A Pipelined Memory less 13.5 Gps AES128 Decryptor

Thesis submitted in Fulfillment of the requirements for the degree of

MASTERS OF TECHNOLOGY (HONS.)

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By
Bhaben Deori
03 CS 3020

Under the guidance of
Prof Dipanwita Roychoudhury

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
KHARAGPUR-721302, INDIA

Contents

ABSTRACT

This thesis presents a fully pipelined implementation of the Advanced Encryption Standard decryption algorithm with 128-bit input and key length (AES-128), implemented on Xilinx' Virtex-E and Virtex-II devices. In the present work, architecture is developed which aims to achieve a high throughput, lesser IO lines, on-chip key generation for physical security. The design implements a combinational logic based Rijndael S-Box implementation for the Inv Sub Byte transformation in the Advanced Encryption Standard (AES) algorithm for Field Programmable Gate Arrays (FPGAs). The Memory less pipelined architecture achieves a speed of 13.5Gbps @ 107 MHz clock. An architecture has been implemented for mix column and inverse mix column which optimizes hardware overhead. Intelligent clocking has been performed to reduce the power. The width of the data input and output buses are 32 bits for easy interface. The key scheduling architecture has been multiplexed to prevent external attack during the normal functioning of the chip.

Chapter 1

1. INTRODUCTION:

The importance of cryptography is constantly increasing, since the amount of sensitive data being transmitted over open environments is growing at an unprecedented pace. Software-based implementations of cryptographic algorithms fall short of the required performance, as the transmission speeds of core networks reach the gigabits per second (Gbps) range. The significance and applicability of hardware-based implementations of cryptographic algorithms is therefore of interest also to the Field Programmable Gate Array (FPGA) design community. FPGAs are nearly ideal candidates for high-speed cryptography for several reasons. The target market is generally low- to medium sized, which makes the usage of Application Specific Integrated Circuits (ASIC) less attractive because of the large initial costs included in starting a ASIC manufacturing process. FPGA-designs also have a quicker time-to-market cycle than ASICs. A programmable platform has also applications in a multi-protocol environment, such as IPSec [5], since the cryptographic algorithm to be used can be configured on-the-fly to the target device in a fraction of a second.

The National Institute of Standards and Technology (NIST) of the United States announced in 1997 an Advanced Encryption Standard (AES) development effort to replace the Digital Encryption Standard (DES). There were five candidates in the last round of the AES algorithm selection process: MARS, RC6, Rijndael, Serpent and Twofish. In autumn 2000 the Rijndael algorithm, developed by Joan Daemen and Vincent Rijmen [4], was selected as the AES algorithm. AES was formally published on November 26 2001 in Federal Information Processing Standards' (FIPS) publication FIPS-PUB 197 [7]. The standard became effective on May 26, 2002.

The implementation of fully unrolled secret-key cryptographic algorithms is feasible on million-gate FPGAs. If the entire algorithm with full inner and outer loop pipelining fits on a single FPGA, the limiting factor for throughput is the achieved clock rate as follows:

$$\textbf{\textit{Throughput = block size x frequency}}$$

Since the block size of AES is fixed at 128 bits, a 100 MHz clock rate implies a throughput of 12.8 Gbps. Clock rates above 100MHz should be achieved in modern FPGAs by partitioning the design into stages and pipelining the entire system.

A typical feature of modern FPGAs is the inclusion of embedded internal memory within the device, for example BlockRAMs in Xilinx' Virtex devices and Embedded System Blocks (ESBs) in Altera's Apex devices . This has several benefits, since lookup tables and conversion functions can be easily implemented as small RAMs within the device. However, the amount of available internal memory may also become a bottleneck when implementing a heavily pipelined design where each stage of the pipeline requires its own unshared memory block. This may be the case with fully pipelined secret-key cryptographic algorithms, for example DES and AES, which implement non-linear substitutions with so-called S-boxes. In these cases, a smaller and less expensive target device requires implementing the design in an entirely combinatorial manner without resorting to memory accesses.

1.1 MOTIVATION OF THE WORK:

Over the years many FPGA and ASIC implementations of Rijndael have been reported. Most of them have used look up tables to implement S-Boxes. Although some ASIC implementation can support any combination of block and key length (128,192,256) it incurs much more hardware complexity to implement LUT based S-Boxes. The advent of composite field GF $(2^4)^2$ arithmetic S-Box was first seen in the works of Rijmen and Rudra et. al.

All works mentioned above either concentrated on either high throughput or area optimization. However in real life applications there are many issues to be addressed such as high throughput without compromising on area, lesser IO lines (standard 32 bits interface which is used in most processors), physical security and maximum utilization of symmetry (reusability of modules/hardwares) in the architecture.

1.3 OBJECTIVE AND DESIGN ISSUES:

Our objective is to come up with an architecture addressing the issues of throughput, easy IO interface, physical security and low area. In the present work, architecture is developed which aims to achieve a high throughput, lesser IO lines, on-chip key generation for physical security. The design has been implemented in Galois Subfield which leads to memory-less architecture. An architecture has been implemented for mix column and inverse mix column to optimize hardware overhead. Intelligent clocking has been performed to reduce the power. The width of the data input and output buses are 32 bits for easy interface. The key scheduling architecture has been multiplexed to prevent external attack during the normal functioning of the chip.

Chapter 2

REVIEW OF RELATED WORKS

In this chapter we present a survey of works related to the present work. First one deals in detail about the AES algorithm and the second one deals with efficient implementation which consider the optimization of area, speed. A discussion with merits and demerits is presented in this section.

2.1 The AES Algorithm

The Advanced Encryption Standard (AES) algorithm is a symmetric block cipher that processes data blocks of 128 bits using cipher keys with lengths of 128, 192 and 256 bits. The AES algorithm is also called the Rijndael algorithm named after its inventors, Joan Daemen and Vincent Rijmen. In the present work, only the 128 bit decryption version is considered. A detailed specification of the AES algorithm, including AES-192 and AES-256 can be found in [7]. In the following chapters, the description generally concentrates on AES-128, but whenever the description is valid for all variants of AES, the generic abbreviation AES is used.

