Parallel Multiplier Designs for the Galois/Counter Mode of Operation

by

Pujan Patel

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, 2008

Abstract

The Galois/Counter Mode of Operation (GCM), recently standardized by NIST, simultaneously authenticates and encrypts data at speeds not previously possible for both software and hardware implementations. In GCM, data integrity is achieved by chaining Galois field multiplication operations while a symmetric key block cipher such as the Advanced Encryption Standard (AES), is used to meet goals of confidentiality. Area optimization in a number of proposed high throughput GCM designs have been approached through implementing efficient composite Sboxes for AES. Not as much work has been done in reducing area requirements of the Galois multiplication operation in the GCM which consists of up to 30% of the overall area using a bruteforce approach. Current pipelined implementations of GCM also have large key change latencies which potentially reduce the average throughput expected under traditional internet traffic conditions. This thesis aims to address these issues by presenting area efficient parallel multiplier designs for the GCM and provide an approach for achieving low latency key changes. The widely known Karatsuba parallel multiplier (KA) and the recently proposed Fan-Hasan multiplier (FH) were designed for the GCM and implemented on ASIC and FPGA architectures. This is the first time these multipliers have been compared with a practical implementation, and the FH multiplier showed note worthy improvements over the KA multiplier in terms of delay with similar area requirements.

Using the composite Sbox, ASIC designs of GCM implemented with subquadratic multipliers are shown to have an area savings of up to 18%, without affecting the throughput, against designs using the brute force Mastrovito multiplier. For low delay LUT Sbox designs in GCM, although the subquadratic multipliers are a part of the critical path, implementations with the FH multiplier showed the highest efficiency in terms of area resources and throughput over all other designs. FPGA results similarly showed a significant reduction in the number of slices using subquadratic multipliers, and the highest throughput to date for FPGA implementations of GCM was also achieved. The proposed reduced latency key change design, which supports all key types of AES, showed a 20% improvement in average throughput over other GCM designs that do not use the same techniques. The GCM implementations provided in this thesis provide some of the most area efficient, yet high throughput designs to date.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Anwar Hasan, for his encouragement and support throughout the last two years. I am very grateful for the flexibility he has given me for this work. I would also like to thank Professor Rosenberg and Professor Aagaard for reviewing this work; Aziz Alkhoraidly and Siavash Bayat-Sarmadi for their helpful advice and entertaining conversations; and last but not least my Family for their never ending supplies of patience and smoothies.

Contents

List of Tables

List of Figures

List of Abbreviations

AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
FH	Fan-Hasan (Subquadratic Parallel Multiplier)
FPGA	Field Programmable Gate Array
GCM	Galois/Counter Mode of Operation
GF	Galois Field
IEEE	Institute of Electrical and Electronics Engineers
IPSec	Internet Protocol Security
IV	Initialization Vector
KA	Karatsuba Algorithm
LUT	Look Up Table
NIST	National Institute of Standards in Technology
OCB	Offset Code Book
PB	Polynomial Basis
Sbox	Substitution Box
TMVP	Toeplitz Matrix Vector Product
VLSI	Very Large Scale Integration

Chapter 1

Introduction and Motivation

Achieving security goals of data integrity and confidentiality for high speed network applications has been a difficult problem to solve. Applying security features to network traffic generally cannot be performed at link speeds because of high computational costs. A recently standardized authentication and encryption scheme, the Galois/Counter Mode of Operation (GCM), has been promising in its ability to deliver high speed software and hardware implementations not previously possible. GCM has a number of different properties such as the ability to process data both sequentially and in parallel that distinguishes it from other similar schemes. The IEEE and the NIST have both applied this mode of operation in a number of different applications ranging from, network, tape storage and link level security.

In GCM, authenticity of messages is met by universal hashing over Galois fields while confidentiality is met by a symmetric block cipher such as the Advanced Encryption Standard (AES). Efficient Galois field operations such as multiplication have been researched extensively for their applications in Elliptic Curve Cryptography and numerous implementations have been presented for the AES block cipher since its acceptance as a NIST Standard. GCM integrates this large body of work, and several high throughput hardware designs of GCM have been proposed in the literature [28, 31, 29, 39, 15].

The driving focus for many of these designs has been in modifying the AES block in order to achieve both area efficient and high speed datapaths. Not much work, however, has been done in reducing area requirements of the Galois multiplication operation in the GCM which consists of up to 30% of the overall area using a bruteforce Mastrovito multiplier. Since this type of multiplier has a low delay, for GCM designs using the composite Sbox, the multiplier is not a part of the critical path by a large margin. The larger delay of the composite Sbox means pipelined stages are unbalanced and result in wasted area resources. Subquadratic parallel multipliers provide a smaller area footprint in hardware and therefore provide an alternative to the brute force approach. Although the delay is higher for this class of multipliers, it is possible to balance the pipelined stages of the GCM in order to get more area efficient designs that do not effect the throughput significantly.

Besides area efficiency, there are other improvements possible for the hardware implementations of GCM. Internet Protocol Security (IPSec) uses the GCM in one of its mode of operations to help authenticate and encrypt packet data at the network layer (Layer 3 in the OSI network model). Since this protocol enables security operations at a lower level, hardware performance is an important factor to consider. In IPSec, a key change usually occurs in a single session based on timeouts or when a threshold is reached for data processed. Within this set limit, numerous packets could potentially be authenticated and encrypted without a key change occurring between a single link of communication, also known as a Security Association (SA). High speed hardware implementations of GCM would maintain hundreds to thousands of SA's at a time for IPSec. A large number of keys are maintained in memory and accessed when needed for performing security operations on different links of communication [35]. Key changes are expected to occur very frequently for these cases so latency for changing keys, especially for smaller packet sizes, can affect performance significantly. Current GCM designs in the literature have large key change latencies equivalent to the latency in the AES pipeline used, making the average throughput of these designs much lower then expected.

1.1 Contributions

The following list summarizes the work done in this thesis in order to address the current issues observed in GCM designs presented in the literature.

- Provide designs of subquadratic parallel multipliers for GCM
- Propose key schedule designs for GCM with low key change latency
- Implement the proposed GCM designs on FPGA and ASIC

The use of subquadratic Galois field multipliers in the GCM is the main contribution of this thesis. This class of multipliers helps realize area efficient GCM designs for both ASIC and FPGA architectures without compromising throughput rates significantly. Two subquadratic space complexity multipliers, the Karatsuba multiplier (KA) [24] and a recently proposed multiplier by Fan and Hasan (FH) [6], were the multipliers designed for the GCM. Although the multipliers are designed specifically for the GCM, the techniques used in this thesis may be generalized for other applications as well. The FH multiplier is theoretically one of the most area and delay efficient subquadratic multiplier design in the literature for intermediate size operands, but actual implementations of this multiplier have not been presented. The main Galois multiplier designs are therefore compared individually before implementing them with full GCM designs.

1.2 Structure

Chapter 2 will provide some background to finite field arithmetic which is useful in understanding the multiplication operation in GCM. An introduction to AES, with a focus on hardware based implementations will also be provided along with a functional specification of GCM. Parallel Galois field multipliers for the GCM will be given in Chapter 3 with implementation results. The GCM datapath and improvement made to reduce key change latencies will be described in Chapter 4. ASIC and FPGA results of GCM designs with parallel multipliers is also given in that chapter. Concluding remarks on the contributions made in this thesis and possible areas of improvement for future work are presented in Chapter 5.

Chapter 2

Background

The Galois/Counter Mode (GCM) of operation is built using two main components, namely a finite field multiplier and a 128 bit symmetric block cipher. In order to understand the GCM better, a brief introduction to finite field arithmetic will be provided in this chapter. Although any block cipher may be used within the GCM, since the Advanced Encryption Standard (AES) is the preferred choice, an overview of that block cipher will be given as well. The general functionality of GCM will then be provided in Section 2.3 with a literature survey of previous state of the art.

2.1 Finite Fields

Finite fields, also known as Galois Fields (GF), are algebraic structures with a finite number of elements. Many cryptographic and signal processing applications use Galois Fields because of their concise representation in hardware and efficient arithmetic operations. They have both multiplication and addition operations defined over the field and contain a prime, or a power of prime number of elements. They are denoted by GF(p) where p is a prime number representing the order, or in the case of an extension field, GF(p^m) where m is an integer [21].

2.1.1 Galois Field Representation

Galois fields are succinctly stored in hardware by mapping their polynomial representations into binary. The binary field GF(2^m) is particularly suited for this and is widely used as a result. Depending on the bases used, a polynomial representation can be constructed in the following manner. Given GF(2)[x], a set of all polynomials with coefficients in GF(2) $\in \{0, 1\}$, and $F(x)$, an irreducible polynomial within that set, then GF(2)[x]/$F(x)$ is a Galois field with 2^m polynomial elements. In other words, this construction is taking all possible polynomials and creating a set modulo $F(x)$. The number of polynomials in that reduced set is 2^m, and it can therefore be isomorphically mapped to the Galois field GF(2^m). $F(x)$ is called the field generating polynomial and can be selected from any irreducible polynomial of degree m. A polynomial that cannot be factored into any polynomials in GF(2)[x] is said to be irreducible [21]. Any element within GF(2^m) is representable by polynomials modulo $F(x)$ and all addition and multiplication operations are also performed modulo the irreducible field polynomial. Since the polynomial coefficients are either {0,1} they can be easily stored as binary strings.

Basis of Representation

There are different bases of representation that are possible for representing GF(2^m) mapped polynomials and their choice impact how Galois arithmetic computations are performed. Polynomial basis and normal basis representations are most commonly used for cryptographic applications [36, 25, 7], but there are other bases such as the dual, shifted polynomial, and triangular basis that have also been found useful in the literature [3, 5, 9]. For the binary extension field, a polynomial basis (PB) is constructed by finding any element $x \in \mathrm{GF}(2^m)$ such that $\{1, x, x^2, \cdots, x^{m-1}\}$ forms a basis, or a linearly independent set, of $\mathrm{GF}(2^m)$ over $\mathrm{GF}(2)$ [20]. All elements in $\mathrm{GF}(2^m)$ can be written with respect to the polynomial basis. The following equation shows a PB representation of an element $A(x) \in \mathrm{GF}(2^m)$ with coefficients $a_i \in \mathrm{GF}(2)$.

$$A(x) = \sum_{i=0}^{m-1} a_i x^i = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x^1 + a_0 \tag{2.1}$$

The normal basis for the binary extension field can similarly be created by finding any element $\beta \in \mathrm{GF}(2^m)$ such that $\{x^{2^0}, x^{2^1}, x^{2^2}, \cdots, x^{2^{m-1}}\}$ forms a basis of $\mathrm{GF}(2^m)$ over $\mathrm{GF}(2)$. It is well prooven that a normal and polynomial basis exists for all fields $\mathrm{GF}(p^n)$ [20]. A normal basis representation of an element $A(x)$ is given in the following equation.

$$A(x) = \sum_{i=0}^{m-1} a_i x^{2^i} = a_{i-1}x^{2^{i-1}} + a_{m-2}x^{2^{m-2}} + \cdots + a_1 x^{2^1} + a_0 x^{2^0} \tag{2.2}$$

The normal basis has some interesting properties that allow for an efficient implementation of the squaring operation which can be computed by a cyclic shift of bits. As a result, it has been applied to Elliptic Curve crypto systems which have frequent squaring operations. Elliptic Curve Cryptography (ECC) makes use of elliptic curves over finite fields to create schemes for public key cryptography.

2.1.2 Polynomial Basis Galois Field Arithmetic

The GCM described in this paper uses a polynomial basis of representation so Galois arithmetic operations performed over the binary extension fields with respect to the polynomial basis will be described here.

Galois Field Addition

The addition of elements in GF$(2)^m$ is equivalent to the addition of mapped polynomials. Since the coefficients in binary extension fields are in GF$(2) \in \{0,1\}$, the additions are modulo 2 which is equivalent to a exclusive-or (*XOR*) operation. Refer to Eq.(2.3) for this construction. Since there is no carry involved in element by element addition, the overall computation is simply a *XOR* operation performed on bit strings. The additive inverse, or subtraction is also defined for the field, and for the binary extension field is also computed using the *XOR* operation.

$$\begin{aligned} A(x), B(x) &\in \mathrm{GF}(2^m) \\ C(x) \equiv A(x) + B(x) \ &\equiv \sum_{i=0}^{m-1} (b_i \oplus a_i) x^i \end{aligned} \tag{2.3}$$

Galois Field Multiplication

Galois field multiplication in relation to addition is slightly more involved since it consists of a polynomial multiplication followed by a modulo reduction using the field polynomial. Given two elements to be multiplied, $A(x), B(x) \in \mathrm{GF}(2^m)$, and $F(x)$, the field polynomial, the result can be computed from Eq.(2.4). The coefficients of the polynomials $A(x)$ and $B(x)$ are denoted by the vectors (a) and (b) respectively.

$$\begin{gathered} C(x) \equiv A(x) \cdot B(x) \bmod F(x) \\ C(x) \equiv \sum_{i=0}^{m-1} b_i x^i \cdot a \bmod F(x) \\ C(x) \equiv (b_0 \cdot a + b_1 x \cdot a + b_2 x^2 \cdot a + \cdots + b_{m-1} x^{m-1} \cdot a) \bmod F(x) \end{gathered} \tag{2.4}$$

A simple method of computing this involves the use of a linear feedback shift register (LFSR). The pseudo code for this multiplier given below simply loops through the summation in Eq.(2.4) and accumulates a modulo reduced answer. The LFSR contains one of the operands A, and depending on its most significant bit, the field polynomial is XORed to the LFSR at each step. The result of the multiplication is generated in the register C by the end of m iterations. This register adds the value of A at each step depending on the coefficients of the other multiplicand H. This design is called a serial multiplier design, and other multiplier designs exist such as the parallel multiplier that is able to compute $C(x)$ in a single iteration. More details will be provided in Section 3.1 for converting Eq.(2.4) into a parallel multiplier structure.

Algorithm 1 GF(2^m) multiplier. [3]

```
Input A,H ∈ GF($2^m$), $F(x)$ Field Polynomial.
Output $C(x)$
$C = 0$
for $i = 0$ to $m$ do
  if $H_i = 1$ then
    $C \leftarrow C \oplus A$
  end if
  if $A_{127} = 0$ then
    $A \leftarrow$ rightshift($A$)
  else
    $A \leftarrow$ rightshift($A$) $\oplus F(x)$
  end if
end for
return C
```

2.2 Advanced Encryption Standard (AES)

The Advanced Encryption standard is a 128 bit block cipher that has been widely used since its acceptance in 2001 [23]. The design of AES was intended to be a more secure replacement of DES (Data Encryption Standard). Many efficient hardware and software designs have been documented, taking into consideration various tradeoffs of speed and area resources. The following sections will provide a general functional description of AES with an increased focus on the hardware design of AES components. High speed hardware datapaths that will be relevant in understanding the GCM datapath will be presented toward the end of this section.

2.2.1 AES Round Block

Each round of AES is modular and consists of four main computations namely, Byte Substitution, Mix Columns, Shift Rows, and Round Key addition. All rounds in AES are identical with the exception of the last round which has no Mix Columns operation. Byte Substitution consists of 16, 8 bit word substitutions while the Mix Columns operation is constructed from a matrix multiplication. Both of these operations are defined by Galois field operations in GF(2^8), but there are different means to implement them. The Shift Rows operation is simply a permutation on the inputs, and the Round Key operation consists of XORing key values generated from a Key Schedule component. The following diagram illustrates the general round structure of AES which is repeated based on the key input. For a 128 bit key, a single round repeats 10 times, while the 192 and 256 bit keys have 12 and 14 rounds of computation respectively for increased security.

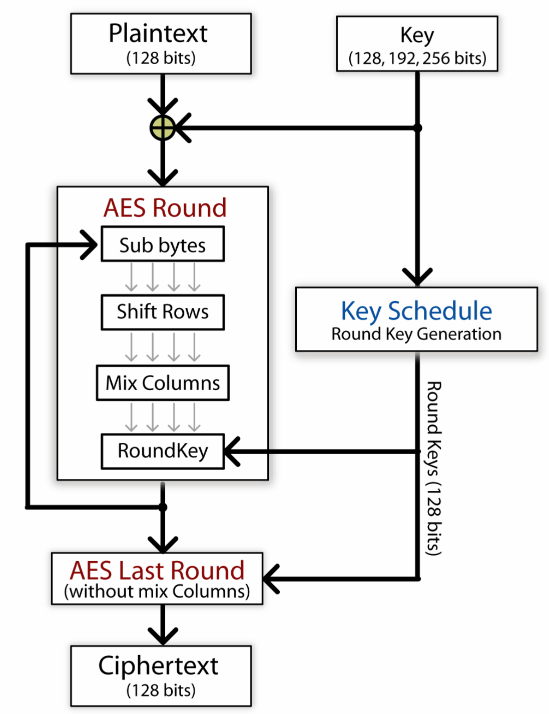

Figure 2.1: AES Round Structure

Different hardware datapaths can be created from this modular round structure. An iterative design can use the same design given above but simply adds a 128 bit data register at the end of the round structure. After a maximum of 14 cycles the AES encryption result can be obtained. This iterative design can be unrolled to create a pipelined implementation that has registers placed between round blocks. This is an outer pipelined AES design and a 128 bit output can be generated at each clock cycle with a full pipeline. There is enough flexibility, however, in choosing locations of the pipelined registers. Within each of the round components, additional pipelined stages can be added within the Sub-bytes operation which will be described in Section 2.2.2. This is labeled as an inner pipelined AES design, and although a higher latency and area is present, higher throughputs are possible.

The 128 bit plain text input is mapped into a state array which is a 4x4 block of 8 bit words that is manipulated in each round. For the following sections the state array block will be used to describe the different round operations so it is important to understand how the input is transformed into the state array. Figure 2.2 shows this transformation, by filling bytes of data into the state array by columns. After the AES encryption round, the last state array outputted is transformed back into a 128 bit stream.

