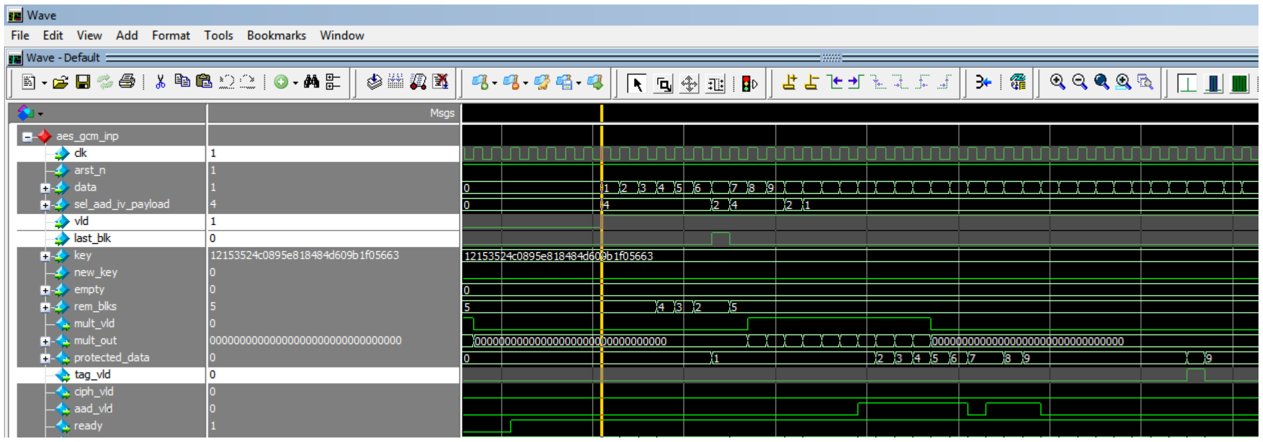


Figure 5.3: Sim results depicting overall latency of AES-GCM Encryptor

In the last, we want to illustrate simulation results for KA Multiplier. The multiplier has 4 Karatsuba stages and its 5 cycles deep pipelined. The 5th register stage is used at the output of polynomial divisor as discussed in section 2.2

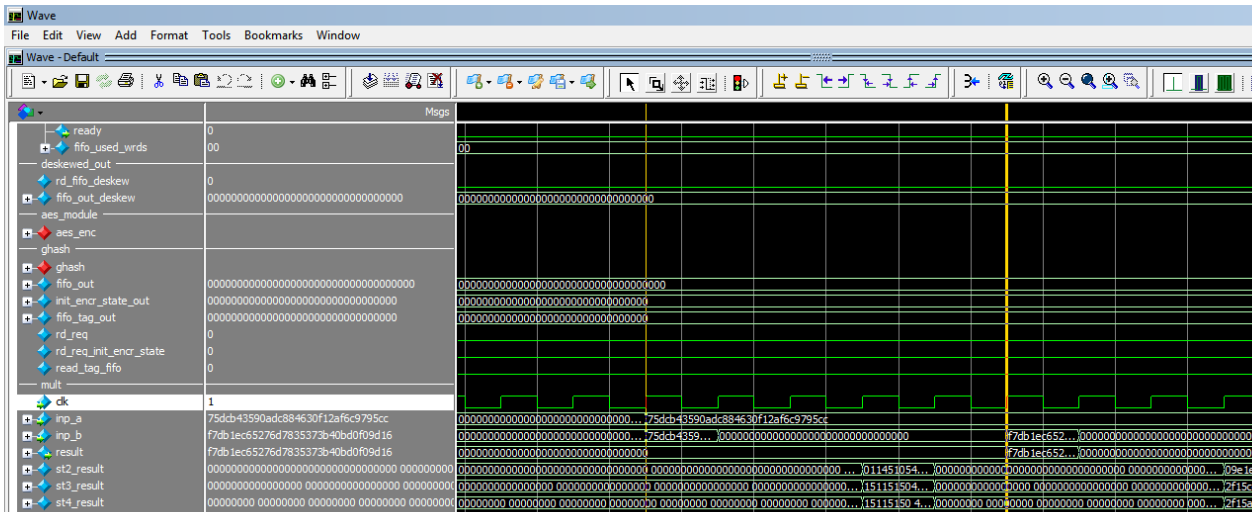

Wave
File Edit View Add Format Tools Bookmarks Window
Wave - Default
Msgs
ready 0
fifo_used_wrds 00
deskewed_out
rd_fifo_deskew 0
fifo_out_deskew 00000000000000000000000000000000
aes_module
aes_enc
ghash
ghash
fifo_out 00000000000000000000000000000000
init_encr_state_out 00000000000000000000000000000000
fifo_tag_out 00000000000000000000000000000000
rd_req 0
rd_req_init_encr_state 0
read_tag_fifo 0
mult
clk 1
inp_a 75dcb43590adc884630f12af6c9795cc
inp_b f7db1ec65276d7835373b40bd0f09d16
result f7db1ec65276d7835373b40bd0f09d16
st2_result
st3_result
st4_result