Data is handled mainly as bytes in the AES algorithm. One byte forms an element in a polynomial representation of Galois Field GF (2^8). A byte can be represented in hexadecimal notation as {ab}, where a represents the four most significant bits (MSB) and b represents the four least significant bits (LSB) of the byte.

In this document, the representation used in the official standard is called F1, formally defined as GF(2) [x]=m(x), where m(x) is an irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

Additions are performed as bitwise XORs between operands in polynomial representations of F1. Multiplications in F1 are performed as a multiplication of the regular polynomials. The multiplication result can be a 14-degree polynomial which doesn’t fit into a byte. Thus the final multiplication result in F1 is the result of the polynomial multiplication modulo m(x).

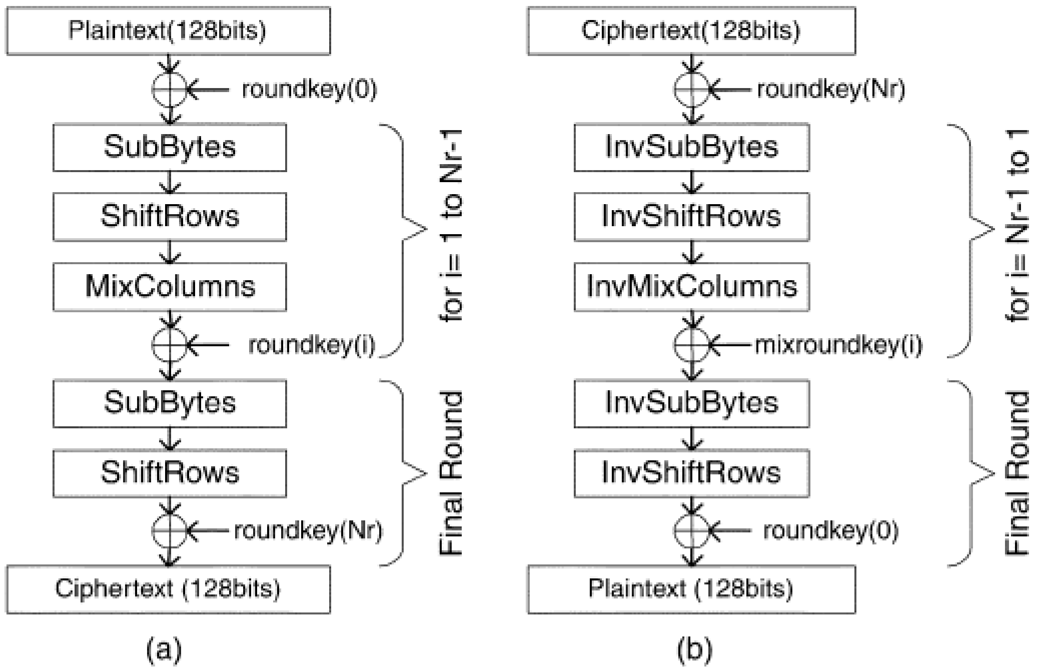
128-bit data block and key are considered as a byte array with four rows and four columns. AES-128 consists of ten rounds. One AES encryption round includes four transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey. The first and last round differ from other rounds in that there is an additional AddRound- Key transformation at the beginning of the first round and no Mix- Columns transformation is performed in the last round. Key Expansion in the AES algorithm calculates RoundKeys based on the original cipher key. The RoundKeys are needed in AddRoundKeys. In AES-128 encryption, the first RoundKey used in the additional AddRoundKey at the beginning of the first round is always the original key. Intermediate results after every transformation are calledStates. Of all the transformation above, the ByteSub is most computationally heavy.

2.2 THE STATE, THE CPHER KEY AND THE NUMBER OF ROUNDS

State: the intermediate cipher result is called the State. The State can be pictured as a rectangular array of bytes. This array has four rows; the number of columns is denoted by Nb and is equal to the block length divided by 32.The Cipher Key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by Nk and is equal to the key length divided by 32. These representations are illustrated in the Figure

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$



2.3 ALGORITHM OPERATIONS

2.3.1 SubByte and InvSubByte

SubByte performs the nonlinear byte-wise substitution.The SubByte transformation is computed by taking the multiplicative inverse followed by an affine transformation. For its reverse, the InvSubByte transformation, the inverse affine transformation is applied first prior to computing the multiplicative inverse.
The steps involved for both transformation is shown below:

SubByte: Multiplicative Inversion in GF (2^8) > Affine Transformation

InvSubByte: Inverse Affine Transformation > Multiplicative Inversion in GF (2^8)

The Affine Transformation and its inverse can be represented in matrix form and it is shown as

$$AT(a)=\begin{pmatrix}1&1&1&1&1&0&0&0\\0&1&1&1&1&1&0&0\\0&0&1&1&1&1&1&0\\0&0&0&1&1&1&1&1\\1&0&0&0&1&1&1&1\\1&1&0&0&0&1&1&1\\1&1&1&0&0&0&1&1\\1&1&1&1&0&0&0&1\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\1\\1\\0\\0\\0\\1\\1\end{pmatrix}$$

$$AT^{-1}(a)=\begin{pmatrix}0&1&0&1&0&0&1&0\\0&0&1&0&1&0&0&1\\1&0&0&1&0&1&0&0\\0&1&0&0&1&0&1&0\\0&0&1&0&0&1&0&1\\1&0&0&1&0&0&1&0\\0&1&0&0&1&0&0&1\\1&0&1&0&0&1&0&0\end{pmatrix}\times\begin{pmatrix}a_7\\a_6\\a_5\\a_4\\a_3\\a_2\\a_1\\a_0\end{pmatrix}\oplus\begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\1\end{pmatrix}$$

The AT and AT^{-1} are the Affine Transformation and its Inverse respectively while the vector **a** is the multiplicative inverse of the input byte from the state array. From here, it is observed that both the SubByte and the InvSubByte transformation involve a multiplicative inversion operation. Thus, both transformations may actually share the same multiplicative inversion module in a combined architecture. An example of such hardware architecture is shown below. Switching between SubByte and InvSubByte is just a matter of changing the value of INV. INV is set to 0 for SubByte while 1 is set when InvSubByte operation is desired.