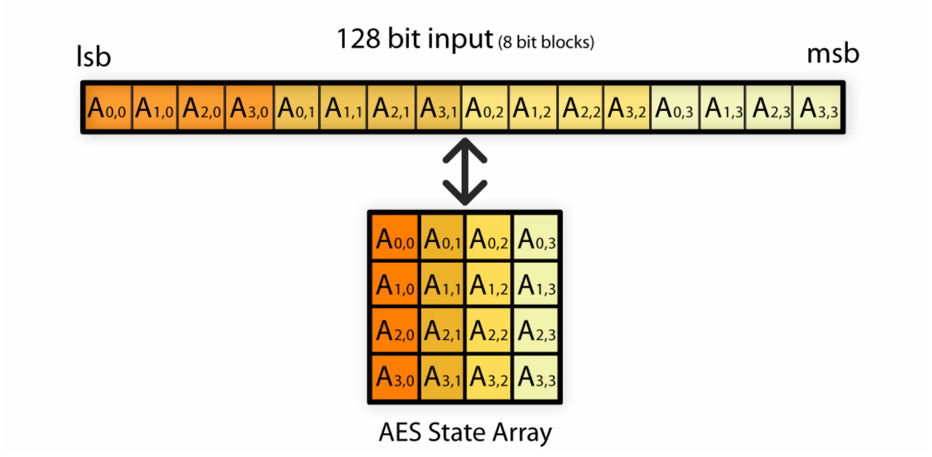
128 bit input (8 bit blocks)
lsb
msb
A0,0 A1,0 A2,0 A3,0 A0,1 A1,1 A2,1 A3,1 A0,2 A1,2 A2,2 A3,2 A0,3 A1,3 A2,3 A3,3
A0,0 A0,1 A0,2 A0,3
A1,0 A1,1 A1,2 A1,3
A2,0 A2,1 A2,2 A2,3
A3,0 A3,1 A3,2 A3,3
AES State Array

Figure 2.2: AES Round State Array Transformation

2.2.2 Byte Substitution (Subbytes)

The subbytes operation uses multiple substitution box components (Sbox) each of which performs an 8 bit substitution. Each 8-bit word of data in the state array, is substituted using the Sbox. This results in 16 Sbox components used for each round block, and is the most hardware area consuming part of an AES round. The Sbox computation is essentially a multiplicative inverse in GF(2^8) followed by an affine transformation which is a linear mapping from one vector space to another [30]. A lookup table of 2^8 values can be used to implement the Sbox component, but it can also be mathematically computed using logic gates.

Sbox Designs

Rijimen, one of the creators of AES showed in [26] a method of computing the Sbox by breaking operations in GF(2^8) down to a composite field GF($(2^4)^2$) resulting in significant hardware area savings which would otherwise not be possible using look-up table implementations. The inverse formula for the Sbox is given in its reduced version, in Eq.(2.5), where λ is $(1100)_2$. The addition, multiplication, and inverse operations are computed in GF($(2^4)^2$), and can be further broken down to the smaller composite fields, GF($(2^2)^2$) and GF(2^2), using the divide and conquer method.

$$a'x + b' = (ax + b)^{-1} = a(a^2\lambda + b(a + b))^{-1}x + (b + a)(a^2\lambda + b(a + b))^{-1} \quad (2.5)$$

Figure 2.3 shows a visual diagram of the composite Sbox. The isomorphic mapping to the composite field, (GF(2^8) $\rightarrow$ GF($(2^4)^2$)) can be implemented using a matrix vector multiplication. The affine transformation consists of a linear transformation followed by a translation which can be achieved by a matrix vector multiplication and vector addition respectively. The isomorphic mapping and affine transformation both use fixed matrices that are sparse so the computation costs of these operations are minimal [30].

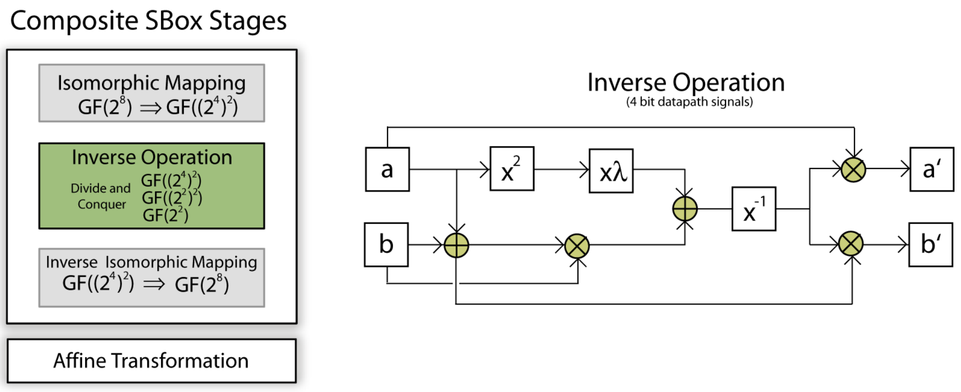

Figure 2.3: AES Composite Sbox design

The area consumed by the composite Sbox circuit is very low in comparison with the lookup table approach (LUT). In the above design by Satoh, the Sbox component consumes 250 gates while a LUT implementation is more than 4 times larger in area [30]. The computational cost does increase the circuit delay, so for high speed designs, LUT, and Binary Decision Diagram (BDD) Sbox implementations are preferred. The BDD implementation provides a slightly faster alternative to the LUT Sbox consuming less area resources. Each bit of the 8-bit output is associated with a binary tree and based on the input bits, each tree helps decide what the output bits should be. The 8 bit input is used as selector values for several layers of multiplexers in order to realize the binary trees in hardware. This type of construction faces large fan out issues for the initial multiplexer layers. An improved alternative to the BDD that improves these issues is the Twisted-BDD and is the fastest reported Sbox in the literature. The area requirements of this design, however, is almost double that of the LUT Sbox design [22].

2.2.3 Shift Rows

The Shift Rows operations consists of cyclically moving elements around in all but the first row of the 128 bit input block. The rows are left shifted by 1, 2, and 3 times respectively for rows 2, 3 and 4. The following mapping illustrates this process. In hardware no logic is required for this step and simple wire connections are used for this step to route the input to the output.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

Figure 2.4: AES Shift Rows

2.2.4 Mix Columns

The mix columns operation consists of a multiplication and reduction operation over GF(2^8). Each column of the state array is multiplied by the polynomial $3x^3 + x^2+x+2$ and reduced modulo the field generating polynomial x^4+1. This operation is generally optimized into a single matrix vector product. The four column blocks are used as the vectors, while a constant 4x4 matrix is used that combines the modulo operation. The result vector is stored in the next state array at the same location as the original column vector. All elements are 8 bits in width and the multiplication and addition operations are performed over GF(2^8).

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{2.6}$$

Since the elements of the matrix are of low degree the multiplications are simplified. A multiplication with 2 in GF(2^8) consists of a shift operation along with a modulo reduction if an overflow occurs. This operation can be reused with multiplying by 3, but an extra addition is required since $3{\cdot}a_i = (2{\cdot}a_i) \oplus a_i$.

2.2.5 Key Schedule

Round keys are *XOR*ed at the end of every round and are generated using a Key Schedule. These keys can be precomputed or generated at each round. The Sbox components used in the subbytes section are also used here for the round key generation. For each inputted key length, the method of generating keys is slightly different, but they contain similar logic components.

The 128 bit key has an Sbox operation done on the last column of the cipher key state array after the column bytes are rotated. This is followed by a *rcon* value *XOR* addition. The *rcon* value is generated based on the exponentiation formula $rcon(i) = x^{254+i} \bmod x^8 + x^4 + x^3 + x + 1$ performed over GF(2^8). These values are usually precomputed and once the *rcon* value is added there is an *XOR* chain on the columns of the state array that creates the next 128 bit round key. Figure 2.5 shows a single round key computation. This process is repeated by using the round key as a cipher for generating the next 128 bits of key material. The rcon i value starts at 1 and increments for each round key.

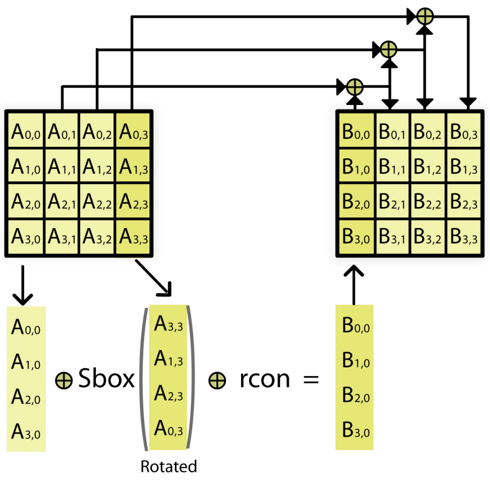

Figure 2.5: AES 128 bit Key Schedule Round

The 192 bit key schedule is similarly defined but the *XOR* chain is extended for another 2 columns to achieve a full 192 bits of key material. Each round unit of AES, however, only uses 128 bits of key material at a time, so the remaining bits are carried over for the next round. Figure 2.6 shows this key generation process. The six column vectors of the key state array are condensed here and shown as $\{A_0, A_1, \cdots, A_5\}$.

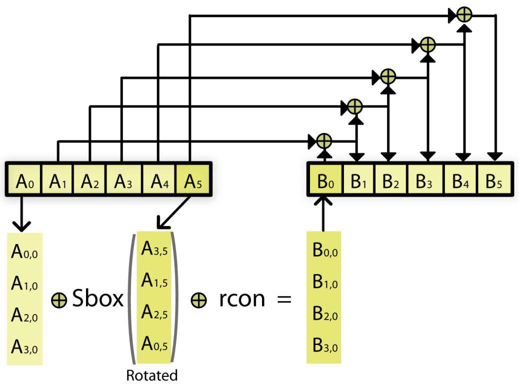

Figure 2.6: AES 192 bit Key Schedule Round

The 256 bit key schedule has an additional Sbox computation involved in generating key material. The first 128 bits of key material is generated as shown in Figure 2.5. For the next 128 bits of key material, an Sbox computation is performed on the fourth column and this follows another chain of *XOR* statements. Note that the *rcon* operation is not performed here with the Sbox.

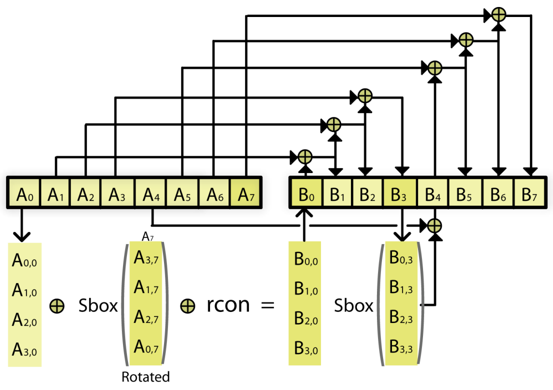

Figure 2.7: AES 256 bit Key Schedule Round

In order to compute the key schedule operation in hardware most designs generally precompute roundkeys before starting data encryption or decryption. Computing the key schedule on the fly, while rounds are being computed is possible for encryption, and has been implemented for iterative AES [16, 1]. There is added complexity when supporting all keys primarily because of the overlap occurring in operations. Figure 2.8 shows that although 128 bits of key material are generated at each round, there is still key material computed from previous round keys for the 192 and 256 bit key schedules. The S_{rcon} represents a Sbox computation with an *rcon* computation, while an S represents a simple Sbox computation. The arrows represent the *XOR* chaining of column vectors.

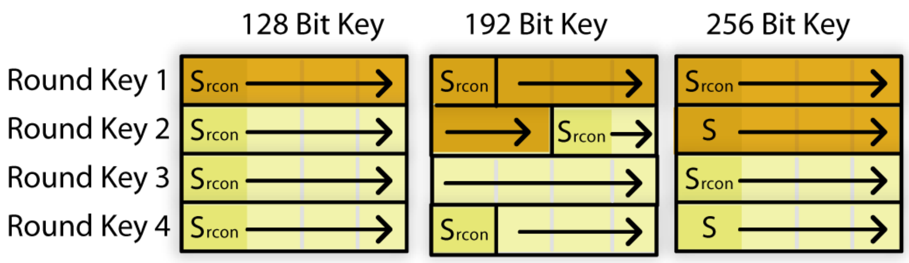

Figure 2.8: AES Key Schedule Pattern

For an Iterative key schedule the Sbox components are needed only once in each iteration for all round keys as can be seen in Figure 2.8. Although the 192 bit key has an Sbox in the middle of the round, it can still be used with the other key types. The delay of the design in this way is also limited to have only Sbox, *rcon*, and an XOR chain of computations regardless of the key used, so compared to a AES round block, it would not be apart of the critical path. The *rcon* values may or may not be included depending on if the key is 256 bits. The control unit for this key schedule drives multiplexers to guide input into the Sboxes, and direct outputs to the correct round key registers based on the key type.

Iterative key schedules have been used in pipelined designs for pre-computing keys, but there is a key latency cost associated with such an integration. If key changes occur more frequently for a flow of data the throughput in pipelined designs would be affected since clock cycles are wasted in updating key material. Having lower key change latencies therefore is very relevant for increasing the average throughput for AES.

2.3 GCM Overview

There have been several schemes devised which combine security goals of both authentication and confidentiality. Data authentication provides a means to detect accidental and unauthorized modifications of data while confidentiality helps ensure that data is readable only by the individuals it was intended for. Schemes such as CCM[34] and EAX[2] meet those goals by using existing authentication schemes in conjunction with a block cipher encryption in counter mode for achieving confidentiality. The authentication and encryption steps for these schemes are computed separately in two passes so pipelining and processing data in parallel is not possible. These schemes are therefore unable to achieve the high throughput rates needed for network applications. The Offset Code Book (OCB) is a scheme that overcame these deficiencies by providing a single pass computation for both the authentication and encryption steps, and is one of the fastest schemes in use today [27]. Despite the advantages of the scheme, since it has been patented, its use has been limited in standards.

The Galois/Counter mode of operation is a combined authentication and encryption scheme designed by David Mcgrew and John Viega, and is seen in a number of recent standards by the NIST and the IEEE. The GCM helped fill a need in the industry since it allows for fast software and hardware implementations that do not have intellectual property restrictions [18]. The GCM is a fully piplinable and parallel scheme, that shows throughput performance that far exceeds 10 Gbps and in some cases also has performance which rivals the OCB scheme [19]. The GCM uses a 128 bit symmetric block cipher such as AES in a counter mode to achieve data confidentiality, while a chained Galois multiplication operation is used for achieving data authentication. Both of these operations are performed sequentially on data blocks, so data can be fed in continuously in a pipeline form. A formal definition of GCM is given here with reference to the NIST SP80038D Standard [4].

2.3.1 GCM specification

The input to GCM includes AAD (Additional Authenticated Data), P (Plaintext), K (AES encryption key), and IV (Initialization vector). The inputted AAD and P are streams of bits broken up into 128 bit blocks, given by $A_0, A_1, A_2, \cdots, A_s$ and $P_0, P_1, P_2, \cdots, P_t$. If the last block is not 128 bits in length then 0's are padded to both the AAD and P streams accordingly. The number of of blocks in AAD is defined by s while the number of blocks in P is t. The IV is also set to 96 bits and this is the ideal for hardware designs since additional operations are required for other IV lengths.

The output of the scheme is C, a stream of 128 bit cipher text blocks, and the authentication tag T, which is also 128 bits. The entire tag does not need to be sent and a predefined number of most significant bits may be sent instead. Although forgery attacks are possible with smaller tags, the choice of smaller tags may be necessitated for video or voice applications that stream a large amount of smaller packets. The NIST has recommneded 32 bit and 64 bit tags for these types of applications with some additional security considerations limiting the amount of data encrypted per IV [4]. The following equation illustrates the GCM operation. For a simpler definition, it is assumed that AAD and P variables have already been padded with zeros and their lengths are divisible by 128. The Y_i values given below represent counter values for AES input.

$$\begin{aligned}
Y_0 &= IV \| 0^{31}1 \\
Y_i &= Y_{i-1} + 1 \\
H &= AES_k(0^{128}) \\
EK_0 &= AES_k(Y_0) \\
C_i &= P_i \oplus AES_k(Y_{i+1}) \\
T &= GHASH(H, A, C) \oplus EK_0
\end{aligned} \tag{2.7}$$

The following is the GHASH function description that has the Galois field multiplication operation. The field polynomial for the multiplication operation is $x^{128} + x^7 + x^2 + x + 1$. The key H is one of the operands for the multiplier and is generated by performing an AES encryption with an all zeros input (0^{128}). The other multiplicand is chosen from A_i, C_i, or the length data block. The $len()$ function, used for generating the length block, computes the total number of bits in the operand and returns a 64 bit wide value. AAD and the plaintext are fed into the len function and the result is then concatenated for creating the length block $(len(A) \parallel len(P))$.

$$GHASH(H, A, C) = X_{s+t+1}$$
$$X_i = \begin{cases} (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1 \text{ to } s \\ (X_{i-1} \oplus C_{i-s}) \cdot H & \text{for } i = s+1 \text{ to } s+t \\ (X_{i-1} \oplus (len(A)||len(P))) \cdot H & \text{for } i = s+t+1 \end{cases} \tag{2.8}$$

A graphical view of the above equation is provided here with more details on the functionality of GCM. Block ciphers such as AES have different modes of operation, and the counter mode of operation is used with GCM as shown in Figure 2.9. A passed initialization vector (IV) is constantly incremented and fed into a AES block for the encryption. Plain text blocks are XORed with the encrypted counter values in order to obtain ciphertext blocks. The first counter value encryption is later used for generating the authentication tag.