Figure 5.4: Sim results for GHASH multiplier

5.4 Synthesis Results

We selected Altera’s FPGA platform based on the availability. We selected Stratix IV EP4S40G2 device. This devices is the smallest device on Stratix IV FPGA Family and it’s designed on 40nm process. We used Altera’s Quartus II logic synthesizer. We synthesized complete AES-GCM Encryptor to make sure that our RTL is ready to be used as an FPGA IP core. Following are the synthesis results:

Parameters	**Count**	**Device Utilization**
LUTs	13,990	14%
Registers	9,755	5%
Memories(M9K)	8	1%
Latency	32 cycles	
Fmax	393MHz	

Table 5.1: AES-GCM Synthesis results

6 Conclusion and Future Work

6.1 Conclusion

In contrast with available IP core solutions available for AES GCM in market and the research work done to optimize the implementation on FPGA, our unique application targeted hardware architecture offers a challenging solution. Not only our solution is matching the required bandwidth for 100Gbps Ethernet, it also saves significant area as compare to other 4-parallel cores based solutions.

6.2 Future Work

With the increase in demand in high bandwidth internet connections, companies are investing a lot of resources on 5G internet. TeraBit or TBe will be the next standard for Ethernet. The target bandwidth is 200Gbps and 400Gbps. As a future work, we can expand this idea and work and propose a security solution for 200Gbps and 400Gbps Ethernet standards.

Bibliography

[1] Cisco whitepaper, The zeta Era. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[2] Federal Information Processing Standard (FIPS 197) Advanced Encryption Standard (AES) URL: csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] The Galois counter mode of operation (GCM) NIST Standard URL: csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

[4] Tianshan Chen, Design and Efficient FPGA Implementation of Ghash Core for AES-GCM, 2010 IEEE

[5] Gang Zhou, Efficient and High-Throughput Implementations of AES-GCM on FPGAs, IEEE FPT 2007

[6] Luca Henzen, FPGA Parallel-Pipelined AES-GCM Core for 100G Ethernet Applications, IEEE 2010

[7] Pujan Patel, Parallel Multiplier Designs for the Galois/Counter Mode of Operation, Thesis done at University of Waterloo, Canada, 2008

[8] Akashi Satoh, High- Speed Hardware Architectures for Authenticated Encryption Mode GCM, IEEE 2013

[9] Akashi Satoh, High-Speed Parallel Hardware Architecture for Galois Counter Mode, IEEE 2012

[10] Karim M. Abdellatif, Improved Method for Parallel AES-GCM Cores Using FPGAs, IEEE 2013

[11] Gang Zhou, Improving Throughput of AES-GCM with Pipelined Karatsuba Multipliers on FPGAs, Springer, 2014

[12] Krzysztof Jankowski, Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions

[13] Intel Altera's Stratix IV Device handbook Volume 4. URL: https://www.altera.com/literature/hb/stratix-iv/stx4_5v4.pdf

[14] Stefan Lemsitzer, Multi-Gigabit GCM-AES architecture optimized for FPGAs,

[15] Dworkin, Block Cipher Modes of operation. NIST Special publication 800-38 A(2001)

[16] Yang Lu, The Research and Efficient FPGA Implementation of Ghash Core for GMAC

[17] "802.1AE - Media Access Control (MAC) Security," Draft 3.5, Jun. 2005. http://www.ieee802.org/1/pages/802.1ae.html

[18] J. Viega, The use of Galois/Counter mode in IPSec Encapsulating Security Payload, http://www.faqs.org/rfcs/rfc4106.htm

[19] H. Fan and M.A. Hasan. A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields. IEEE Transactions on Computers

[20] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. IEEE Transactions on Computers

[21] E.D. Mastrovito. VLSI Architectures for Computation in Galois Fields, PhD Thesis. Dept. of Electrical Eng., Linkoping Univ., Sweden