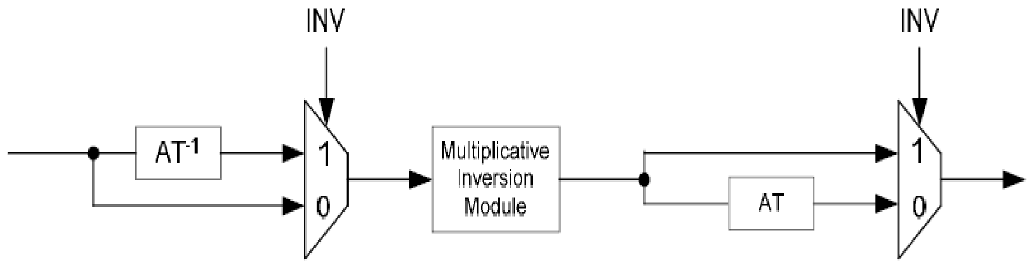


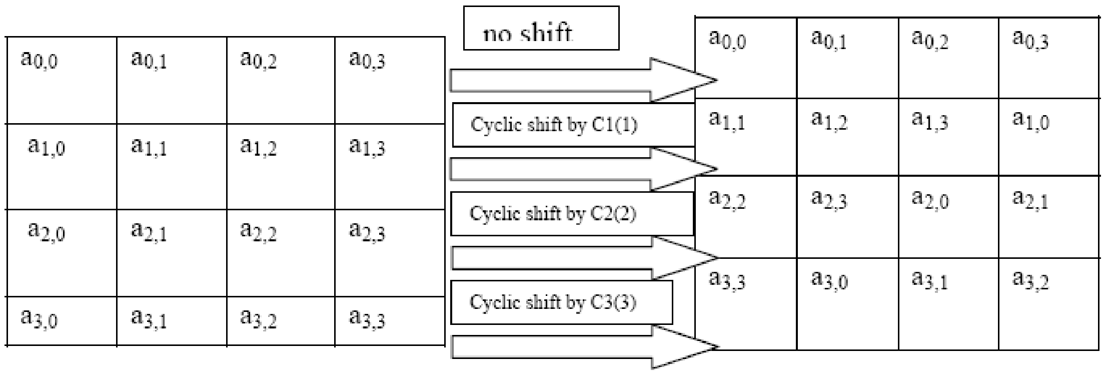
Figure 1.1. Combined SubByte and InvSubByte sharing a common multiplicative inversion module.

2.3.2 SHIFT ROW AND INV SHIFT ROW

The ShiftRows transformation performs a cyclical left shift on the last three rows of the State. The first row is not shifted. The second row is shifted one byte, the third row is shifted two bytes and the fourth row is shifted three bytes. The same operation is done in InvShiftRow but is shifted right. Thus, ShiftRows proceeds as follows:

$$s_{r,c} = s_{r,(c+r) mod 4} \quad 0< r < 3 \text{ and } 0 <c < 3;$$

where $s_{r,c}$ is the byte (row r, column c) of the State.



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
no shift
Cyclic shift by C1(1)
Cyclic shift by C2(2)
Cyclic shift by C3(3)
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

2.3.3 MIX COLUMN AND INVMIX COLUMN

The MixColumns transformation operates separately on every column of the State. A column is considered as a polynomial over F1 and multiplied modulo $x^4 + x + 1$ with the polynomial

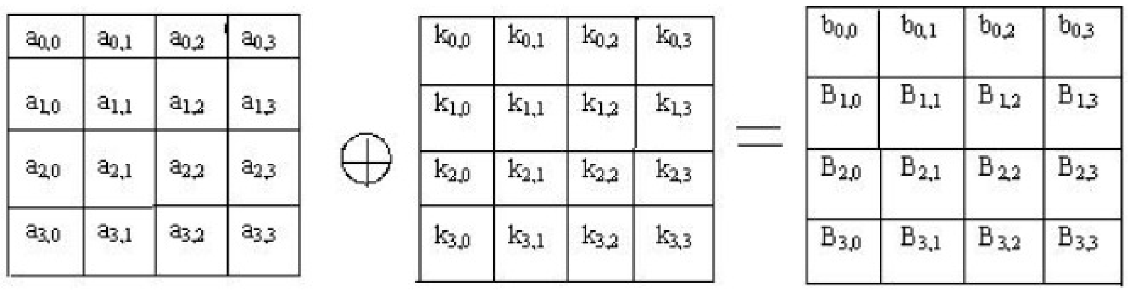
$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

The transformations in the decryption process perform the inverse of the corresponding transformations in the encryption process. Specifically, the InvMixColumns transformation multiplies the polynomial formed by each column of the State with $a^{-1}(x)$ modulo $x^4 + 1$, where

$$a^{-1}(x) = \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}x + \{0e\}$$

2.3.4 ADD ROUND KEY

The Round Key addition is the last element of every round and a Round Key is applied to the State by a simple bitwise XOR. The Round Key is derived from the key schedule, which we'll talk later. Note this implies that Round Key addition is its own reverse. The operation is represented in Figure


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$\oplus$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
$=$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

2.3.5 KEY EXPANSION

The Key Expansion calculates RoundKeys for every AddRound- Key transformation. In AES-128 encryption, the original cipher key is the first RoundKey rk[0] used in the additional AddRound- Key at the beginning of the first round. RoundKey rk[i], where $i > 0$, is calculated from the previous RoundKey rk[i -1].

Let p[j], where $0 < j < 3$, be the column j of the previous Round- Key rk[i-1] and let w[j] be the column j of the RoundKey being calculated. Then the new RoundKey rk[i] is calculated as follows:

w[0] = p[0] xor (RotWord(SubWord(p[3])) xor rcon[i])

w[1] = p[1] xor w[0]

w[2] = p[2] xor w[1]

w[3] = p[3] xor w[2]

RotWord() is a function that takes a four byte input [a0;a1;a2;a3] and returns it rotated: [a1;a2;a3;a0]. The function SubWord() performs a SubBytes transformation for four bytes. The Round constant rcon[i] contains values [x^{i-1}; {00}; {00}; {00}] where x^{i-1} are the powers of x (x is denoted as {02}) in F1.