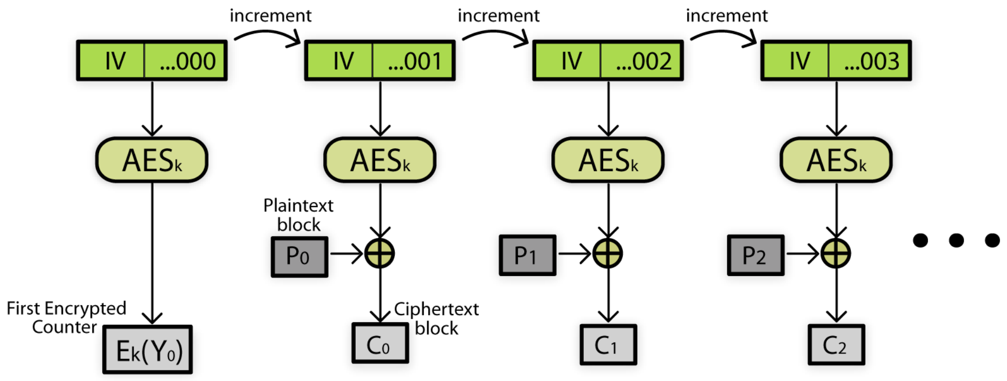

Figure 2.9: Counter mode AES used for GCM

GCM supports the authentication of both encrypted and unencrypted data. This is a useful feature that is included in the IPSec standard, where packet IP information that needs to be in the clear and read by routers, is authenticated along with the encrypted payload. The unencrypted data blocks are provided first into a Galois field multiplier chain followed by the ciphertext blocks. Figure 2.10 shows this process. After all the cipher text blocks are exhausted, the length block that appends the number of bits in unencrypted and encrypted data blocks is fed into the GF multiplier. The $E_k(Y_0)$ value that was the first encrypted counter value, is XORed at the end to get the final authentication tag.

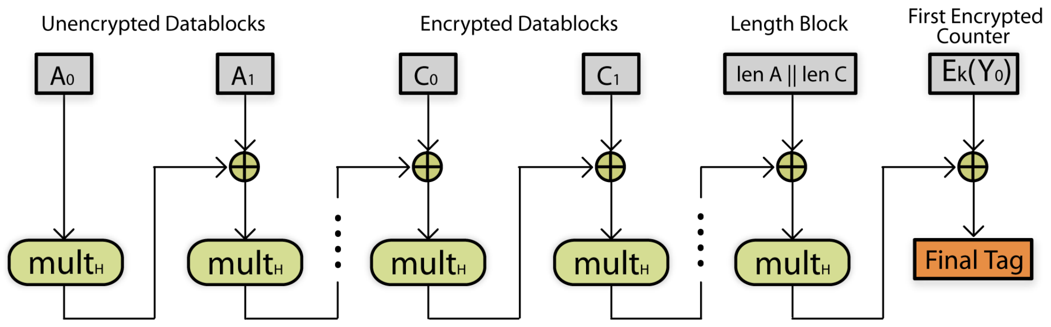

Figure 2.10: Galois Multiplier Blocks in GCM

The authentication tag is transmitted with the unencrypted and cipher text blocks to the destination. The decryption operation is similar to encryption, with the only change being that the cipher text blocks are XORed with the counter encryptions resulting in plain text blocks as the output. In Figure 2.9 this can be visualized by swapping the plaintext and ciphertext blocks. The authentication tag is generated in the same way at the destination by using the transmitted authenticated data and ciphertext blocks. If the generated tag matches the tag that was transmitted then the data can be excepted as valid, otherwise it is discarded.

2.3.2 High Speed GCM Implementations

ASIC implementations of GCM reported in the literature have throughputs of up to 42 Gbps using outer pipelined AES rounds and a parallel Mastrovito multiplier for the Galois operation [29]. Composite Sbox implementations were used to get more area efficient designs, and for higher throughput GCM, the BDD Sboxes were used. The datapath width of the design is 128 bits, so at every clock cycle a block is outputted. Commercial designs have reported similar throughputs for standalone GCM implementations [32, 13]. A parallel architecture that computes four GCM operations simultaneously was proposed by Satoh and is the highest reported throughput to date capable of up to 160 Gbps [28]. This design, however, has an increased number of input pads required for each parallel GCM operation. A pipelined multiplier architecture was proposed in [31] achieving 54.9 Gbps in a synthesized version of the design. This design made use of an inner and outer pipelined AES block with a low latency pipelined multiplier which was constructed from the parallel architecture GCM. By using this type of construction, the feedback condition for the multiplier was maintained and a high throughput was achieved due to the increased pipelined stages. The higher key change latency which ranges from 40 to 56 cycles, reduces the overall efficiency if there are more cold starts that occur as a result of key updates.

FPGA designs have also been proposed for the GCM in [15] and using a Shifted Polynomial basis parallel multiplier their designs achieved up to 15.3 Gbps. The width of the datapath proposed ranged from 8 to 128 bits, and provided designs with varying latency and area tradeoffs. Three Sbox solutions, the LUT, composite and a Block Ram implementation were used in the GCM. Another FPGA design with notable results was presented in [39] which provided a Karatsuba Algorithm multiplier(KA) implementation. This design made use of the composite Sbox with the KA multiplier to achieve a throughput rate of 15.23 Gbps using outer round pipelining. Using inner round pipelining and a brute force multiplier a 20.61 Gbps throughput was achieved but had double the latency. This implementation also only supported 128 bit AES keys unlike the implementation given in [15] which supports all key types.

Chapter 3

Parallel Multiplier Designs for GCM

Due to the feedback chaining present for the Galois multiplication operation in the GCM, pipelined designs have generally chosen parallel multipliers to complete the multiplication step in a single clock cycle. The Mastrovito multiplier has been a prime choice for its low critical path but it unfortunately has a quadratic space complexity. Parallel multipliers that have subquadratic area are therefore a good option to replace the Mastrovito design in GCM. A popular subquadratic multiplier based on the Karatsuba multiplication algorithm (KA) will be introduced in Section 3.3 and a recently proposed subquadratic multiplier design by Fan-Hasan(FH) will be introduced in Section 3.4. The Mastrovito multiplier is also presented in order to allow a better comparison between the multiplier designs.

The three multipliers presented are separable into two categories based on the approach taken to multiply field elements. The Mastrovito and FH multipliers use a matrix vector product (MVP) which can compute a modulo reduced result in a single step. The matrix used in the operation is constructed from the field defining polynomial, so this method is applicable when a field polynomial or a set of polynomials is known ahead of time which is the case for GCM. The KA multiplier on the other hand is not as tightly coupled with the field polynomial, but computes multiplication and modulo reduction in two separate steps. The MVP approach is first described before going into specific multiplier designs for GCM. A comparison of these parallel multipliers, with FPGA and ASIC implementation results will be provided toward the end of the chapter. The multipliers are designed specifically for the GCM operation but may be generalized for other applications as well.

3.1 Matrix Vector Product Based Multiplication

The original GF multiplication operation given in Eq.(2.4) can be modified to formulate the matrix vector product and the rearranged equation is provided below. The polynomial matrix P is computed using the coefficients of $A(x)$ while the vector portion is simply the transposed coefficients of $B(x)$. The matrix vector product shown here computes the multiplication and reduction operations in a single step and is applicable to both the Mastrovito and FH multipliers presented in this chapter.

$$\begin{gathered} C(x) \equiv A(x) \cdot B(x) \bmod F(x) \\ C(x) \equiv \sum_{i=0}^{m-1} (x^i \cdot a \bmod F(x)) \cdot b_i \\ C = P \cdot b^T \\ P = \{a^{(0)}, a^{(1)}, a^{(2)}, ..., a^{(m-1)}\} \end{gathered} \tag{3.1}$$

In Eq.(3.1), C is the column vector corresponding to the polynomial $C(x)$. An expansion of the polynomial matrix P is given in Eq.(3.2). The $a^{(i)}$ coefficients are essentially column vectors that are modulo reduced versions of $x^i a \bmod F(x)$.

$$\begin{aligned}
a &\equiv ax^0 \bmod F(x) \\
a^{(1)} &\equiv ax^1 \bmod F(x) \\
a^{(2)} &\equiv ax^2 \bmod F(x) \equiv a^{(1)}x \bmod F(x) \\
a^{(3)} &\equiv ax^3 \bmod F(x) \equiv a^{(2)}x \bmod F(x) \\
&\ldots \\
a^{(i)} &\equiv a^{(i-1)}x \bmod F(x)
\end{aligned} \tag{3.2}$$

The first column of P, $a^{(0)}$ has the coefficients of $A(x)$ while each subsequent column is the previous column multiplied by x and modulo reduced by $F(x)$. When this matrix is multiplied by the coefficients of $B(x)$, the result $C(x)$ is achieved.

3.1.1 GCM Polynomial Matrix

A polynomial matrix P was created with the GCM field generating pentanomial, $x^{128} + x^7 + x^2 + x + 1$, based on the method given in Eq.(3.2). The top left hand side of the GCM polynomial matrix is shown in Eq.(3.3) to give a better idea of what this matrix looks like. Coefficient terms listed such as $(a_0 a_{127} a_{126})$ denote the *XOR* addition operation $(a_0 \oplus a_{127} \oplus a_{126})$. The repeated elements occurring at the diagonals of the polynomial matrix are useful in optimizing the multiplication operation and is utilized in both the Mastrovito and FH multipliers. For the GCM polynomial, rows 3 to 7 and rows 8 to 128 show this repetition at the diagonals.

$$P = \begin{pmatrix} a_0 & a_{127} & a_{126} & a_{125} & a_{124} & a_{123} & \cdots \\ a_1 & a_0 a_{127} & a_{127} a_{126} & a_{126} a_{125} & a_{125} a_{124} & a_{124} a_{123} & \cdots \\ a_2 & a_1 a_{127} & a_0 a_{127} a_{126} & a_{127} a_{126} a_{125} & a_{126} a_{125} a_{124} & a_{125} a_{124} a_{123} & \cdots \\ a_3 & a_2 & a_1 a_{127} & a_0 a_{127} a_{126} & a_{127} a_{126} a_{125} & a_{126} a_{125} a_{124} & \cdots \\ a_4 & a_3 & a_2 & a_1 a_{127} & a_0 a_{127} a_{126} & a_{127} a_{126} a_{125} & \cdots \\ a_5 & a_4 & a_3 & a_2 & a_1 a_{127} & a_0 a_{127} a_{126} & \cdots \\ a_6 & a_5 & a_4 & a_3 & a_2 & a_1 a_{127} & \cdots \\ a_7 & a_6 a_{127} & a_5 a_{126} & a_4 a_{125} & a_3 a_{124} & a_2 a_{123} & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix} \qquad (3.3)$$

3.2 Mastrovito Multiplier Design using MVP

The Mastrovito multiplier is a widely used parallel multiplier with a quadratic space complexity [17]. The design is essentially a brute force multiplier in the sense that the MVP is computed like traditional matrix multiplication. It does optimize the operation since the repeated values that are present in the polynomial matrix can be computed once and then reused as signals in hardware for the brute force multiplication portion. Hardware resources are saved to some extent in this way. Elements in P are in GF(2), so *AND* and *XOR* gates are used for element wise multiplication and addition respectively. Since the Mastrovito multiplier uses the brute force approach, after computing elements in P, the Mastrovito design has a single layer of *AND* gates for element multiplication followed by layers of *XOR* gates to compute the final result. The simplicity of the Mastrovito design is evident in Figure 3.1 which provides an overview of the multiplier. The design is easy to code into a low level design using any hardware description language such as VHDL.

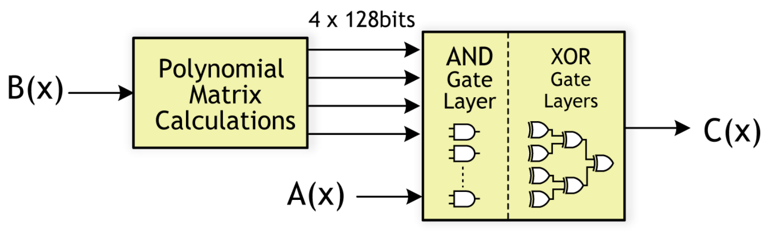

Figure 3.1: Mastrovito multiplier for GCM

The area complexity of the Mastrovito multiplier design for the brute force portion is m^2 *AND* gates while the number of *XOR* gates is $m^2 - m$. The *XOR* gate count for the polynomial matrix computations will vary based on the field polynomial, and is computed using the Hamming weight of the matrix. For the GCM this is equal to 784 *XOR* gates.

The time complexity can be summarized as $T_A + (\lceil \log_2 m \rceil + \lceil \log_2 \theta + 1 \rceil) T_X$, where T_A and T_X is the *AND* gate and *XOR* gate delays respectively. The θ constant is the maximum Hamming weight from all the columns of the polynomial matrix. Figure 3.2 which shows the number of *XOR* gate computations required for the top part of the polynomial matrix created from the GCM field pentanomial (20 rows shown). The bottom portion of the P matrix has elements repeated at the diagonals and does not show any other interesting areas. The maximum Hamming weight per column from this can be seen to be $\theta = 6$ so the maximum delay that is observed for creating the polynomial matrix is equivalent to a $\lceil \log_2(6+1) \rceil T_X = 3 \cdot T_X$ gate delay.

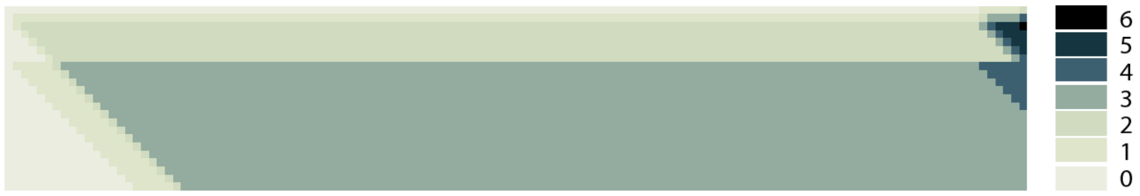

Figure 3.2: GCM Polynomial Matrix *XOR* gate counts per element

3.3 Karatsuba Algorithm Subquadratic Multiplier

The Karatsuba Algorithm (KA) was originally used to compute digit multiplication [14], and was mapped to polynomials by [24]. It has a subquadratic area complexity but with a larger delay in comparison with the Mastrovito multiplier. Subquadratic multipliers such as KA generally decrease the number of multiplication operations while increasing the number of addition computations. Since the cost of adding GF elements is low and equivalent to *XOR*ing bit streams in hardware, the KA is a suitable approach for GF multiplication. Using divide and conquer techniques the multiplication operation is divided up into smaller and smaller operations followed by an expansion to get the final product. This reduction and subsequent expansion is constructed by levels of *XOR* operations and as a result causes the delay of the multiplier to increase.

3.3.1 KA Multiplier Formulation

The elements $A(x), B(x) \in \mathrm{GF}(2^m)$ are first each split into two polynomials of max degree $\frac{m}{2} - 1$. A_h and B_h represents the upper polynomial coefficients while A_l and B_l represents the lower coefficients of the elements. The following equations show $A(x)$ split into two smaller polynomial elements, A_h and A_l

$$\begin{aligned} A(x) &= x^{\frac{m}{2}} A_h + A_l \\ A_h &= (a_{m-1}, a_{m-2}, \cdots, a_{m/2+2}, a_{m/2+1}) \\ A_l &= (a_{\frac{m}{2}}, a_{m/2-1}, \cdots, a_1, a_0) \end{aligned} \tag{3.4}$$

The multiplication of the two elements in $\mathrm{GF}(2^m)$ is first computed to get a polynomial of max degree $2m-2$ ($C'(x)$). The $\oplus$ operation represents *XOR*ing bit streams in Eq.(3.5) and multiplication operations shown are with sub-polynomials.

The original multiplication is divided into three lower degree polynomial multiplications and this can be further split recursively. The $C'(x)$ element is obtained once the recursion unrolls, and this is then modulo reduced separately to get the final $C(x)$ element.

$$\begin{aligned}
&D_0,\ D_1,\ D_2 \text{ have max degree } m/2-1\\
D_0 &= A_l B_l\\
D_1 &= (A_h \oplus B_l)(A_l \oplus B_h)\\
D_2 &= A_h B_h\\
C'(x) &= x^m D_2 \oplus x^{\frac{m}{2}}(D_1 \oplus D_0 \oplus D_2) \oplus D_0\\
C(x) &= C'(x) \bmod F(x)
\end{aligned} \tag{3.5}$$

3.3.2 Modulo Reduction

Modulo reduction of $C'(x)$ using the field polynomial can be performed by a multiplication with a fixed reduction matrix. Using the GCM field polynomial as an example, the higher order coefficients of $C'(x)$ can be modulo reduced based on the following equations.

$$\begin{aligned}
0 &\equiv x^{128} + x^7 + x^2 + x + 1 \mod F(x)\\
x^{128} &\equiv x^7 + x^2 + x + 1 \mod F(x)\\
x^{129} &\equiv x^8 + x^3 + x^2 + x \mod F(x)\\
&\cdots
\end{aligned} \tag{3.6}$$

The reduction matrix has $2m-2$ columns and m rows. The matrix essentially maps $C'(x)$ to $C(x)$ and is shown in Figure 3.3 for the GCM. The first m columns of the matrix form an identity matrix since elements of degree 1 to $m-1$ do not need to be reduced. Using Eq.(3.6), all elements of degree m to $2m-2$ can be modulo reduced and then used in creating the remaining $m - 2$ columns of the reduction matrix. The cost of this operation in relation to the KA multiplication is small and is dependent on the field polynomial. The Hamming weight of the reduction matrix for the GCM shows that this operation requires 527 *XOR* gates. Having low order terms within the field polynomial helps reduce the cost of the operation since higher order terms have additional feedback terms which increase the cost of the operation. For a field polynomial such as $x^{128} + x^{40} + x^2 + x + 1$ the cost of the operation is 623 *XOR* gates. The delay for the reduction operation can be computed by $(\lceil \log_2 \theta + 1 \rceil)T_X$, where θ is the largest Hamming weight computed by row of the reduction matrix. For the GCM reduction matrix this delay is computed to be $3T_X$.

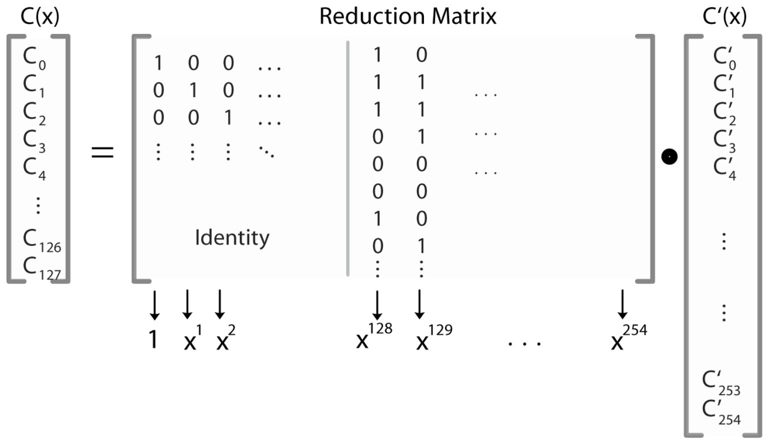

Figure 3.3: Reduction Matrix for GCM

3.3.3 KA Multiplier Design for GCM

The Karatsuba algorithm generally works best with elements of even degree since each step in the recursion splits polynomials equally. The input element size for the GCM Galois operation is 128 bits, a power of 2, so the KA multiplier can be easily applied without any changes required. A high level view of the Karatsuba multiplier is provided here with all the major components required.

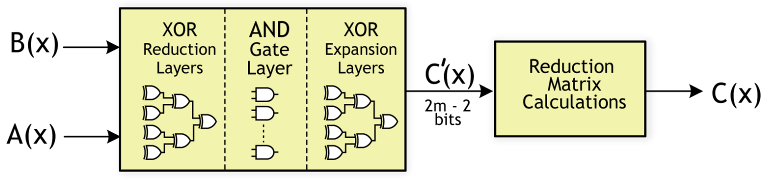

Figure 3.4: Abstract view of the Karatsuba Multiplier

The polynomial elements can be conveniently split down to single element multiplications but this is not always desirable in terms of area efficiency. When the ending condition of the recursion is changed and brute force multiplication performed instead, this leads to some savings in terms of *AND* and *XOR* gates. The following table shows the number of gates required for halting at different polynomial sizes. The gate counts do not include the reduction operation which has a fixed number of gates and a fixed delay of $3T_X$. The ending condition delays are based on the brute force multiplication delay which is $T_A + log_2(n)T_X$ where n is the halting value.

Table 3.1: Area of the KA Multiplier with varied ending conditions

Halt	XOR gates	AND gates	Total Gates	NAND Gates	Delay
2	9913	2916	12829	45484	$T_A + 19T_X$
4	8455	3888	**12343**	**41596**	$T_A + 17T_X$
8	7969	5184	13153	42244	$T_A + 15T_X$
16	8455	6912	15367	47644	$T_A + 13T_X$
32	9913	9216	19129	58084	$T_A + 11T_X$
64	12415	12288	24703	74236	$T_A + 9T_X$

We can see from Table 3.1 that it is worthwhile halting the KA when the polynomial size is 4 since it provides the lowest area and delay complexity. Since the cost of *XOR* gates in hardware is usually larger than that of *AND* gates, in order to get more accurate area estimates for ASIC implementations, the *NAND* gate count is included. The area cost of 1 *XOR* gate is bounded by the area of 4 *NAND* gates while one *AND* gate is bounded by the area of 2 *NAND* gates. When taking the *NAND* gate count into consideration the results still showed halting at 4 as the optimal choice in terms of area.

3.4 Fan-Hasan Subquadratic Multiplier

The FH multiplier is a subquadratic area, parallel multiplier that was recently proposed in [6, 8]. Its asymptotic space complexity is quoted to be 8% lower than that of KA while its time complexity is 33% lower for a trinomial as the field polynomial. When compared with the Mastrovito multiplier, although it has a larger delay its area footprint is less than half for $m = 128$. The FH multiplier takes advantage of the Toeplitz structure found in the polynomial matrix P in order to perform more area efficient multiplication. A Toeplitz matrix vector product (TMVP) is used for this multiplier and will be described first before going into implementation details for the GCM.

3.4.1 TMVP Formulation

A Toeplitz matrix is a matrix which has all diagonal elements equal. The m by m matrix T given in Eq.(3.7) for example, is a Toeplitz matrix and has elements in (row i, column j) equal to $(i+1,\, j+1)$. A variant of this matrix, the Hankel matrix, has equal diagonal elements in the opposite direction where elements (i, j) equals $(i-1,\, j+1)$. In order to uniquely define a Toeplitz matrix only $2n-1$ elements generated from the first row and column are needed.

The regularity present in the Toeplitz matrix can be used to our advantage when computing addition and multiplication. Addition of two square Toeplitz matrices for example only requires $2n-1$ element additions, while the remaining elements can be copied at the diagonals. Although a brute force method of multiplication on a TMVP does not provide any benefit, by creating a recursive multiplier design similar to the KA, it is possible to reduce the area complexity. An important quality of T useful for the recursive formulation is that all sub matrices of T are in Toeplitz form as well. Matrices T_0, T_1, and T_2 shown here are all Toeplitz matrices within the matrix T.

$$T = \begin{bmatrix} a_3 & a_4 & a_5 & a_6 \\ a_2 & a_3 & a_4 & a_5 \\ a_1 & a_2 & a_3 & a_4 \\ a_0 & a_1 & a_2 & a_3 \end{bmatrix} = \begin{bmatrix} T_1 & T_0 \\ T_2 & T_1 \end{bmatrix} \tag{3.7}$$

$$T_1 = \begin{bmatrix} a_3 & a_4 \\ a_2 & a_3 \end{bmatrix}, \quad T_0 = \begin{bmatrix} a_5 & a_6 \\ a_4 & a_5 \end{bmatrix}, \quad T_2 = \begin{bmatrix} a_1 & a_2 \\ a_0 & a_1 \end{bmatrix}.$$

Given a Toeplitz matrix $T = \begin{bmatrix} T_1 & T_0 \\ T_2 & T_1 \end{bmatrix}$ and vector $\mathrm{V} = \begin{bmatrix} V_0 \\ V_1 \end{bmatrix}$, the product $C = T \cdot V$ can be recursively constructed as given in Eq.(3.8). A single matrix vector product is broken down into three smaller products of matrices of half the size.

$$\begin{aligned} C &= \begin{bmatrix} P_0 + P_2 \\ P_1 + P_2 \end{bmatrix} \\ P_0 &= (T_1 + T_0) \cdot V_1 \\ P_1 &= (T_2 + T_1) \cdot V_0 \\ P_2 &= T_1 \cdot (V_0 + V_1) \end{aligned} \tag{3.8}$$

The addition of Toeplitz matrices given in the calculation of P_0 and P_1 can be optimized based on the repeated signals [6]. Assuming T is a m by m matrix, adding half matrices T_1 and T_0 is computed in $m - 1$ gates, and $m/2 - 1$ of these addition signals can be reused in computing T_2 and T_1. This results in having only $3m/2 - 1$ gates for computing both additions.

The multiplications for P_0, P_1 and P_2 are all TMVP computations that can be split further and the recursive design stops upon reaching single element multiplications. Like the KA implementation it is possible to get more savings by halting the recursion earlier and performing brute force matrix multiplication. Table 3.2 summarizes area of designs halting at different input sizes for a 128 bit FH multiplier. It is clearly seen that splitting when the matrix size is 4 has an optimal total gate count. Taking into consideration the cost of *AND* and *XOR* gates in an ASIC implementation with a *NAND* gate count, we still see halting at 4 is the optimal.

3.4.2 FH Multiplier Designs for GCM

Although the P matrix for GCM is not a complete Toeplitz matrix, it was shown in Section 3.1.1 that there are two regions in the matrix that have Toeplitz form (rows 3 to 7 and rows 8 to 128). Since the approach provided in the previous

Table 3.2: Area of the FH multiplier with varied ending conditions

Halt	XOR Gates	AND Gates	Total Gates	NAND Gates	Delay
2	9074	2916	11990	42128	$T_A + 13T_X$
4	7859	3888	**11747**	**39212**	$T_A + 12T_X$
8	7616	5184	12800	40832	$T_A + 11T_X$
16	8291	6912	15203	46988	$T_A + 10T_X$
32	9884	9216	19100	57968	$T_A + 9T_X$
64	12479	12288	24767	74492	$T_A + 8T_X$

section works only with square Toeplitz matrices there is some adjustment that needs to take place on the GCM polynomial matrix. Figure 3.5 shows three possible approaches to deal with this problem and are summarized here before going into them in detail. The first approach performs the TMVP on the larger Toeplitz section by extending it into a 128 x 128 Toeplitz matrix and performing brute force multiplication on the smaller section. The second approach aims to use the TMVP on a 122 x 122 portion of the larger Toeplitz section and get the final result by combining brute form multiplication results of the remaining sections. The last approach adds values to the existing polynomial matrix in order to convert it into a full 128 x 128 Toeplitz matrix and then compensate the additions in the end result. From these three methods the first was implemented in this thesis for its low delay.

Approach 1: Based on [8], the first row of the polynomial matrix is movable to the bottom of the matrix without disrupting the Toeplitz form. The C_0 term of $C(x)$ as a consequence, moves to the bottom of the result vector. When this is done a large 122 by 128 bit section is formed that has Toeplitz structure. This section of the polynomial matrix can be extended to create a full 128 by 128 Toeplitz matrix by padding 6 zeros to the first column and copying elements over. The TMVP can then be applied to this matrix. The remaining 6 rows at the top of the matrix can be computed using the Mastrovito multiplier approach. Since 5 rows in that region are in Toeplitz form, it may be possible to use the TMVP, but due to the smaller dimensions, the Mastrovito multiplier provides better results.

Approach 2: Another method would be to make a square portion out of the large 122 by 128 bit section and compute that using the TMVP. This means a 122 by 122 matrix is formed that has a Toeplitz structure. As this matrix is being split, there will be cases where the resulting sub matrices have non-even numbered dimensions. In that case zeros can be padded to a row and column to create an even matrix dimension. It is possible to remove a row and column as well but brute force multiplications would be required to compensate the action. The first 6 columns that are unaccounted for in the 122 by 128 bit section can be computed by brute force multiplication and the result compensated in the final $C(x)$. The top part of the polynomial matrix can be computed using the Mastrovito design as done in Approach 1.

Approach 3: Since some rows at the top of the GCM polynomial matrix prevents a full Toeplitz structure from occuring, another approach would be to add certain elements to the problematic rows to create a Toeplitz matrix and then compensate the changes in the end result. To describe this further, suppose we have the following matrix vector product computation where w, x, y, and z are polynomial matrix computations that prevent a Toeplitz structure.

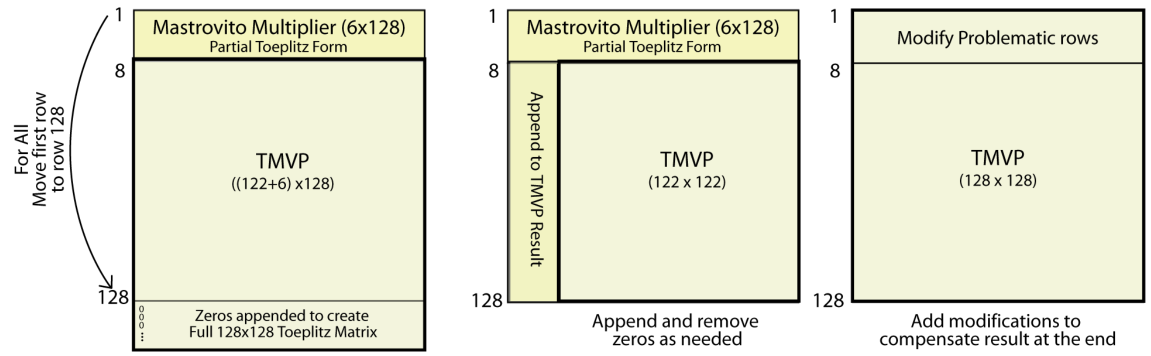

Figure 3.5: Possible FH Multiplier Designs for GCM

$$C = \begin{bmatrix} a_3 & a_4 & a_5 & a_6 \\ a_2 & a_3 \oplus w & a_4 \oplus x & a_5 \oplus z \\ a_1 & a_2 & a_3 \oplus w & a_4 \oplus x \oplus y \\ a_0 & a_1 & a_2 & a_3 \oplus w \end{bmatrix} \cdot \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \tag{3.9}$$

The above computation can be broken up into two MVP where one matrix is in Toeplitz form and the other compensates the matrix changes made. The subtraction operation given here is simply a bit wise *XOR* computation. For the GCM matrix the compensation would only occur for the first 6 rows of the polynomial matrix since row 7 onward is in Toeplitz form. This is assuming that the first row is moved to the bottom of the matrix.

$$C = \begin{bmatrix} a_3 \oplus w & a_4 \oplus x \oplus y & a_5 \oplus z & a_6 \\ a_2 & a_3 \oplus w & a_4 \oplus x \oplus y & a_5 \oplus z \\ a_1 & a_2 & a_3 \oplus w & a_4 \oplus x \oplus y \\ a_0 & a_1 & a_2 & a_3 \oplus w \end{bmatrix} \cdot \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} - \begin{bmatrix} w & x \oplus y & z & 0 \\ 0 & 0 & y & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \tag{3.10}$$

3.4.3 FH Multiplier Design Comparisons

In terms of delay, Approach 2 and 3 both add an additional *XOR* gate delay because of the compensation operations required on the end result. Approach 3 has the lowest gate count out of the three, since the matrix compensation step for the first 6 rows is a sparse matrix requiring less brute force multiplications. The first approach provides the lowest delay and is relatively simpler to implement. By computing the first few rows using the Mastrovito multiplier has a timing benefit over using only the compensation step as described in Approach 3. From Figure 3.2 it was seen that the top 8 rows have the largest delay in computing the polynomial matrix elements. In Approach 3, the largest polynomial matrix computation ($\lceil \log_2(6 +$

$1)\rceil T_X$ gate delay) in addition to the TMVP computation creates the critical path of the multiplier. By using the Mastrovito multiplier on the higher delay polynomial matrix computations found in the first 8 rows, it decreases the critical path of the multiplier to some degree, since the Mastrovito delay is much smaller than the TMVP computations. The critical path is limited to the $\lceil \log_2(4+1)\rceil T_X$ delay for the polynomial matrix computation on row 8 plus the TMVP computation in this way. An abstract view of this implemented multiplier is provided here.

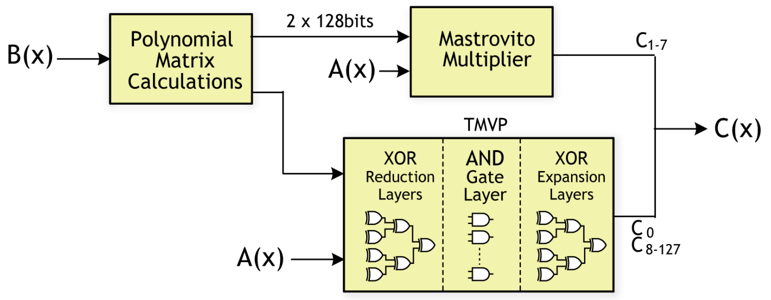

Figure 3.6: Implemented GCM FH multiplier design

3.5 ASIC and FPGA implementation

The implementation results provided here are full bit parallel multiplier designs for the GCM and include polynomial/reduction matrix calculations. Java programs were used in creating low level hardware descriptions of the three multipliers and generic code for this is provided in Appendix A. The programs essentially output VHDL assignment commands in conjunction with *AND* and *XOR* commands to achieve optimized bit level operations. The FH multiplier design provided was modified from the pseudocode given in [6] and the KA multiplier was created using the same concept as well. Test benches to verify the multipliers were created from a LFSR multiplier implementation.

ASIC implementation results were generated using an 130nm Tower Standard Cell Library. Each multiplier was synthesized using the Cadence RTL compiler and incrementally optimized several times until no further improvements were seen. Timing constraints were set according to the type of optimization desired. A faster clock frequency was set for speed optimization, while area optimizations were achieved by setting a lower clock frequency. Implementation results are provided in Table 3.3 for each multiplier, including area and delay results. The expected area complexities of *AND* and *XOR* gates for all the multipliers is also included in that table. In order to get more focused results, the multipliers implemented here did not include the feedback condition and register delays of the GCM multiplier block, but only the logic elements of the multiplier.

The hardware platform used for FPGA implementation was the Xilinx Virtex4-FX100 which contains 42176 slices. The multipliers were synthesized with a high effort level using Xilinx ISE and optimized for speed. Each slice on the Virtex4 contains two, four input look up table blocks (4-LUT) and two flip flops. Besides their use as logic components, slices are also used for routing signals within the FPGA. Both the slice and 4-LUT counts are provided along with the expected delay of the multipliers in Table 3.4.

Table 3.3: ASIC Results of Parallel Multipliers for GCM

Multiplier	**Area Complexity**	**Gate Count**	**Optimize**	**Delay(ns)**	**Area** (μm^2)
FH [6]	$\#\text{AND} = m^{\log_2 3}$ $\#\text{XOR} = 5.5m^{\log_2 3} - 5m - 0.5$	46932	Speed	1.614	365580
			Area	3.600	144226
KA [24]	$\#\text{AND} = m^{\log_2 3}$ $\#\text{XOR} = 6m^{\log_2 3} - 8m + 2$	44204	Speed	1.694	392176
			Area	3.710	141061
Mastrovito [17]	$\#\text{AND} = m^2$ $\#\text{XOR} = (m-1)^2$	100928	Speed	1.130	570566
			Area	2.204	323735

Table 3.4: FPGA Implementation of Parallel Multipliers for GCM (Virtex4)

Multiplier	**Delay(ns)**	**4 input LUT**	**Slices**	**Utilization (%)**
FH [6]	6.125	6616	3781	8.96
KA [24]	6.637	6136	3518	8.34
Mastrovito [17]	4.583	14199	8163	19.35

3.5.1 Results Analysis

FPGA results showed the Mastrovito multiplier taking up significant area resources over the FH and the KA multipliers. Almost 20% of the Virtex4's resources were taken up with this multiplier, whereas the FH and KA used a maximum of 9%. The delay for the FH showed a 8% improvement over the KA. The *NAND* gate counts for the FH are much higher than the KA becuase of the additional brute force computations required and this can be seen by the extra slice counts of the FH.