2.4 AES DESIGN CONSIDERATIONS:

The central design principle of AES algorithm is simplicity [25]. Simplicity facilitates implementations on different platforms under different sets of constraints. The simplicity is realized by two means: adoption of symmetry at different levels and choice of basic operations. The first level of symmetry lies in the fact that AES algorithm encrypts/decrypts 128 bits block of plaintext by repeatedly using the same round transformation as outlined in the above fig. AES-128 applies 10 rounds, AES-192 applies 12 rounds and AES-256 applies 14 rounds. Symmetry can be found within the definitions of round transformations of AES. The symmetry in the structure allows the reuse of hardware components leading to economic implementations. The basic operations in AES can be very easily defined in terms of the operations defined over the finite field GF (2^8). This property allows us to reason about the algorithm using established mathematical techniques, facilitating security analysis as well as construction of optimal implementations. Moreover finite field arithmetic can be very efficiently implemented in hardware as compared to integer arithmetic.

2.5 HARDWARE APSECTS OF AES

AES operations are byte oriented. So they can be executed efficiently on 8 bit processors. On 32 bit processors also AES is important because some 8 bits operations can be combined to form 32 bits operations. In hardware implementation any word size is suitable. Most hardware implementations prefer 128 bit architecture. This offers the greatest degree of parallelism to increase concurrency of AES computations. A higher degree of concurrency offers higher throughput.

The size of the architecture defines the size of the circuit. A 32 bit architecture will require 4 S-Boxes to compute the Sub Byte operation. S-Boxes are the most spacious component of the AES implementation. Use of an efficient architecture is important for AES hardware module. Basic option is to use look up tables using ROMs. Another technique is to compute S-Box output by first computing the multiplicative inverse and then the affine transformation. Wolkerstorfer [26] pointed out that use of combinational logic can do this as efficiently as ROMs can do table look ups. In particular use of combinational logic is superior in many cases of AES hardware implementation when decryption is needed too.

The storage requirement of an AES implementation also has an impact on the overall size of the circuit. It requires at least 256 bits: 128 bits to store the state and 128 bit key to store the actual round key. On some platforms it is more efficient to store additional memory. On Xilinx FPGA, duplication of the State can reduce the overall hardware cost. Memory considerations are also important regarding the round key generation. On area efficient hardware implementation round keys are generated on the fly. Software implementations on 32-bit platforms generate the round keys before hand thus saving time.

2.5.1 LUT BASED APPROACHES

Byte Sub can be calculated using an S-Box (LUT) which contains precalculated values of the transformation. One 256 x 8 bit S-Box is required for each byte of the state. Therefore 16 parallel S-Boxes are required if Byte Sub is performed for the entire state at once. If S-Boxes are implemented on Xilinx FPGAs using Block RAMs, 100 block RAMs are needed, because one dual port block RAM can implement two S-Boxes. Block RAM based S-box implementations have been reported in many implementation[27,28].

In addition to Byte Sub, Mix Column can be implemented using LUT approach. An LUT combining Byte Sub and Mix Column is called a T Box. Fisher and Drutarovsky studied implementation techniques based on S-Boxes and T-Boxes on an Altera FPGA[29]. They concluded that slightly faster performance was attained with the T-Box but the memory requirement increased.

2.5.2 COMPOSITE FIELD BASED APPROACHES

Rijndael involves arithmetic on GF (2^8) elements. In the straightforward implementation inversion, multiplication and substitution are the operations that are determine the overall complexity. The most common approach is to use table look up approach for these operations. A major drawback is that the size of the memory may be the bottle neck. Another approach is to use combinational logic to implement the multiplicative inversion and affine transformation for the Byte Sub transformation. Inversion in GF (2^8) can be implemented using inversion in GF (2^4) or GF (2^2) accompanied by Galois field addition and multiplication.

In particular, composite field inversions are found to be more effective than GF (2^8) and are used to implement compact AES implementation[15]. In contrast to Rijmen's proposal which suggests the optimal normal basis representation of finite field representation, the use of polynomial representation of finite field elements results in far more flexible architecture without the necessity of the complex conversion from one representation to another.

The most popular technique used in composite field implementation of Byte Sub operation is:

- Map the elements of GF (2^8) to the composite field F using an isomorphic Function (ø)
- Compute the multiplicative inversion over the field F.
- Finally map the computation back into the original field using the inverse isomorphic function ($ø^{-1}$)

2.5.3 AREA OPTIMIZED APPROACHES

The highest benefit of combinational implementation of Byte Sub is that it can be pipelined and high throughput can be achieved. This however increases the latency of the implementation. The slice requirements also increases compared to block RAM implementation because the Byte Sub is implemented using logic. In many applications it is important to optimize area than to optimize throughput.

FeldHofer et al [30] implemented 128 bit AES "on a grain of sand". It is a 8-bit architecture which exploits composite field GF$(2^4)^2$ for S-Box optimization. Pramsteller implemented AES encryption and decryption with all key lengths. It used a novel state representation which solves the problem of accessing both the rows and column of the state.

2.6 PERFORMANCE COMPARISON OF DIFFERENT DESIGNS

The performance comparisons of different AES implementation is hard due to various reasons. First the large variety of target devices available makes a fair comparison difficult. Second, many authors don't specify their device well enough to ensure easy comparison eg. size or the speed grade of the device is not provided. Third area comparison is difficult because both slices and embedded memory i.e block RAMs in Xilinx is used. Performance on different devices is not ideal to compare because performance of the implementation is greatly determined by the device.

Chapter 3

GALOIS FIELD ARITHMETIC AND COMPOSITE FIELD

Composite fields are frequently used in the implementation of Galois Field Arithmetic. In cases where arithmetic operations involve table look up, subfield arithmetic is used to reduce look up related costs. This technique has been used to obtain relatively efficient implementation of specific operations such as multiplication, inversion and exponentiation.