The ASIC results for the Mastrovito multiplier, as expected, show its advantage in terms of delay but it has 35% more area than the FH when optimized for speed and slightly more then double the area requirements over the subquadratic multipliers when optimized for area. Between the subquadratic multipliers, the FH had a 7% advantage in area and a 5% speed improvement over KA when optimized for speed. There are some possible reasons why the FH did not perform better in terms of delay. It should be noted that the original published paper on the TMVP used the Shifted Polynomial Basis (SPB) with a trinomial as the field polynomial which is expected to have a lower delay. The GCM pentanomial adds additional delay which is a possible reason for the descreapency in the results. The standard cell library used had a wide range of cells so it is possible that lower level optimizations caused the KA to have performance closer to the FH. For example, the KA multiplier for its ending steps has overlapping polynomial additions which take place with a delay of $2T_X$. Standard cells with more input pads, such as a four input *XOR* gate would help reduce this delay in the KA multiplier for some areas. This type of optimization is not as pronounced for the FH multiplier since each step is more divided. When using a 180nm library with a basic set of standard cells, the delay of the FH was up to 14% faster than the KA, a result closer to theoretical expectations.

With a FH multiplier design, the maximum possible throughput is 79 Gbps while for the Mastrovito multiplier we have close to 113 Gbps. In [10, 12], it was shown that for AES, throughput rates closer to 80 Gbps is achievable only with inner pipelining which consumes a significant amount of area resources and when using outer round pipelining, a maximum throughput of 48.2 Gbps was achieved with a LUT Sbox. Although these results were presented for a 180nm Standard cell library, we can deduce it is difficult to achieve AES throughput rates which match the multiplier throughput when a composite Sbox is used. With high speed FPGA implementations of AES as given in [11], a similar argument can be made.

This means that using the FH or KA over the Mastrovito multiplier would be the preferred choice for this implementation since it would not be the delay critical portion of the datapath. With a 130nm ASIC implementation of AES implemented in [29], the FH multiplier would be part of the critical path because of the low latency BDD Sbox design was used. By using a LUT Sbox design, which is 20% slower and less area consuming, more balanced pipelined stages may be achieved with the FH even though it is apart of the critical path. Based on this, the subquadratic multipliers were implemented into a pipelined AES-GCM datapath and the results are detailed in the following chapter.

Chapter 4

GCM ASIC and FPGA Implementation

Previously proposed GCM designs have focused primarily on increasing the overall throughput of pipelined datapaths. Attempts at reducing the hardware area have been approached by using iterative AES designs or by implementing smaller Sbox components since they encompass a significant portion of the AES block. From the multiplier implementations shown in the previous chapter we can see that it is possible to optimize the Galois field multiplier design of GCM without affecting the throughput significantly. Modifications of existing AES and GCM designs were done with the aim of increasing the average throughputs and to balance pipeline stages so that area resources are used efficiently. The GCM datapath used in this work is detailed first followed by ASIC and FPGA implementation results of the GCM using the different multiplier designs.

4.1 GCM Datapath

The GCM pipelined datapath employed for this work is provided in Figure 4.1. The datapath used here is similar to the one given in [38] and [15] but differs more in its lower level components. Block inputs for the GCM are 128 bits in length and are fed into the pipelined GCM which processes them sequentially. The data path provided supports both GCM encryption and decryption. An online key schedule is employed in such a way that a 1 to 4 clock cycle latency is experienced for key changes. Some of the important design decisions made while developing the GCM are provided here with reference to previous designs in the literature. Justification of adjustments made to previous designs is also provided.

4.1.1 Design Considerations

The GCM has an overall latency of 13, 15, or 17 cycles based on the key type used. After the AES operation, a 2 clock cycle latency is present for the Galois multiplication operation, and a multiplexer. The timing diagram of the GCM design is provided in Figure 4.2. The IV vector input was fixed to 96 bits in this design which has been recommended by NIST for providing higher throughput designs [4]. For variable length IV inputs an additional clock cycle of latency is added to the design since the IV needs to be fed into the Galois multiplication operation before it can be used. A simple incremental counter is used upon getting the IV input and a register holds the current Y_i value. Although a register is present, the design adds no additional latency to the design and its outputs are fed directly into the pipelined AES block.

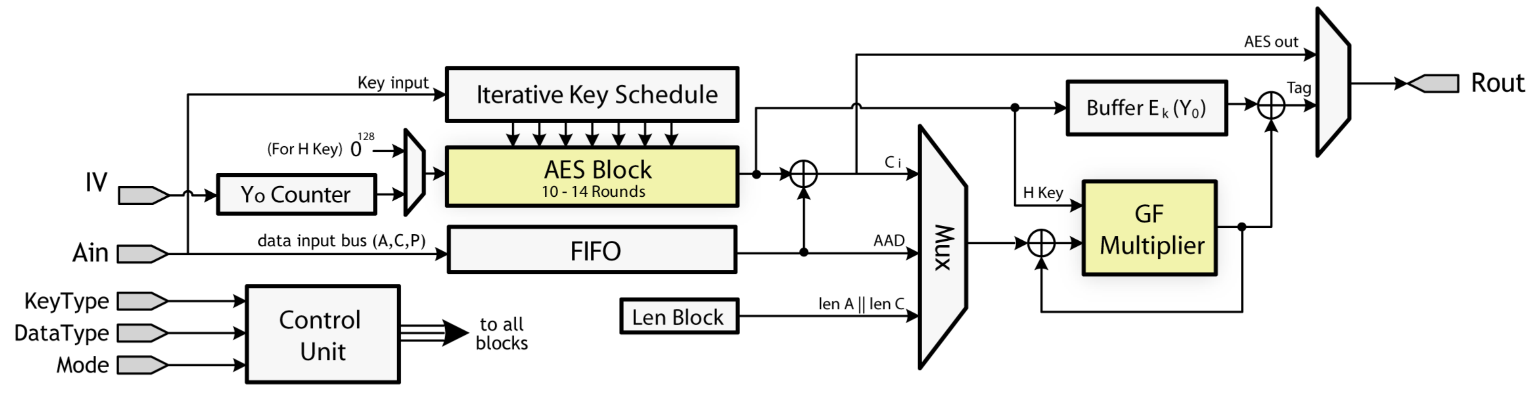

Figure 4.1: Implemented GCM Datapath

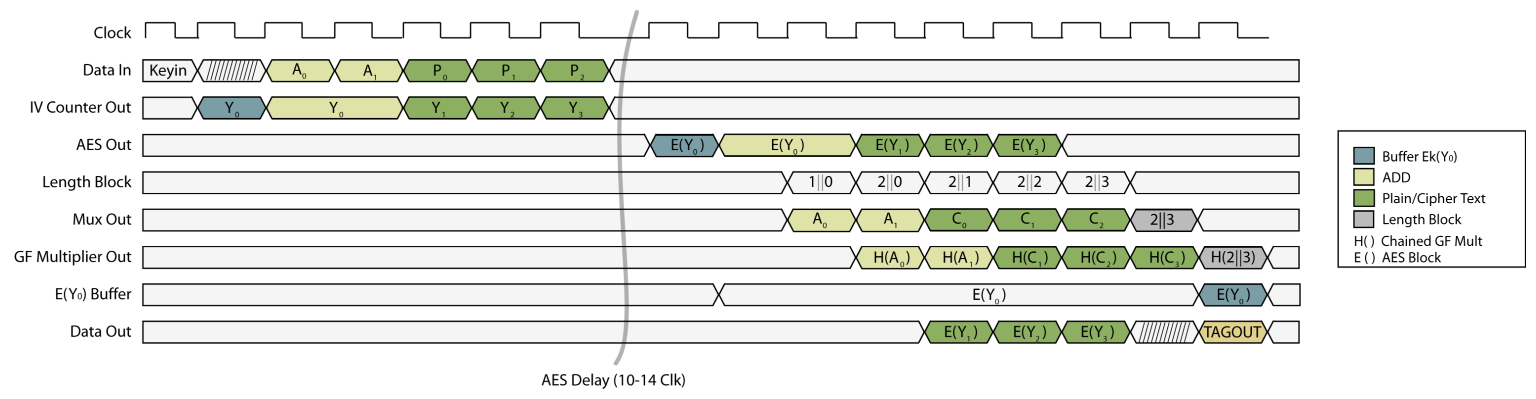

Figure 4.2: Implemented GCM Timing Diagram

The sequential nature of the datapath simplifies the control signals significantly. The data type signal is pipelined through the entire datapath and all blocks read the signals at the appropriate stages to direct outputs accordingly. GCM encryption and decryption operations are easily controllable by a *Mode* signal. The only change in functionality is in the multiplexer feeding input into the Galois multiplier. Under encryption, *AAD* and AES encrypted ciphertext blocks are fed into the multiplier sequentially. For decryption, the multiplexer selects the *AAD* and ciphertext blocks from the FIFO.

Pipelined registers are included within the major blocks and also in some locations to help increase throughput. The multiplexer used for feeding in data into the multiplier, for example, is held in a register for one clock cycle. This was done to reduce some of the combinatorial delay going into the GF multiplier. The use of the FIFO for feeding in datablocks is used in previous designs but more recently proposed designs by [29] omit it.

AAD and ciphertext blocks can be processed simultaneously under some conditions when no FIFO block is present but such a design has a more complicated control unit. Another possible reason for omitting the FIFO is that while a key update is taking place, *AAD* data blocks can be fed directly into the multiplier without requiring the need for the FIFO. This, however, requires the multiplier key to be generated using an Iterative AES block which was not used in [29]. By not including the FIFO block, an external source is forced to drive the chip and provide input blocks at the correct time to synchronize with the AES output blocks. Including the FIFO helps provide a better interface for the chip and a simpler control unit design.

4.2 Pipelined AES Design

An outer pipelined AES block was used for the GCM design that has a latency of 10,12 or 14 clock cycles depending on a 128 bit, 192 bit or 256 bit key chosen respectively. Inner pipelined rounds were not used since the parallel multipliers have a larger critical delay which would not be able to keep up with a higher frequency clock. There were two types of Sbox designs implemented, the composite Sbox design presented in [30] as well as a lower delay LUT design. The Mix Columns operation was performed by brute force multiplication to have minimal delay. The key schedule design is a modified version of [29] to provide lower latency for overall key updates in both the AES and multiplier blocks.

4.2.1 Key Schedule Considerations

Varying schemes for updating key values in the GCM datapath have been used in the literature. The design given in [38] uses an iterative AES block for updating the key for the GF multiplier while an offline key schedule is used for computing 128 bit AES round keys. Satoh uses an iterative key schedule design supporting all AES key types, but round keys are pre-calculated for their pipelined architectures and are not used until the entire key schedule has finished updating them [29]. Satoh does not specify additional details for key changes in his work but given that keys are pre-calculated, there is a 10-14 cycle latency in his design. The overall average throughput would decrease if more frequent key changes occur. Although online key schedules have been proposed in [33, 16], they have either been limited to 128 bit keys, or for iterative AES designs. To the best of the authors knowledge, a low latency online key schedule that supports all key types for a pipelined AES–GCM has not been proposed.

The key schedule implemented in this thesis is identical to iterative the key schedule in [29] but is made to be better integrated into the GCM pipeline. The latency for this proposed design is a maximum of 13 cycles depending on the type of input blocks and key sizes used. If the number of datablocks processed per key is larger then 14 blocks, then the maximum latency would be 1 on average. The blocks of data being processed in the AES pipeline by a previous key are not affected by the changes and only future input blocks are. Given these operating conditions, this key schedule integration provides a better average throughput over previous work which have latencies equivalent to the size of the AES pipeline.

4.2.2 Low Latency Iterative Key Schedule

An initial design attempt was made by unrolling an iterative key schedule and pipelining it. Although such a design would be ideal and have the potential of changing AES keys at every cycle of input with no latency cost, it is a very area intensive design. This unrolled design requires an additional 14 x 4 Sbox components which is equivalent to a little less then a third of the total area of the AES block. An iterative key schedule, on the other hand, is much more area efficient and consumes only 4 Sbox components along with some smaller logic components. Each round of the pipelined AES block has its own round key register which is updated by the key schedule. In Figure 4.3, we can see that as a new key is updated the old round key register values still remain until the key schedule updates them at the appropriate iteration. This property can be used in creating a lower latency key schedule.

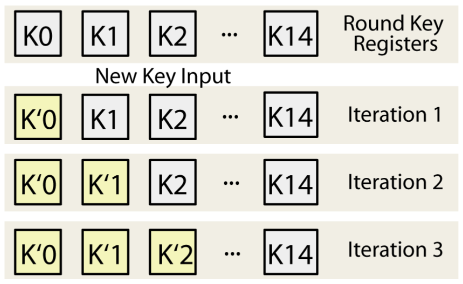

Figure 4.3: Key Schedule round key register updates by iteration

Since old key values are not updated on the key change, they are still usable for the current datablocks in the AES pipeline. If all the round keys registers are filled, then the GCM design would incur a one clock cycle latency for doing a key update. This one cycle latency is due to the multiplier key block which needs to be computed by encrypting an all zero input in AES. The new AES round keys are computed on the fly, and are available in time to do the multiplier key computation. This latency is assuming that the AES pipeline is filled.

There are some issues with this setup that need to be addressed. The first problem is that if a key change occurs before all the round key registers have been filled, then either a stall needs to occur, or the input staggered until all round keys have been updated. When a new key comes in, the iterative key schedule will shift to updating the first round key and stop its current key computation. The blocks in the AES pipeline that were dependent on the old AES key will no long have the correct round keys added since the Key schedule did not finish updating them. The worst case condition for this is present for a 256 bit key. If a key change occurs after a single block has entered the pipeline, then a 13 cycle delay latency is present to fill the remaining round keys and only after that can encryption begin on new blocks.

There is another problem that can occur even if the round keys have been updated. In Figure 4.4 we can see AES round blocks with the corresponding output based on the key type. When a key change occurs from a larger key size to a smaller one, such as a 256 bit key to a 128 bit key, there is a conflict that will occur. Supposing 256 bit datablocks are in rounds 10 - 14 in the AES pipeline, and 128 bit encrypted values fill rounds 1 to 9. In the next clock cycle, both the 128 bit encrypted values and the 256 bit encrypted values will be ready for output to the final block. The pipeline needs to stall for 4 clock cycles in this case to allow data to finish processing from the old key values. Similarly going from a 256 bit key to a 192 bit key or a 192 bit to a 128 bit key a 2 clock cycle stall needs to occur.

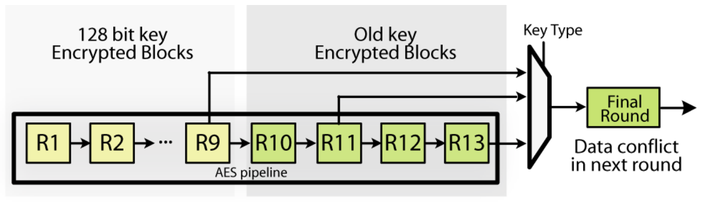

Figure 4.4: Collision on change in input keys

Pipeline stalling can be avoided by staggering inputs and adding a four clock cycle latency for all key changes. This would allow the current data blocks in the pipeline to empty so that collisions are avoided. An alternative method to avoid pipeline stalls would be to buffer the input of 128 bit and 192 bit key encrypted rounds to avoid conflicts. The latency of the entire AES block would be fixed to 14 clock cycles but key changes would occur in a single clock cycle for full pipelines. Figure 4.5 shows the additional registers required for achieving this. Note that in this case, key type signals need to be pipelined as well which is a minimal cost. A total of 6 additional 128 bit registers are added to the design with this setup but the benefit of having a single clock cycle key change latency is gained.

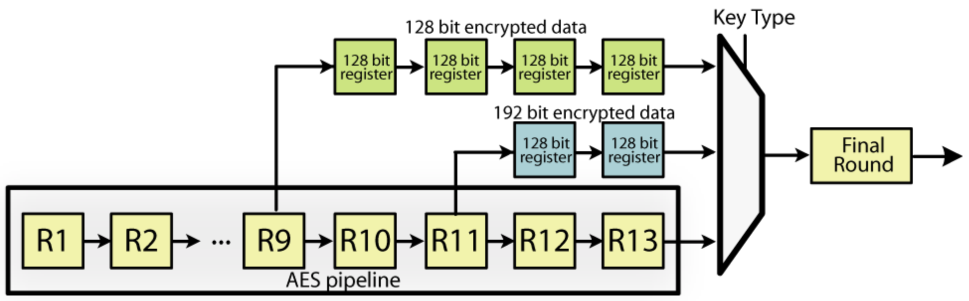

Figure 4.5: AES datapath for single cycle key change latency

Given the increased area requirements of the last scheme proposed, the key schedule with a fixed four cycle latency was employed with the iterative key schedule in the GCM datapath. Despite the added latency, the average throughput expected would still be higher than designs which have a fixed 14 cycle latency for all key changes.

4.3 GCM Implementation Results

4.3.1 ASIC Results

All combinations of the three parallel multipliers presented in Chapter 3 along with two Sbox designs, the composite and LUT Sbox, were implemented on an ASIC with a 130 nm Standard Cell Library under average case operating conditions. Incremental optimizations were used with a high effort level and a sample synthesis script is provided in Appendix A.3. The synthesis results are summarized on Table 4.1 and gate counts have been approximated using a two way *NAND* gate as the unit size which is a standard method of comparison. Design # 1 was the GCM design with the LUT Sbox while Design # 2 was implemented with the composite Sbox.

The timing constraints were chosen to allow either area or speed optimizations to take place. The performance measurement of $Kbps/gate$ allows a better comparison of speed and area tradeoffs, and these results can be seen in graphical form in Figure 4.7.

The Mastrovito multiplier has 12% more area requirements than the other multipliers with Design # 1 and a 18% increase in area resources when implemented with the composite Sbox (Design #2). The critical path for the GCM design was the GF multiplier in Design #1 since the LUT Sbox has a low delay. When synthesizing this design, the area of the subquadratic multipliers increase in order to achieve a smaller delay. This area increase is still lower then the Mastrovito multiplier design so a 12% benefit is seen in overall area requirements. Design #2, however, had the composite Sbox as part of the critical path. The GF multipliers as a result are synthesized with a lower area since there is more freedom in terms of delay. The area benefit for the subquadratic multipliers is much more as a result over the Mastrovito multiplier design. Figure 4.6 has overlapped pie charts showing area of the AES and multiplier blocks. Notice that for GCM design #2 the multiplier area is proportionally much higher with the Mastrovito multiplier.

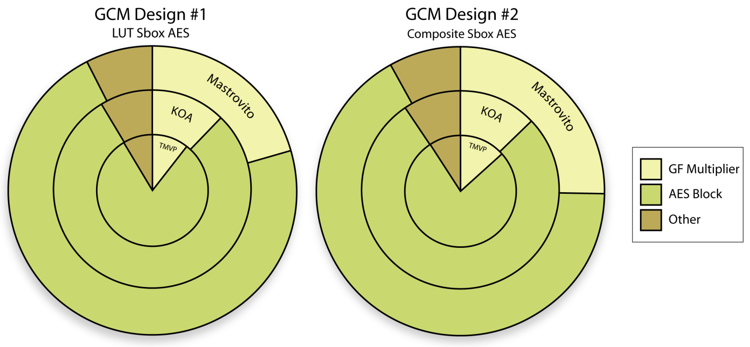

Figure 4.6: GCM Area Distribution

Table 4.1: GCM ASIC Synthesis Results Summary

GCM Design #1, LUT Sbox, (* BDD Sbox)								
Ref	**Multiplier**	**Optimize**	**Delay (ns)**	**Gates**	**Area (μm^2)**	**Frequency (MHz)**	**Throughput (GBps)**	**Efficiency (Kbps/gate)**
This Work	FH	Speed	2.510	219336	1474816.61	398	51.00	232.50
		Area	3.500	216813	1457853.302	286	36.57	168.68
	KA	Speed	2.644	222964	1499211.281	378	48.41	217.13
		Area	3.500	216512	1455828.033	286	36.57	168.91
	Mastrovito	Speed	2.500	245531	1650953.134	400	51.20	208.53
		Area	3.500	245062	1647798.233	286	36.57	149.23
[29]*	Mastrovito	Speed	3.000	297542	–	333	42.67	143.40

GCM Design #2, Composite Sbox								
Ref	**Multiplier**	**Optimize**	**Delay (ns)**	**Gates**	**Area (μm^2)**	**Frequency (MHz)**	**Throughput (GBps)**	**Efficiency (Kbps/gate)**
This Work	FH	Speed	3.560	177364	1192594	281	35.96	202.72
		Area	4.500	148407	997886	222	28.44	191.67
	KA	Speed	3.560	175805	1182111	281	35.96	204.52
		Area	4.500	148040	995420	222	28.44	192.14
	Mastrovito	Speed	3.560	205285	1380335	281	35.96	175.15
		Area	4.500	177103	1190839	222	28.44	160.61
[29]	Mastrovito	Speed	4.000	181198	–	250	32.00	176.60
		Area	5.000	174016	–	200	25.60	147.11

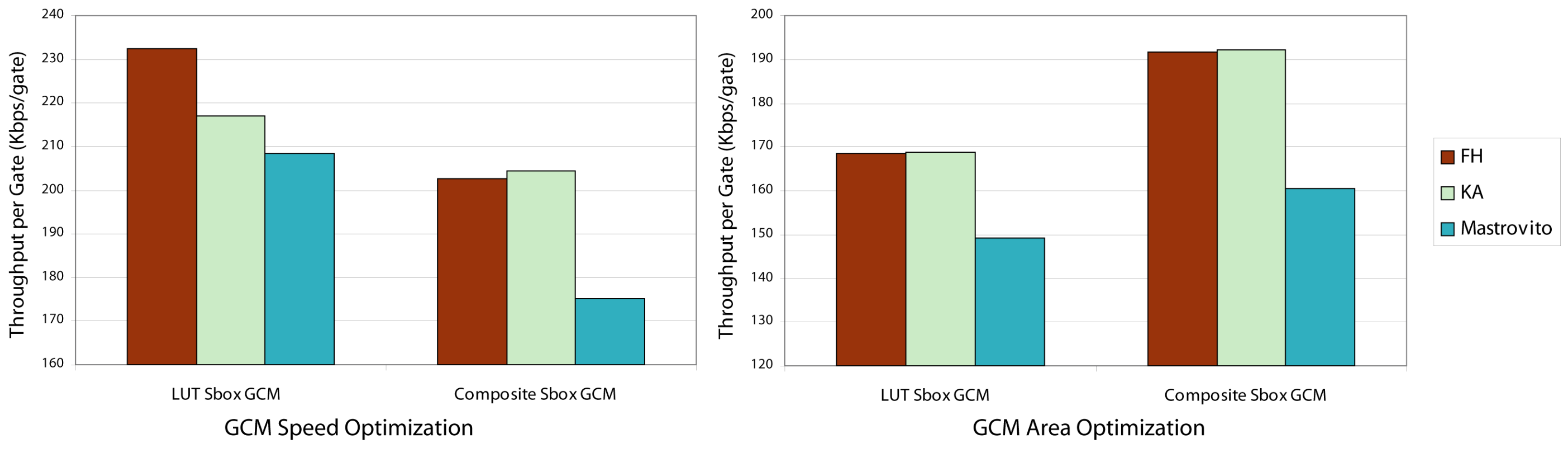

Figure 4.7: Throughput per Gate Performance for ASIC GCM Designs

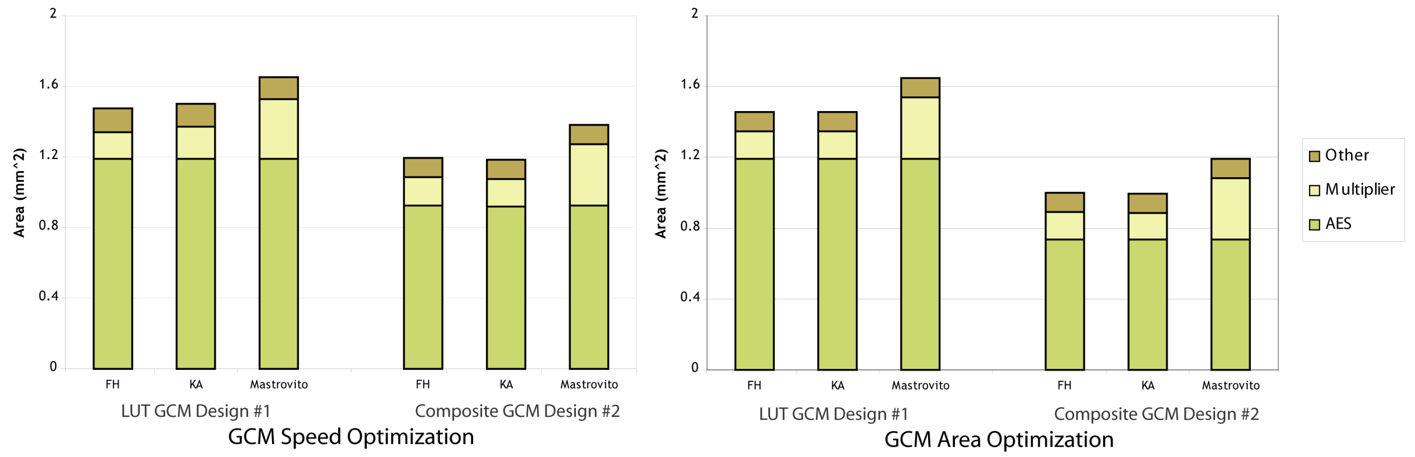

Figure 4.8: Area Histograms for ASIC GCM Designs

Table 4.2: GCM FPGA Place and Route Results Summary

GCM Design #3, Block RAM Sbox							
Ref	**Multiplier**	**Delay (ns)**	**Frequency (MHZ)**	**Throughput (Gbps)**	**Slices**	**Block Ram**	**Kb/slice**
This work	FH	7.738	129	16.54	9958	113	1661.15
	KA	8.547	117	14.98	9352	113	1601.37
Full AES [15]	Mastrovito	9.09	110	14.08	13200	114	1066.77

GCM Design #4, LUT Sbox							
Ref	**Multiplier**	**Delay (ns)**	**Frequency (MHZ)**	**Throughput (Gbps)**	**Slices**	**Block Ram**	**Kb/slice**
This work	FH	8.191	122	15.63	23718	1	658.86
	KA	9.098	110	14.07	23373	1	601.93
Full AES [15]	Mastrovito	8.333	120	15.36	27800	0	552.54

GCM Design #5, Composite Sbox							
Ref	**Multiplier**	**Delay (ns)**	**Frequency (MHZ)**	**Throughput (Gbps)**	**Slices**	**Block Ram**	**Kb/slice**
This work	FH	9.226	108	13.87	19394	1	715.37
	KA	9.811	102	13.05	18731	1	696.52
Full AES [15]	Brute Force	11.111	90	11.52	23200	0	496.56
128-AES [39]	KA	8.393	119	15.25	13523	0	1127.77

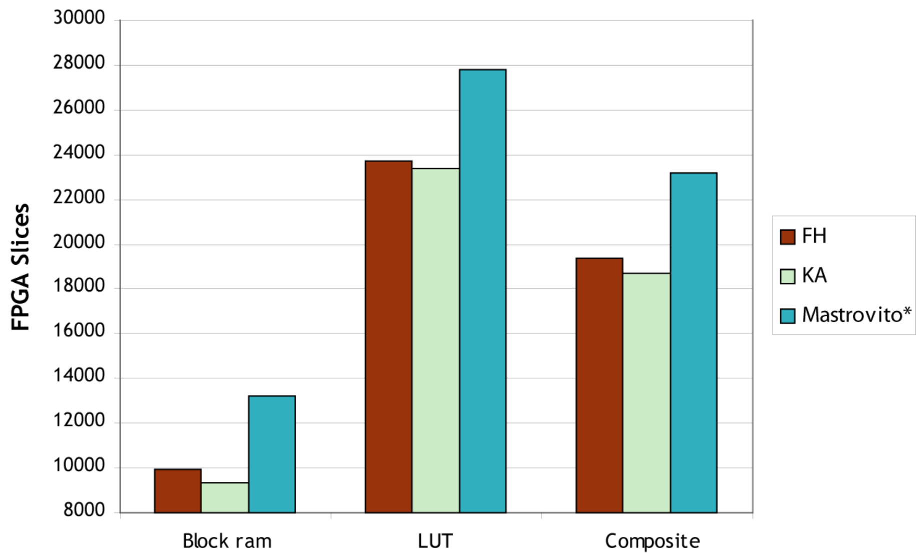

Figure 4.9: Area Histogram for FPGA GCM Designs (*[15])

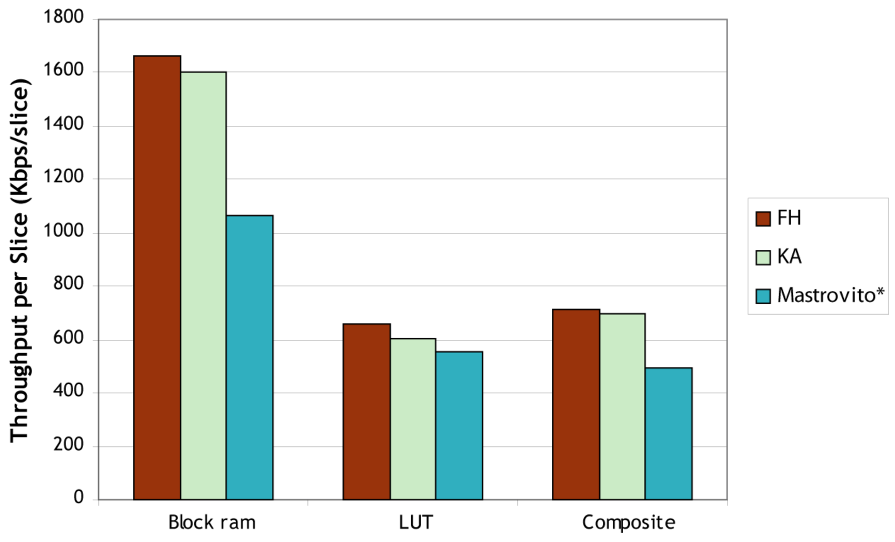

Figure 4.10: Throughput per Slice Performance for FPGA GCM Designs (*[15])

There was no notable difference in area between GCM designs implemented with the FH and KA multipliers. The brute force computations required for six rows of the Polynomial matrix make the FH design slightly more area consuming than the KA implementation, which is a reason for the lack of difference in overall area for those GCM designs. The delay advantage of the FH even though it has comparable area requirements to the KA allows its throughput per gate efficiency to be the highest out of all GCM designs when used with the LUT Sbox.

When compared with GCM designs in the literature, the sub-quadratic multipliers showed good throughput per gate efficiency. For the outer pipelined AES-GCM implementation given in [29] combined with a Mastrovito multiplier, a 42.67 Gbps throughput with 143.40 *Kbps/gate* efficiency was observed. The FH multiplier GCM design showed a higher 232.50 *Kbps/gate* efficiency, and when compared with similar throughput rates can be calculated to have a 198 *Kbps/gate* efficiency. Similar advantages with the composite Sbox designs can be seen where a 182 *Kbps/gate* efficiency is observed using matching throughput rates of comparable designs in [29] which showed a 176 *Kbps/gate* efficiency.

The focus of the results given above was to show the area efficiency of the designs, but the throughputs achieved in relation to practical applications are also competitive. The high throughput demands for network applications of GCM in the industry range from 10 to 40 Gbps currently so the designs presented with subquadratic multipliers are in line with those requirements [32]. Using the subquadratic multipliers over the Mastrovito approach would in fact be perferred since the brute force approach provides a higher than needed throughput. Although it is possible that with increasing throughput demands the Mastrovito multiplier would have to be used, but by combining the advantage of the subquadratic multipliers in GCM with better CMOS processes such as a 90nm standard cell library could help meet this demand. Other applications of GCM such as tape storage are currently limited by device read and write speeds so the ASIC designs presented here would not be as relevant for those applications.

4.3.2 FPGA Results

FPGA implementations of GCM were synthesized, placed and routed using Xilinx ISE and results are provided in Table 4.2. Three Sbox types were implemented namely, Block RAM, LUT and composite Sbox. The Block RAM solution is like a look up table approach, but it uses memory elements within the Virtex 4 FPGA and is able to provide results after a clock cycle. Due to this property the Block RAM components act as pipelined registers for AES. The iterative key schedule used LUT Sbox components, however, to reduce the complexity in using Block RAM resources.

The area advantage of the subquadratic multiplier over the brute force approaches used in [15] provide better slice utilization for comparable throughput rates. The KA multiplier GCM design provided in [39] supports only 128 bit AES keys, but the multiplier has been optimized to a greater degree. The halting condition was chosen based the lower level LUT component rather than *XOR* and *AND* gates, since they are the building blocks of FPGAs. The slice utilization as a result is higher, but it is difficult to compare the designs since not all key types were supported. The FH multiplier GCM implementations for all cases provided the highest throughput with the exception of the KA implementation given in [39]. The 16.54 Gbps throughput achieved with the Block RAM solution is, to the best of the authors knowledge, the fasted reported FPGA implementation to date of GCM.

4.3.3 Subquadratic Multipliers in Parallel GCM

Given the area flexibility in the subquadratic multiplier GCM designs, they could potentially be used in current high throughput GCM designs as well. The parallel GCM architecture could benefit from sub-quadratic multiplier to help reduce its large area requirements. The 4-parallel design presented in [28] uses four AES encryption blocks and 4 Mastrovito multiplier blocks. When implemented with the composite Sbox reported results showed a throughput of 102Gbps and with a gate count of 600 Kgates. Given the 20K gate savings achieved by moving to a subquadratic multiplier design, by using the FH or KA multiplier in the Parallel GCM architecture, an anticipated 80K gates could be saved without affecting the overall throughput of the design. This is more then 13% of the current area consumed by the design. The 4-Parallel GCM using BDD Sboxes has a 900K gate count and 160 Gbps throughput and could also benefit with roughly 8% decrease in area. The throughput for the design would decrease to 140Gbps in that case, however, since the subquadratic multiplier would be defining the critical path.

4.4 Key Change Latency Comparisons

In order to calculate the effect of a key change on the overall throughput of GCM, Eq.(4.1) is used. The function, $f(\alpha)$, represents the number of clock cycles for computing α bytes of data under a full pipeline. When adding a latency of β cycles, the efficiency in the throughput is equal to σ, a percentage of efficiency for the given key change latency. When this σ value is multiplied by the maximum throughput of the design the expected throughput for that particular packet size is obtained.

$$\begin{aligned} f(\alpha) &= \frac{\alpha * 8 \text{ (bits/byte)}}{128 \text{ (bits/block)}} \\ \sigma &= \frac{f(\alpha)}{f(\alpha)+\beta} \end{aligned} \tag{4.1}$$

Figure 4.3 shows the expected throughputs for GCM designs with different packet sizes based on a incurred key change latency cost. Both the high throughput designs shown here are from [29, 31]. A 54 Gbps inner and outer pipelined AES-GCM design with a pipelined multiplier design and a 42.7 Gbps outer pipelined design were compared with the proposed key schedule design in this thesis. The latency for the 54Gbps GCM design is 40 clock cycles for a 128 bit key change, and this lower bound was used in the computations rather than the upper bound of a 56 clock cycle delay expected for 256 bit key changes. The outer pipelined design had a fixed 14 clock cycle latency while the proposed designs computed latency based on the varying packet size inputs. The maximum throughput for the thesis design was set to 36Gbps with the key change latencies of 1 and 4-14 clock cycles.