The two pairs {GF (2^n),Q(y)} and {GF(2^n)^m, P(x)} constitute a composite field if GF (2^n) is constructed from GF(2) by Q(y) and GF(2^n)^m is constructed from GF (2^n) by P(x), where Q(y) and P(x) are polynomials of degree n and m respectively. The fields GF(2^n)^m and GF (2^k), k=nm are isomorphic to each other. Since the complexity of various arithmetic operations differs from one of these fields to other, we can take advantage of the isomorphism to map a computation from one to the other in search of efficiency. For a given underlying field of GF, our gain depends on the choice of n and m respectively as well as of the polynomials Q(y) and P(x).

3.1 RIJNDAEL IN A COMPOSITE FIELD:

Rijndael involves arithmetic on GF (2^8) elements. In a straightforward implementation, inverse, multiplication and substitution are likely to be the operations that determine the overall complexity of the implementation.

3.2 ISOMORPHISM BETWEEN COMPOSITE FIELDS F1 and F2:

The SubBytes/InvSubBytes transformation of the AES algorithm can be implemented with lookup tables located in Block-RAMs. This has obvious benefits, but in designing a fully pipelined design, the amount of available internal memory may become a bottleneck. Consequentially, a more expensive target device may be needed if every SubBytes transformation is implemented as a lookup table.

Instead of the table implementation in F1 it was decided to perform the SubBytes transformation by calculating the multiplicative inverse of the SubBytes in F2:= GF (2^4) [x] =(x^2 +Ax+B)

To make this work a byte representing an element in F1 must be transformed to a byte representing an element in F2. All multiplications in GF (2^4) are performed in GF (2) [y]= (y^4 +y+1). Constants A and B can be chosen freely as long as x^2 +Ax+B is irreducible. The problem is to find the isomorphism $\emptyset$: F1$\rightarrow$F2.

3.3 FINDING A TRANSFORM:

We show the construction of the conversion matrix *T* from the composite field *GF* $(2^4)^2$ to binary field *GF* (2^8). Let *GF* (2^8) be constructed using the primitive polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$ and α be a root of *p(x)*, thus α is a primitive element in *GF* (2^8). We have $\gamma = \alpha^r$ is a primitive element in the ground field *GF* $(2^4)^2$.

We construct the composite field *GF* $(2^4)^2$ over the field *GF* (2^4) using the irreducible polynomial $q(x)$. The irreducible polynomial *q(x)* which is of degree 2 and its coefficient are from the ground field *GF* (2^4). In order to represent the elements of the ground field *GF* (2^4), we use the constant term in *q(x).*

An element A is expressed in basis *B2* as

$$A = a'_0 + a'_1\alpha$$

where $a'_j \in GF(2^4)$. We can express a'_j using γ as the basis element

$$a'_j = a''_{j0} + a''_{j1}\gamma + a''_{j2}\gamma^2 + a''_{j3}\gamma^3$$

where $a''_{ji} \in GF(2)$ for j = 0 ,1 and i=0, 1,2,3. Therefore, the representation of *A* in the composite field is found as:

$$A = a''_{00} + a''_{01}\alpha^{17} + a''_{02}\alpha^{34} + a''_{03}\alpha^{51} \\ + a''_{11}\alpha + a''_{12}\alpha^{18} + a''_{13}\alpha^{35} + a''_{13}\alpha^{52}$$

The next step is to reduce the terms α^{17i+j} for *j*= 0, 1 and *i*=0,1,2,3 using the generating polynomial $p(x)= x^8 + x^4 + x^3 + x + 1$. This will give us α terms in the above expression with exponents between 0 and 7. A term of the form α^{17i+j} is reduced modulo *p(x)* by successively using the relation $\alpha^8 = \alpha^4 + \alpha^3 + \alpha + 1$. We obtain the representation of *A* in the binary field *GF* (2^8) using the basis B1={ 1, α,......... α^7}as**:**

$$\mathbf{A} = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 + a_5\alpha^5 + a_6\alpha^6 + a_7\alpha^7 + a_8\alpha^8$$

The relationship between the terms a_h for h=0,1,2…7 and a''_{ji} for j=0,1 and i=0,1,2,3 determines the elements t_{jih} of the conversion matrix T. The first row of the matrix T is obtained by gathering the constant terms in the right-hand side after the substitution, gives the constant coefficient in the left hand side, i.e., the term a0.

The transformation matrix ø$^{-1}$: F2→F1 is expressed in matrix form as:

$$\Phi^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}.$$

The inverse of the above transformation Ø: *F1* →*F*1 is defined by inverting the matrix F with the result as follows:

$$\Phi = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

In addition to the multiplicative inverse also other transformations in the AES-128 decryption algorithm are calculated in F2. This makes decryption faster and saves significant amounts of space since the transformations ø and ø -1are performed only once. The transformation ø is performed for both the key and data block at the beginning of the decryption and the inverse transformation ø$^{-1}$ is performed for the decrypted data block at the end of the last round. The next subsections describe how the mapping of InvSubBytes, InvMixColumns, AddRoundKey, InvShiftRows and Key Expansion to F2 was performed.

3.4 SUB BYTE AND INVSUB BYTE in F2:

If a byte is mapped to F2 with the transformation F, the multiplicative inverse can be calculated as follows

$$(bx + c)^{-1} = b\,(b^2\lambda + c\,(b+c))^{-1}\,x + (c+b)\,(b^2\lambda + c\,(b+c))^{-1}$$

where b are the four most significant and c the four least significant bits of the byte. As already mentioned, it was chosen that A = 0b0001 = {1} and λ = 0b1000 = {8}. Also the affine transformation defined by Equation (1.1) must be mapped to F2. Since ø is also a linear transformation, the affine transformation can be calculated as follows:

Let $b' = Tb + c$ be the affine transformation in F1 and
Let $b_ø' = T_ø\, b_ø + c_ø$ be the affine transformation in F2

Now because $b' = ø^{-1}\, b_ø' = ø^{-1}(T_ø\, b_ø + c_ø) = ø^{-1}(T_ø\, øb + c_ø)$

Or $\quad b' = (ø^{-1}T_ø\, ø)b + ø^{-1}c_ø$

Comparing with above we get,

$$T = = ø^{-1}T_ø\, ø$$

And similarly, $c_ø = øc$

The Affine Transformation in F2 can therefore be expressed as:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}.$$

And similarly for inverse affine transformation.

3.5 INVERSE MIX COLUMN:

The InvMixColumns transformation of the AES-128 decryption algorithm must also be mapped to F2. The addition in F2 is calculated in a similar fashion as in F1 (that is, by bitwise XORing the operands), and therefore only the multiplications must be mapped to F2.

The InvMixColumns transformation multiplies the polynomial formed by each column of the State with with $a^{-1}(x)$ modulo $x^4 +1$, where

$$\mathbf{a^{-1}(x) = \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}x + \{0e\}}$$

And in Mix Column a column is considered as a polynomial over F1 and multiplied modulo $x^4 +x+1$ with the polynomial

$$\mathbf{a\,(x) = \{03\}x^3 + \{01\}\, x^2 +\{01\}x+\{02\}}$$

We use different procedure to implement the Inv Mix Column. The Mix Column multiplication will be used to implement the Inv Mix Columns Because F maps {01} to {01} it suffices to map only the multiplications with {02}.Writing,

$$\mathbf{a = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0}$$

multiplication with {02}* a in F1 can be calculated as:

$$\begin{aligned}\{02\} * a &= x\,(a7x^7 + a6x^6 + a5x^5 + a4x^4 + a3x^3 + a2x^2 + a1x + a0)\\ &= a7x^8 + a6x^7 + a5x^6 + a4x^5 + a3x^4 + a2x^3 +\\ &\qquad a1x^2 + a0x \bmod x^8 + x^4 + x^3 + x + 1\\ &= a6x^7 + a5x^6 + a4x^5 + (a3+ a7)x^4 + (a2 + a7)x^3 +\\ &\qquad a1x^2 + (a0 + a7)x + a7\end{aligned}$$

And multiplication {04} * a results in the following equation:

$$\begin{aligned}\{04\} * a = {}& a5x7 + a4x6 + (a3+a7)\ x5 + (a2 + (a6+a7))\ x4\\ & + (a1+a6)\ x3 + (a0+a7)\ x2 + (a6+a7)\ x + a6\end{aligned}$$

In matrix form the above multiplication is expressed as

$$\{02\} \bullet \mathbf{a} = M_2\mathbf{a} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

The matrices Mø2 can be expressed in F2 and can be calculated from M2 as follows:

$$M_{\phi 2} = \mathbf{\Phi} M_2 \mathbf{\Phi}^{-1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

3.6 ADD ROUND KEYS AND INVSHIFT ROWS IN F2

Because addition is calculated as a bitwise XOR in both F1 and in F2 there is no need for changes in the AddRoundKey transformation. Also the ShiftRows transformation remains unchanged, because no calculations are required there.

3.7 KEY EXPANSION IN F2:

In the Key Expansion, the function SubWord() and the round constant rcon[i] must be mapped to F2. SubWord(), which consists of four SubBytes, is mapped as described earlier.The rcon[i] values (powers of x) are mapped to F2 by multiplying them with the matrix ø. The values of rcon[i] are presented in Table 1. All the transformations of the AES-128 encryption algorithm have now been mapped from F1 to F2.

All the transformations of the AES-128 decryption algorithm have now been mapped from F1 to F2. The decryption can be implemented as follows: first both the 128-bit data block and the 128-bit key are mapped to F2 with the transformation Ø and then the decryption is carried out as described above. At the end of the last round the encrypted data is mapped back to F1 with the inverse transformation $Ø^{-1}$.

Chapter 4

DESIGN OVERVIEW

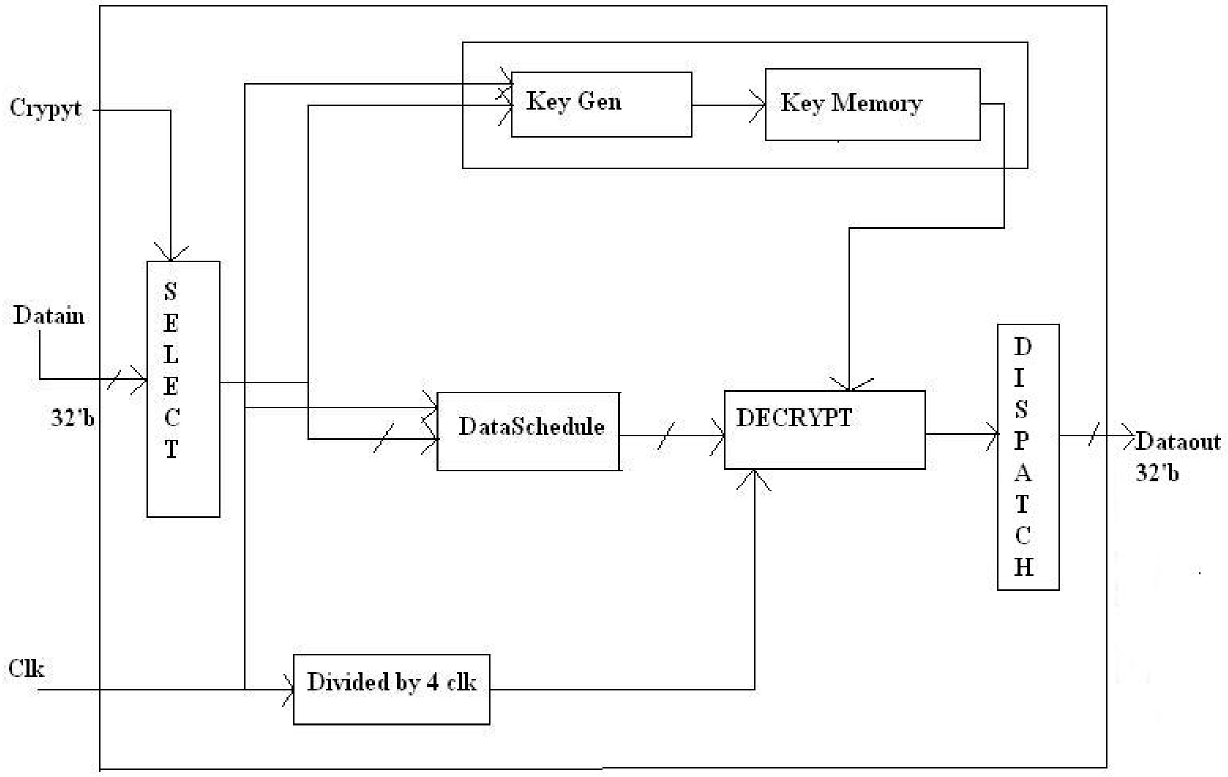
The proposed design is based upon Rijndael private-key Cryptosystem. The design contains an on-chip Key Generation Unit which generates the round keys used in the crypto-system. The architecture comprises of a full-ten round implementation of 128-bit block decryption. The 32-bit data input pins have been multiplexed between the key-generation unit and the text inputs to the crypto-system.