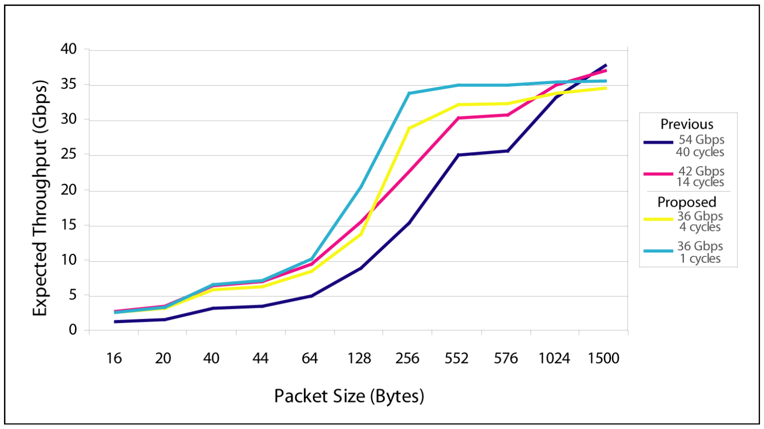

Figure 4.11: Average Throughputs of different key change latency designs (fixed packet size)

The packet sizes, however, are not fixed for typical Internet traffic flows. In order to calculate average throughput of designs the varied distribution needs to be taken into consideration, which can be computed using the Internet Performance

Index given in [19]. The average throughput expected for each of the designs based on a distribution of 60%, 20%, 15% and 5% for 1500, 576, 552, and 44 byte packets is provided in Table 4.3. The percentage difference from the maximum throughput is also provided.

Table 4.3: Throughput of GCM designs with varied key change latencies

Max Throughput	Key Change Latency (Cycles)	Average Throughput	Percentage Difference
54 Gbps [31]	40-56	31.75	59
42 Gbps [29]	14	33.32	78
36 Gbps [this work]	4-14	32.35	90
36 Gbps [this work]	1-14	33.99	94

These results show the advantage of the proposed GCM datapath design over previous state of the art. The average throughputs achieved are competitive with designs that use much more area resources as well. Average throughputs for the design in [29] would increase by 4 to 5 Gbps if the improved latency design is used. An argument against using an online key schedule such as the one proposed here is that it is possible to load pre-computed round keys from memory thereby saving the need to compute them on the fly. A key schedule unit that computes the round keys would be needed in that case and a one time latency cost of key changes would also exist. If a large number of keys are stored then this could significantly increase the memory requirements. For certain small router designs, it may be more cost effective to simply compute keys on the fly as described in this thesis

Chapter 5

Concluding Remarks

5.1 Contribution Summary

High throughput GCM designs with subquadratic parallel multipliers have been presented in this thesis. Both ASIC and FPGA results show higher area utilization for GCM designs with subquadratic multipliers than implementations using the brute force Mastrovito multiplier. For composite Sbox GCM implementations, it has been shown that subquadratic multipliers are an ideal choice since they do not effect the overall throughput and also reduce the area requirements of the GCM. This is the first time ASIC implementation results have been presented for the FH subquadratic multiplier and results show note worthy delay improvements over the KA multiplier. Although when implemented for the GCM pentanomial, the FH multiplier has increased area requirements, its efficiency in terms of throughput per gate is the highest out of all GCM designs. The highest FPGA throughput to date supporting all AES key types has also been achieved using the FH multiplier and a Block RAM Sbox.

5.2 Future Work

Although ASIC implementation throughputs of GCM are limited by the subquadratic multiplier, there is scope to improve the multiplier delays. There are two methods presented here that may provide faster throughputs. For the FH multiplier design it is possible to improve the delay slightly by computing the Polynomial Matrix step on the key input separately from the multiplier thereby saving a $3T_X$ combinatorial delay which is 20% of the current delay. Figure 5.1 shows how this may be achieved by moving the multiplier key register.

When implementing the multiplier it was noticed that the critical path comes from the multiplier key inputs $B(x)$, since those signals are sent through a polynomial matrix calculation step. By moving the location of the multiplier key register after the polynomial matrix calculations, the delay of that operation gets pushed from the multiplier to the last AES round block. This design would require a $2m-1$ bit register and for the AES block a LUT Sbox would have to be used since its critical path is smaller than the multipliers. If the delay of the LUT Sbox AES is affected by the added delay, however, then it is also possible to adjust the register within the polynomial matrix computations in order to better balance the delay. The composite Sbox AES has a larger delay than the subquadratic multiplier so it would not be appropriate for this implementation.

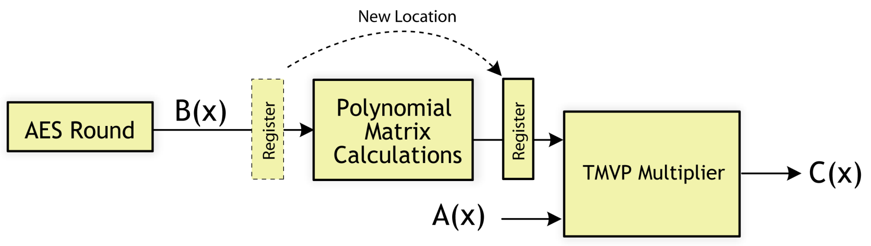

Figure 5.1: Improved delay FH design

The halting conditions provided in Tables 3.1 and 3.2 for the KA and FH multipliers respectively, provide good tradeoff conditions that may also be used to improve the area utilization results. Halting the recursion at 4 is found to be the most area efficient for both the subquadratic multipliers, but the delay improves linearly when halting earlier. Although the area increases exponentially, halting at 32, 16, or 8 provides a reasonable area tradeoff for the increase in speed. Figure 5.2 shows this trend comparing the area and delay tradeoffs for varying halting conditions for both the subquadratic multipliers.

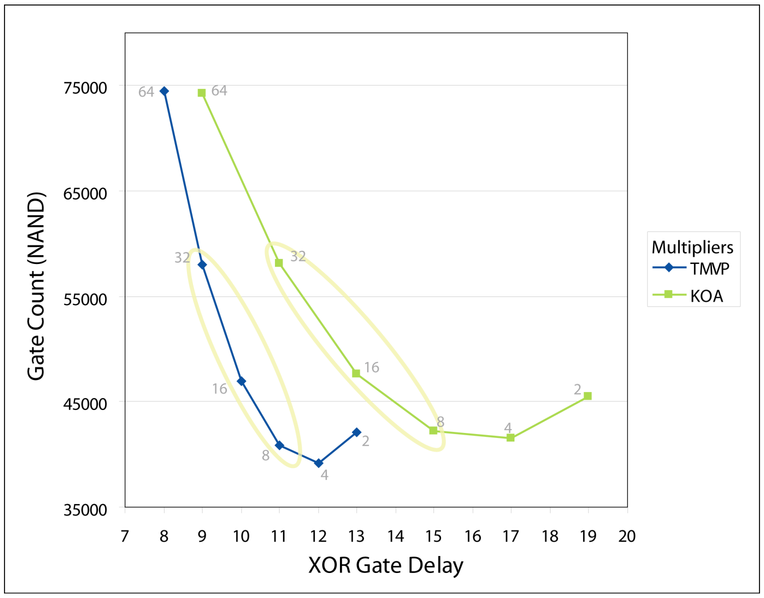

Figure 5.2: Tradeoff between Gate counts and delay for different halting conditions

The Mastrovito multiplier for GCM requires a total of 100420 NAND gates so when halting the recursion at 32, the subquadratic multiplier area would be a little less than 60% of the area requirements of the Mastrovito multiplier. For the FH multiplier, this earlier halting would result in a delay complexity of $T_A + 9T_X$ which is only a two *XOR* gate delays higher than the Mastrovito multiplier. The KA multiplier sees similar benefits, and although approaches the FH multiplier area complexities, it is still comparitively not as fast as the FH.

Appendix A

Programs for Constructing Parallel Multipliers

A.1 Java Program for Creating Generic FH Multiplier

```
import java.util.*;
/**
 * This class generates VHDL code for a generic Toeplitz Matrix Vector Product
 * Multiplier with degree 2^m
 *
 * entity TMVPMultiplier is
 * Port(
 *     A : in  Std_Logic_Vector(0 to 127);
 *     B : in  Std_Logic_Vector(0 to 254);
 *     C : out  Std_Logic_Vector(0 to 127)
 * );
 * end TMVPMultiplier;
 *
 * @author Pujan Patel
 */
public class ToeplitzMult {
    Stack signal_s;
    Stack assign_s;
    private int varcount = 0;
    int xorcount = 0;
    int andcount = 0;
    int[] xordelay;

    public ToeplitzMult() {
        signal_s = new Stack();
        assign_s = new Stack();
        varcount = 0;
        xorcount = 0;
        andcount = 0;
        xordelay = new int[8];
        for (int i = 0; i < 8; i++) {
            xordelay[i] = 0;
        }

    }
    public String[] BruteForce(String[][] T, String[] V) {
        String[] C = new String[V.length];
        for (int i = 0; i < V.length; i++) {
            C[i] = "";
            for (int j = 0; j < T.length; j++) {
                if (C[i].equals(""))
                    C[i] = "(" + T[i][j] + " and " + V[j] + ")";
                else
                    C[i] = C[i] + " xor (" + T[i][j] + " and " + V[j] + ")";
            }
        }
        return C;
    }
    /**
     *
     * @param T
     *            Toeplitz Matrix Reference
     * @param V
     *            is one of Multiplier parameters
     * @param length
     *            Length of matrix
     */
    public String[] TMVP(String[][] T, String[] V) {
        // Loop variables
        int i, j, k;
        String[][] T1, T3, T4, T0, T2;
        String[] V1, V0, V2, C, C0, C1, P0, P1, P2;
        C = null;
        xordelay[(int) Math.round(Math.log((double) (T.length)) / Math.log(2))]++;

        // End Condition
        if (T.length == 4) {

            P2 = new String[V.length];
            P2[0] = "(((" + T[0][0] + ") and (" + V[0] + ")) xor "+
                    "((" + T[0][1] + ") and (" + V[1] + ")) xor " +
                    "((" + T[0][2] + ") and (" + V[2] + ")) xor " +
                    "((" + T[0][3] + ") and (" + V[3] + ")))";
            P2[1] = "(((" + T[1][0]+ ") and (" + V[0] + ")) xor " +
                    "((" + T[0][0] + ") and (" + V[1] + ")) xor " +
                    "((" + T[0][1] + ") and (" + V[2] + ")) xor " +
                    "((" + T[0][2] + ") and (" + V[3] + ")))";
            P2[2] = "(((" + T[2][0]+ ") and (" + V[0] + ")) xor " +
                    "((" + T[1][0] + ") and (" + V[1] + ")) xor " +
                    "((" + T[0][0] + ") and (" + V[2] + ")) xor " +
                    "((" + T[0][1] + ") and (" + V[3] + ")))";
            P2[3] = "(((" + T[3][0]+ ") and (" + V[0] + ")) xor " +
                    "((" + T[2][0] + ") and (" + V[1] + ")) xor " +
                    "((" + T[1][0] + ") and (" + V[2] + ")) xor " +
                    "((" + T[0][0] + ") and (" + V[3] + ")))";
            // this.xorcount+=5; //2split counts
            // this.andcount+=3;
            this.xorcount += 12; // 4 split counts
            this.andcount += 16;
            this.varcount++;
            return P2;

        } else {

            // Tmp Variables
            T1 = new String[T.length / 2][T.length / 2];
            T0 = new String[T.length / 2][T.length / 2];
            T2 = new String[T.length / 2][T.length / 2];

            T3 = new String[T.length / 2][T.length / 2];
            T4 = new String[T.length / 2][T.length / 2];

            V1 = new String[T.length / 2];
            V0 = new String[T.length / 2];
            V2 = new String[T.length / 2];

            P1 = new String[T.length / 2];
            P0 = new String[T.length / 2];
            P2 = new String[T.length / 2];

            C = new String[V.length];
            C0 = new String[T.length / 2];
            C1 = new String[T.length / 2];
            // Fill arrays
            for (i = 0; i < (T.length / 2); i++) {
                for (j = 0; j < (T.length / 2); j++) {
                    T1[i][j] = T[i][j];
                    T2[i][j] = T[i + (T.length / 2)][j];
                    T0[i][j] = T[i][j + (T.length / 2)];
                }
            }
            for (i = 0; i < (V.length / 2); i++) {
                V0[i] = V[i];
                V1[i] = V[i + (V.length / 2)];
            }

            // Compute Additions
            AddToeplitz(T1, T0, T2, T3, T4);
            AddSingle(V1, V0, V2);
            // Comput Sub Multiplications
            P0 = TMVP(T3, V1);
            P1 = TMVP(T4, V0);
            P2 = TMVP(T1, V2);

            // Create signal for P2 in P2
            signal_s.push("signal tmp_6_" + (this.varcount)
                    + " : std_logic_vector(0 to " + (P2.length - 1) + ");");
            for (i = 0; i < P2.length; i++) {
                assign_s.push("tmp_6_" + (this.varcount) + "(" + i + ") <= ("
                        + P2[i] + ");");
                P2[i] = "tmp_6_" + (this.varcount) + "(" + i + ")";
            }
            this.varcount++;

            // Compute final C answer
            AddSingle(P0, P2, C0);
            AddSingle(P1, P2, C1);
            for (i = 0; i < (C0.length); i++) {
                C[i] = C0[i];
                C[i + (C0.length)] = C1[i];
            }
            signal_s.push("signal tmp_7_" + (this.varcount)
                    + " : std_logic_vector(0 to " + (C.length - 1) + ");");
            for (i = 0; i < C.length; i++) {
                assign_s.push("tmp_7_" + (this.varcount) + "(" + i + ") <= ("
                        + C[i] + ");");
                C[i] = "tmp_7_" + (this.varcount) + "(" + i + ")";
            }
            return C;
        }

    }
    /**
     * Prints Signal and Assignment Stacks
     *
     */
    public void printStack() {
        System.out.println("----------------------------------");
        System.out.println("-- XOR COUNT : " + this.xorcount);
        System.out.println("-- AND COUNT : " + this.andcount);
        System.out.println("----------------------------------");

        for (Iterator it=this.signal_s.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
        System.out.println("Begin");
        for (Iterator it=this.assign_s.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }

    }
    /**
     * Adds a single string array with another via xor
     *
     * @param v1 First operand
     * @param v0 Second operand
     * @param v2 Added result stored here
     */
    private void AddSingle(String[] v1, String[] v0, String[] v2) {
        // TODO Auto-generated method stub
        if (v1 == null || v0 == null || v2 == null) {
            // throw(new )
            return;
        }
        signal_s.push("signal tmp_5_" + (this.varcount)
                + " : std_logic_vector(0 to " + (v2.length - 1) + ");");
        for (int i = 0; i < v2.length; i++) {
            assign_s.push("tmp_5_" + (this.varcount) + "(" + i + ") <= ("
                    + v1[i] + ") xor (" + v0[i] + ");");
            v2[i] = "tmp_5_" + (this.varcount) + "(" + i + ")";
            this.xorcount++;
        }
        this.varcount++;

    }
    /**
     * Adds a 2x2 toeplitz matrix set of passed var and returns t3 and t4, as
     * the additions
     *
     * @param t1
     * @param t0
     * @param t2
     * @param t3
     * @param t4
     */
    private void AddToeplitz(String[][] t1, String[][] t0, String[][] t2,
            String[][] t3, String[][] t4) {
        int i, j;
        // Compute Initial Additions
        // Signal Init
        signal_s.push("signal tmp_3_" + (this.varcount)
                + " : std_logic_vector(0 to "
                + (((2 * t3.length) - 2) + t4.length) + ");");
        int count = 0;
        // Leftmost Column (bottom to top)
        for (i = t1.length - 1, count = 0; i >= 0; i--, count++) {
            assign_s.push("tmp_3_" + (this.varcount) + "(" + count + ") <= ("
                    + t1[i][0] + ") xor (" + t0[i][0] + ");");
            t3[i][0] = "tmp_3_" + (this.varcount) + "(" + count + ")";
            this.xorcount++;
        }
        // Topmost Row (left to right)
        for (i = 1; i < t1.length; i++, count++) {
            assign_s.push("tmp_3_" + (this.varcount) + "(" + count + ") <= ("
                    + t1[0][i] + ") xor (" + t0[0][i] + ");");
            t3[0][i] = "tmp_3_" + (this.varcount) + "(" + count + ")";
            this.xorcount++;
        }

        // Compute Extended Values
        for (i = 1; i < t1.length; i++) {
            for (j = 1; j < t1.length; j++) {
                t3[i][j] = t3[i - 1][j - 1];
            }
        }
        // //////////////////////////////////////////////////
        // Leftmost Column (bottom to top) (For bottom matrix)
        for (i = t1.length - 1; i >= 0; i--, count++) {
            assign_s.push("tmp_3_" + (this.varcount) + "(" + count + ") <= ("
                    + t1[i][0] + ") xor (" + t2[i][0] + ");");
            t4[i][0] = "tmp_3_" + (this.varcount) + "(" + count + ")";
            this.xorcount++;
        }
        // Topmost Row (left to right)
        for (i = 1; i < t1.length; i++) {
            t4[0][i] = "tmp_3_" + (this.varcount) + "(" + (i - 1) + ")";
        }
        // Compute Extended Values for T4
        for (i = 1; i < t1.length; i++) {
            for (j = 1; j < t1.length; j++) {
                t4[i][j] = t4[i - 1][j - 1];
            }
        }
        this.varcount++;
    }
    public String[][] Tprint(String t, int len) {
        int count, i, j;
        String[][] T = new String[len][len];
        signal_s.push("signal " + t + " : std_logic_vector(0 to "
                + ((2 * len) - 2) + ");");
        for (i = len - 1, count = 0; i >= 0; i--, count++) {
            T[i][0] = t + "(" + count + ")";
            assign_s.push(t + "(" + count + ") <= B(" + i + ");");
        }
        for (i = 1; i < len; i++, count++) {
            T[0][i] = t + "(" + count + ")";
            assign_s.push(t + "(" + count + ") <= B(" + i + ");");
        }
        for (i = 1; i < len; i++) {
            for (j = 1; j < len; j++) {
                T[i][j] = T[i - 1][j - 1];
            }
        }
        return T;
    }

    public String[][] Tprint(String t, int len, String[] up, String[] acc) {
        int count, i, j;
        String[][] T = new String[len][len];
        signal_s.push("signal " + t + " : std_logic_vector(0 to "
                + ((2 * len) - 2) + ");");
        for (i = len - 1, count = 0; i >= 0; i--, count++) {
            T[i][0] = t + "(" + count + ")";
            assign_s.push(t + "(" + count + ") <= " + up[count] + ";");
        }
        for (i = 1; i < len; i++, count++) {
            T[0][i] = t + "(" + count + ")";
            assign_s.push(t + "(" + count + ") <= " + acc[i] + ";");
        }
        for (i = 1; i < len; i++) {
            for (j = 1; j < len; j++) {
                T[i][j] = T[i - 1][j - 1];
            }
        }
        return T;
    }
    public void reset() {
        signal_s = new Stack();
        assign_s = new Stack();
        this.varcount = 0;
    }

    public static void main(String [] args) {
        int[] c;
        int i, j, k;
        ToeplitzMult tm = new ToeplitzMult();
        String[][] T = null;
        String[] V = null;
        String[]up,acc;//First column, First row of T matrix
        int ROOT = 128;
        // Create tmp V value
        V = new String[ROOT];
        up = new String[ROOT];
        acc = new String[ROOT];
        for (i = 0,j=ROOT-1; i < ROOT; i++,j--) {
            V[i] = "A(" + i + ")";
            up[i] = "T(" + j + ")";
            acc[i] = "T(" + i+ROOT + ")";
        }

        T = tm.Tprint("T", ROOT, up, acc);

        String[] C = tm.TMVP(T, V); // TMVP Multiplier
        //String []C = tm.BruteForce(T, V); //Mastrovito Multiplier

        //Print lines for creating multiplier code
        tm.printStack();
        for (i = 0; i < C.length - 7; i++) {
            System.out.println("C(" + (i + 7) + ") <= " + C[i] + ";");
        }
        System.out.println("C(0) <= " + C[i] + ";");
    }
}
```

A.2 Java Program for Creating Generic Karatsuba Multiplier

```
package tmvppack;

import java.util.Iterator;
import java.util.Stack;
/*
* entity KaratsubaMultiplier is
* Port(
*      A : in  Std_Logic_Vector(0 to 127);
*      B : in  Std_Logic_Vector(0 to 127);
*     CT : out  Std_Logic_Vector(0 to 254)
*
* );
* end KaratsubaMultiplier;
*/
public class Karatsuba {
    Stack signal_s;
    Stack assign_s;
    int varcount=0;
    int xorcount=0;
    int andcount=0;
    int [] xordelay;
    public Karatsuba(){
        signal_s = new Stack();
        assign_s = new Stack();
        varcount = 0;
        xorcount = 0;
        andcount = 0;
        xordelay = new int[16];
        for(int i=0;i<16;i++){
            xordelay[i]=0;
        }
    }
    private String[] karatsuba(String[] a, String[] b) {
        // TODO Auto-generated method stub
        String [] D0,D1,D2;
        String [] AH,AL,BH,BL;
        String [] D1A,D1B;
        //Result String
        String [] C;
        xordelay[ (int) Math.round(Math.log((double)(a.length))/Math.log(2))] ++;
        if(a.length ==4){
            C = new String[7];
            /*C[0] = "(" + a[0] +" and "+b[0]+ ")";
            C[1] = "(" + a[0] +" and "+b[1] +") xor ("+ a[1] +" and "+b[0]+ ")";
            C[2] = "(" + a[1] +" and "+b[1]+ ")";*/
            C[0] = "(" + a[0] + " and " + b[0] + ")";
            C[1] = "(" + a[0] + " and " + b[1] + ") xor " +
                   "(" + a[1] + " and " + b[0] + ")";
            C[2] = "(" + a[0] + " and " + b[2] + ") xor " +
                   "(" + a[1] + " and " + b[1] + ") xor " +
                   "(" + a[2] + " and " + b[0] + ")";
            C[3] = "(" + a[0] + " and " + b[3] + ") xor " +
                   "(" + a[1] + " and " + b[2] + ") xor " +
                   "(" + a[2] + " and " + b[1] + ") xor " +
                   "(" + a[3] + " and " + b[0] + ")";
            C[4] = "(" + a[1] + " and " + b[3] + ") xor " +
                   "(" + a[2] + " and " + b[2] + ") xor " +
                   "(" + a[3] + " and " + b[1] + ")";
            C[5] = "(" + a[2] + " and " + b[3] + ") xor " +
                   "(" + a[3] + " and " + b[2] + ")";
            C[6] = "(" + a[3] + " and " + b[3] + ")";
            // Store result in tmp variable
            signal_s.push("signal tmp_4_" + (this.varcount)
                    + " : std_logic_vector(0 to " + (C.length - 1) + ");");
            for (int i = 0; i < C.length; i++) {
                assign_s.push("tmp_4_" + (this.varcount) + "(" + i + ") <= ("
                        + C[i] + ");");
                C[i] = "tmp_4_" + (this.varcount) + "(" + i + ")";
            }
            this.varcount++;
            this.xorcount += 9;
            this.andcount += 16;

        } else {
            // Populate half terms:
            AH = new String[a.length / 2];
            AL = new String[a.length / 2];
            BH = new String[a.length / 2];
            BL = new String[a.length / 2];
            System.arraycopy(a, a.length / 2, AH, 0, a.length / 2);
            System.arraycopy(a, 0, AL, 0, a.length / 2);
            System.arraycopy(b, b.length / 2, BH, 0, b.length / 2);
            System.arraycopy(b, 0, BL, 0, b.length / 2);

            D1A = new String[a.length / 2];
            D1B = new String[a.length / 2];
            AddPoly(AL, AH, D1A);
            AddPoly(BL, BH, D1B);

            D0 = karatsuba(AL, BL);
            D1 = karatsuba(D1A, D1B);
            D2 = karatsuba(AH, BH);

            // Need to add D1,D0 and D2 into D1
            AddPoly(D1, D0, D2, D1);
            C = new String[(D0.length * 2) + 1];
            for (int i = 0; i < C.length; i++) {
                C[i] = "";
            }
            // D0 and D2 are the same length
            for (int i = 0; i < D0.length; i++) {
                C[i] = D0[i];
            }
            int index = 0;
            for (int i = 0; i < D2.length; i++) {
                index = i + (C.length - D2.length);
                if (C[index].equals(""))
                    C[index] = D2[i];
                else {
                    C[index] += " xor " + D2[i];
                    this.xorcount++;
                }
            }
            for (int i = 0; i < D1.length; i++) {
                index = i + ((C.length - 1) / 2 - ((D1.length - 1) / 2));
                if (C[index].equals(""))
                    C[index] = D1[i];
                else {
                    C[index] += " xor " + D1[i];
                    this.xorcount++;
                }
            }
            signal_s.push("signal tmp_5_" + (this.varcount)
                    + " : std_logic_vector(0 to " + (C.length - 1) + ");");
            for (int i = 0; i < C.length; i++) {
                assign_s.push("tmp_5_" + (this.varcount) + "(" + i + ") <= ("
                        + C[i] + ");");
                C[i] = "tmp_5_" + (this.varcount) + "(" + i + ")";
            }
        }
        return C;
    }

    private void AddPoly(String[] A, String[] B, String[] C, String[] D) {
        // TODO Auto-generated method stub
        signal_s.push("signal tmp_3_" + (this.varcount)
                + " : std_logic_vector(0 to " + (D.length - 1) + ");");
        for (int i = 0; i < A.length; i++) {
            assign_s.push("tmp_3_" + (this.varcount) + "(" + i + ") <= ("
                    + A[i] + " xor " + B[i] + " xor " + C[i] + ");");
            D[i] = "tmp_3_" + (this.varcount) + "(" + i + ")";
            this.xorcount += 2;
        }
        this.varcount++;
    }

    private void AddPoly(String[] A, String[] B, String[] C) {
        // TODO Auto-generated method stub
        signal_s.push("signal tmp_2_" + (this.varcount)
                + " : std_logic_vector(0 to " + (C.length - 1) + ");");
        for (int i = 0; i < A.length; i++) {
            assign_s.push("tmp_2_" + (this.varcount) + "(" + i + ") <= ("
                    + A[i] + " xor " + B[i] + ");");
            C[i] = "tmp_2_" + (this.varcount) + "(" + i + ")";
            this.xorcount++;
        }
        this.varcount++;
    }

    public void printStack() {
        System.out.println("----------------------------------");
        System.out.println("-- XOR COUNT : " + this.xorcount);
        System.out.println("-- AND COUNT : " + this.andcount);
        System.out.println("----------------------------------");
        for (Iterator it=this.signal_s.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
        System.out.println("Begin");
        for (Iterator it=this.assign_s.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }

    /**
     * Karatsuba Multiplier Design
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int i, j, k;
        int ROOT = 128;
        String[] A = new String[ROOT];
        String[] B = new String[ROOT];
        for (i = 0; i < A.length; i++) {
            A[i] = "A(" + i + ")";
            B[i] = "B(" + i + ")";
        }
        Karatsuba ks = new Karatsuba();
        String[] C = ks.karatsuba(A, B);

        //Print lines for creating multiplier code
        ks.printStack();
        for (i = 0; i < C.length; i++) {
            System.out.println("CT(" + (i) + ") <= " + C[i] + ";");
        }
    }
}
```

A.3 ASIC Synthesis Script Sample

```
set_attribute hdl_vhdl_environment common
set_attribute library /secure2/p7patel/vcs/tw13uyfsdsc_tt.lib

set_attr hdl_search_path /secure2/p7patel/GCMcode/
read_hdl -vhdl {GFPackage_lut.vhd}
read_hdl -vhdl {ABubble.vhd AESmain.vhd ToeplitzMultiplier.vhd lutsbox.vhd reglutsbox.vhd BlockCounter.vhd
             BufferBlock.vhd Chain192.vhd compositesbox.vhd CtrlBubble.vhd CtrlRegister.vhd GCMmain.vhd
             GF2_4mult.vhd GFMultBlock.vhd IVcounter.vhd KeySchedule.vhd KeyScheduleBlock.vhd
             MastrovitoMultiplier.vhd mixcolumns.vhd Mux3to1.vhd regcompositesbox.vhd round_unit.vhd
             shiftrows.vhd SingleReg.vhd StateTransform.vhd subbytesshiftrows.vhd XorBlock.vhd XorChain.vhd }

elaborate GCMmain

define_clock -name clk -period 3500 [find -port clk]
external_delay -input 0 -c clk /designs/*/ports_in/*
external_delay -output 0 -c clk /designs/*/ports_out/*

synthesize -to_mapped
synthesize -incremental
report timing
report area
synthesize -incremental -effort high
report timing
report area
synthesize -incremental -effort high
report timing
report area
report gates
write_hdl > GCM_lut_tsplit_ac_tower_area.v
write_sdc > GCM_lut_tsplit.sdc
```

References

[1] M. Alam, S. Ghosh, D. RoyChowdhury, and I. Sengupta. Single Chip Encryptor/Decryptor Core Implementation of AES Algorithm. *21st International Conference on VLSI Design, 2008. VLSID 2208.*, pages 693–698, 2008. 18

[2] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. *Fast Software Encryption, 11th International Workshop, FSE*, pages 5–7, 2004. 19

[3] J.S. Horng C.Y. Lee and I.C. Jou. Low-Complexity Bit-Parallel Multiplier over GF(2^m) Using Dual Basis Representation. *Journal of Computer Science and Technology*, 2006. 6

[4] M. Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. 2007. 20, 21, 47

[5] H. Fan and Y. Dai. Fast Bit-Parallel GF (2^n) Multiplier for All Trinomials. *IEEE Transactions on Computers*, 54(4):485–490, 2005. 6

[6] H. Fan and M.A. Hasan. A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields. *IEEE Transactions on Computers*, 56(2):224–233, 2007. 3, 35, 37, 41, 43

[7] H. Fan and M.A. Hasan. Subquadratic Computational Complexity Schemes for Extended Binary Field Multiplication Using Optimal Normal Bases. *IEEE Transactions on Computers*, 56(10):1435–1437, 2007. 6

[8] M.A. Hasan. On matrix-vector product based sub-quadratic arithmetic complexity schemes for field multiplication. *Proceedings of SPIE*, 6697:669702, 2007. 35, 38

[9] M.A. Hasan and V.K. Bhargava. Architecture for a low complexity rate-adaptive Reed-Solomon encoder. *IEEE Transactions on Computers*, 44(7):938–942, 1995. 6

[10] A. Hodjat and I. Verbauwhede. Speed-area trade-off for 10 to 100 Gbits/s throughput AES processor. *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers, 2003.*, 2:2147–2150, 2003. 45

[11] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. FCCM 2004.*, pages 308–309, 2004. 45

[12] A. Hodjat and I. Verbauwhede. Area-Throughput Trade-Offs for Fully Pipelined 30 to 70 Gbits/s AES Processors. *IEEE Transactions on Computers*, 55(4):366–372, 2006. 45

[13] Safenet Inc. SafeXcel IP-AES/GCM/XTS Accelerators Product Brief. ***http://www.safenet-inc.com/products/ip/safeXcel_IP_AESGCMXTS_Acc.asp***, 2008. 24

[14] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595, 1963. 31

[15] S. Lemsitzer, J. Wolkerstorfer, N. Felber, and M. Braendli. Multi-gigabit GCM-AES Architecture Optimized for FPGAs. *Lecture Notes in Computer Science*, 4727:227, 2007. ix, 2, 24, 25, 47, 58, 59, 61

[16] S.Y. Lin and C.T. Huang. A High-Throughput Low-Power AES Cipher for Network Applications. *Asia and South Pacific Design Automation Conference, 2007. ASP-DAC'07.*, pages 595–600, 2007. 18, 50

[17] E.D. Mastrovito. VLSI Architectures for Computation in Galois Fields, PhD Thesis. *Dept. of Electrical Eng., Linköping Univ., Sweden*, 1991. 29, 43

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

[19] D.A. McGrew and J. Viega. The Security and Performance of the Galois/-Counter Mode (GCM) of Operation. *Progress in Cryptology-Indocrypt 2004: 5th International Conference on Cryptology in Chennai, India, Proceedings*, 2004. 20, 64

[20] A.J. Menezes and I.F. Blake. *Applications of Finite Fields*. Springer, 1993. 7

[21] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. 6

[22] S. Morioka and A. Satoh. A 10-Gbps full-AES crypto design with a twisted BDD S-Box architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):686–691, 2004. 14

[23] NIST. Specication for the Advanced Encryption Standard (AES). Technical Report FIPSPUB197. 10

[24] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, 1996. 3, 31, 43

[25] A. Reyhani-Masoleh and M.A. Hasan. On Efficient Normal Basis Multiplication. *Progress in Cryptology-Indocrypt 2000: First International Conference in Cryptology in Calcutta, India, December 10-13, 2000: Proceedings*, pages 213–224, 2000. 6

[26] V. Rijmen. Efficient Implementation of the Rijndael S-box. *`http://www.esat.kuleuven.ac.be/rijmen/rijndael/sbox.pdf`, 2000.* 13

[27] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003. 20

[28] A. Satoh. High-Speed Parallel Hardware Architecture for Galois Counter Mode. *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007.*, pages 1863–1866, 2007. 2, 24, 62

[29] A. Satoh. High-speed hardware architectures for authenticated encryption mode gcm. *IEEE International Symposium on Circuits and Systems, 2006. ISCAS 2006. Proceedings.*, pages 4831–4834, 21-24 May 2006. 2, 24, 45, 49, 50, 51, 56, 60, 63, 64

[30] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. *Advances in Cryptology-ASIACRYPT*, pages 239–254, 2001. 12, 13, 14, 50

[31] A. Satoh, T. Sugawara, and T. Aoki. High-Speed Pipelined Hardware Architecture for Galois Counter Mode. *In: J. Garay et al. (Eds.) ISC, 2007. Lecture Notes in Computer Science*, pages 1863–1866, 2007. 2, 24, 63, 64

[32] Elliptic Semiconductor. CLP-15: Ultra-High Throughput AES-GCM Core 40 Gbps. *`http://www.ellipticsemi.com/products-clp-15.php`*, 2008. 24, 60

[33] C.P. Su, C.L. Horng, C.T. Huang, and C.W. Wu. A configurable AES processor for enhanced security. *Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 361–366, 2005. 50

[34] D. Whiting, N. Ferguson, and R. Housley. Counter with CBC-MAC (CCM). *NIST submission availible at: `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/`*, 2002. 19

[35] D. Whiting, B. Schneier, and S. Bellovin. AES Key Agility Issues in High-Speed IPsec Implementations. *Counterpane Internet Security, `http://www.counterpane.com/aes-agility.html`*, 2000. 3

[36] H. Wu. Low Complexity Bit-Parallel Finite Field Arithmetic Using Polynomial Basis. *Cryptographic Hardware and Embedded Systems: First International Workshop, CHES'99, Worcester, MA, USA, August 1999: Proceedings*, 1999. 6

[37] H. Wu. On Computation of Polynomial Modular Reduction. *Technical Report. Center for Applied and Cryptographic Research (CACR)*, 2000.

[38] B. Yang, S. Mishra, and R. Karri. High Speed Architecture for Galois/Counter Mode of Operation (GCM). *http://eprint.iacr.org/2005/146.pdf*. 47, 50

[39] G. Zhou, H. Michalik, and L. Hinsenkamp. Efficient and High-Throughput Implementations of AES-GCM on FPGAs. *International Conference on Field-Programmable Technology, 2007. ICFPT 2007.*, pages 185–192, 2007. 2, 24, 58, 61