In the present work, all the ten rounds of the cryptosystem have been unrolled, so that there are ten blocks of data which are processed at the same time. This technique is known as outer loop pipelining. The number of rounds unrolled increases the throughput proportionately. Inside the rounds the concept of inner loop pipelining has been studied. Registers inside the cipher rounds have been found to significantly increase the cipher throughput but at the expense of significant increase in area. The S-Box in Rijndael poses a heavy burden on the area as well as the processing time. To reduce this overhead all computations involved in the S-Box have been performed in Galois composite field.

The input data is streamed in 32 bits and is converted into a 128-bit block by the Data Scheduler at the expense of four clock edges. The data block is decrypted using a slower clock (divided by four). Finally, the 128 bit block is being streamed out in units of 32 bit by the Dispatch Unit as output data. It may be mentioned that the Data Scheduler converts the input data from to GF (2^8) to GF $(2^4)^2$ and after the processing, the Dispatch Unit converts the elements back to GF (2^8).

4.1 TOP LEVEL ARCHITECTURE

The schematic view of the top level architecture is shown in the next page.



4.2 DESIGN CONSTAINTS:

The present subsection gives an overview of the constraints under which the design was performed. The design strategies adopted to achieve the constraints are detailed in the sections 3-8. The architecture has been developed in a step by step method to achieve an efficient architecture.

It is intended that the implementation should provide a physically secured key generation unit. The memory less design architecture and the novel implementation of the InvMixColumns using Galois Subfield helps in reducing the area without imposing much penalty on the throughput.

Another important issue is to provide an easy interface by reducing the pin counts.
A 32'b data input-output interface is to be achieved. Finally the design for testability issue should be carefully handled to support both structural and functional modes of testing.

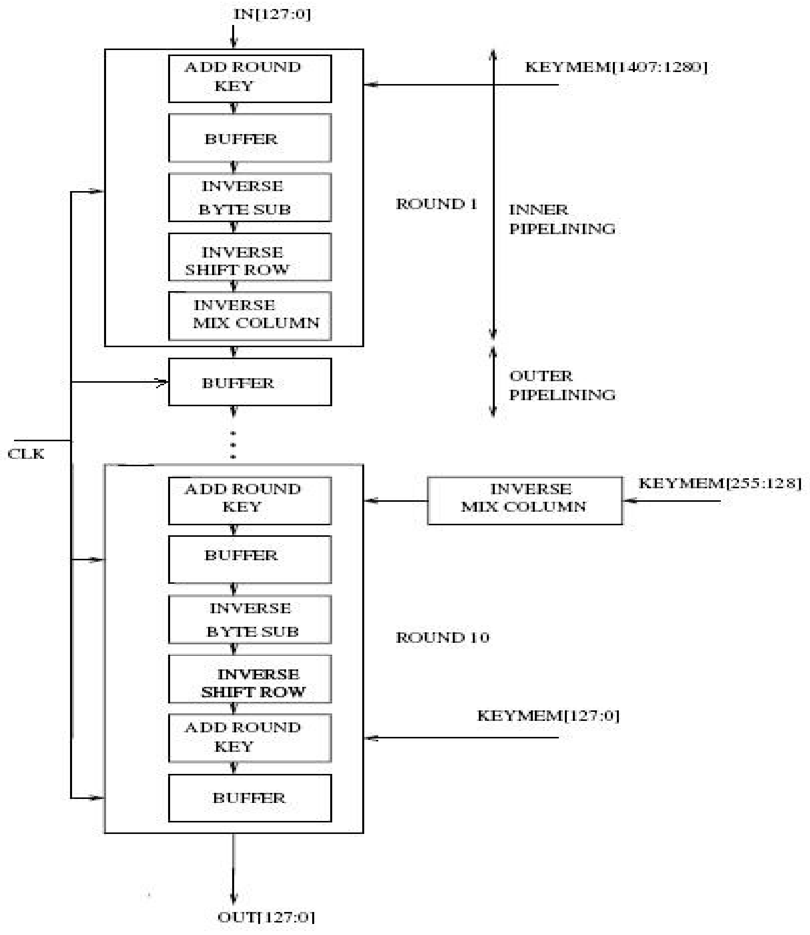
4.3 THE DECRYPTING UNIT

In a direct Inverse Cipher the sequences of transformations differ from that of the Cipher, while the same KeyScheduling is used. However, the inverse cipher may be manipulated to have the same flow or sequence[5]. The following properties are utilised to serve the purpose:

- The Inverse SubByte and the Inverse ShiftRow being operations on separate bytes may be interchanged.
- The Add Round Key and the Inverse Mix Columns can be reversed.

We have the following fact,
InvMixColumns(state xor RoundKey) = InvMixColumn(state) xor InvMixColumn(RoundKey)
since both the operations are linear.

Thus the Round Keys require a further transformation through the InverseMixColumn Unit. The operations are not performed for the first or the last 4 words of the KeyMemory, since those do not work with an InvMixColumn.Figure below describes the decrypting unit.



Chapter 5

DETAILED HARDWARE IMPLEMENTATION ARCHITECTURES

In this section, we present detailed architectures for each of the nontrivial transformations in the AES decryption algorithm.

5.1 InvSubBytes IMPLEMENTATION

The multiplicative inversion in involved in the Sub-Bytes/InvSubBytes is a hardware demanding operation, it takes at least 620 gates to implement by repeat multiplications in [13]. However, the gate count can be reduced greatly by using composite field arithmetic.

The InvSubBytes has essentially two steps:

1. InvAffine Transformation in GF (2^4)
2. Multuplicative inversion in GF (2^4)

The InvAffine transformation is already described in earlier section.

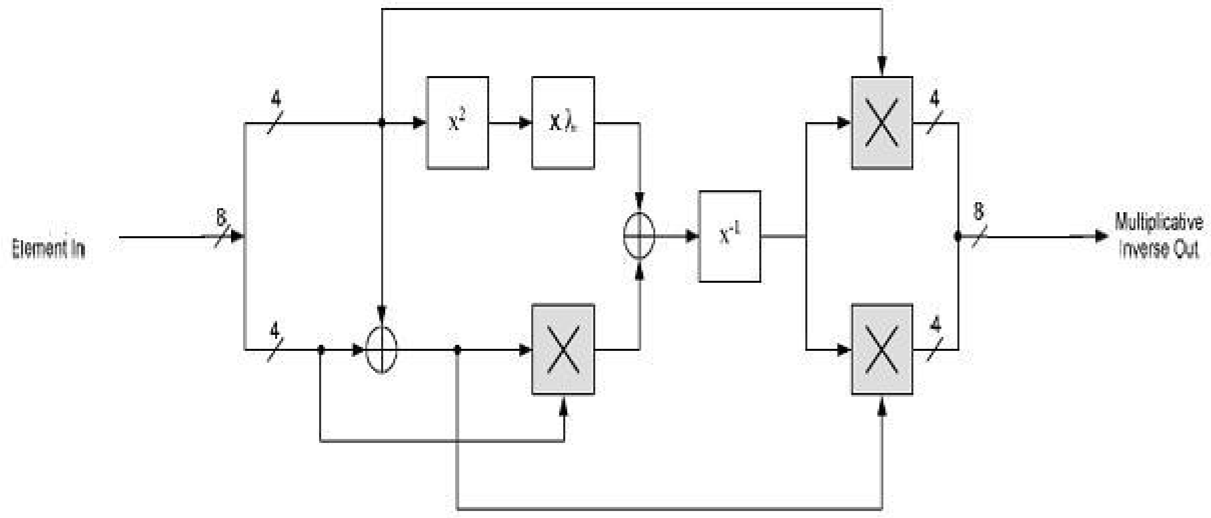
5.1.1 MULTIPLICATIVE INVERSION MODULE

This section illustrates the steps involved in constructing the multiplicative inverse module using composite field arithmetic. Both the SubByte and InvSubByte transformation are similar other than their operations which involve the Affine Transformation and its inverse. The individual bits in a byte representing a GF(2^8)element can be viewed as coefficients to each power term in the GF(2^8)polynomial. For instance, {10001011}2 is representing the polynomial $q^7 + q^3 + q + 1$ in GF (2^8). Any arbitrary polynomial can be represented as bx + c, given an irreducible polynomial of $x^2 + Ax + B$. Thus, an element in F1 after transformed to an element in F2 may be represented as bx + c where b is the most significant nibble while c is the least significant nibble. From here, the multiplicative inverse can be computed using the equation below.

$$(bx + c)^{-1} = b\,(b^2\lambda + c\,(b+c))^{-1}\,x + (c+b)\,(b^2\lambda + c\,(b+c))^{-1}$$

The Proof of this equation is given in appendix.

The above equation indicates that there are multiply, addition, squaring and multiplication inversion in GF (2^4)operations in Galois Field. Each of these operators can be transformed into individual blocks when constructing the circuit for computing the the multiplicative inverse. From this simplified equation, the multiplicative inverse circuit GF (2^4)can be produced as shown in Figure.



Multiplicative inversion module for the S-Box. [1]

x^{-1}	Multiplicative Inversion in GF(2^4)
✕	Multiplication operation in GF(2^4)
x^2	Squarer in GF(2^4)
$x\lambda$	Multiplication with constant, λ in GF(2^4)
⊕	Addition operation in GF(2^4)

Legends for the building blocks within the multiplicative inversion module.

5.1.1.1 ADDITION IN GF (2^4)

Addition of 2 elements in Galois Field can be translated to simple bitwise XOR operation between the 2 elements.

5.1.1.2 SQUARING IN GF (2^4)

Let C be the square of $A = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 + a_5\alpha^5 + a_6\alpha^6 + a_7\alpha^7 + a_8\alpha^8$.

Then C=A.A which can be computed by GF multiplication and repeatedly using the irreducible polynomial $y^4 + y + 1$. The result is

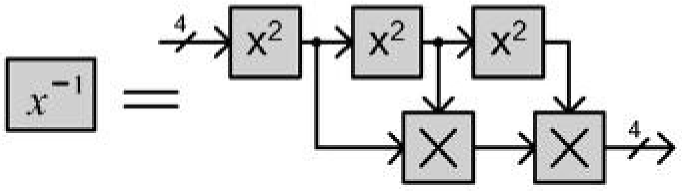
$$C_0 = a_0 + a_2$$
$$C_1 = a_2$$
$$C_2 = a_3 + a_1$$
$$C_3 = a_3$$

5.1.1.3 MULTIPLICATION IN GF (2^4):

Similar to squaring with only instead of A.A it is A.B

5.1.1.4 INVERSION IN GF (2^4)::

The multiplication is implemented as:



Thus the squaring consists of squaring and multiplication.

5.2 INV SHIFT ROW OPERATION:

In the InvShiftRows, the first row of the State does not change, while the rest of the rows are cyclically shifted to the right by the same offset as that in the ShiftRows. Hence this operation is independent of representation.

5.3 INV MIX COLUMN:

In matrix form, the InvMixColumns transformation can be expressed by

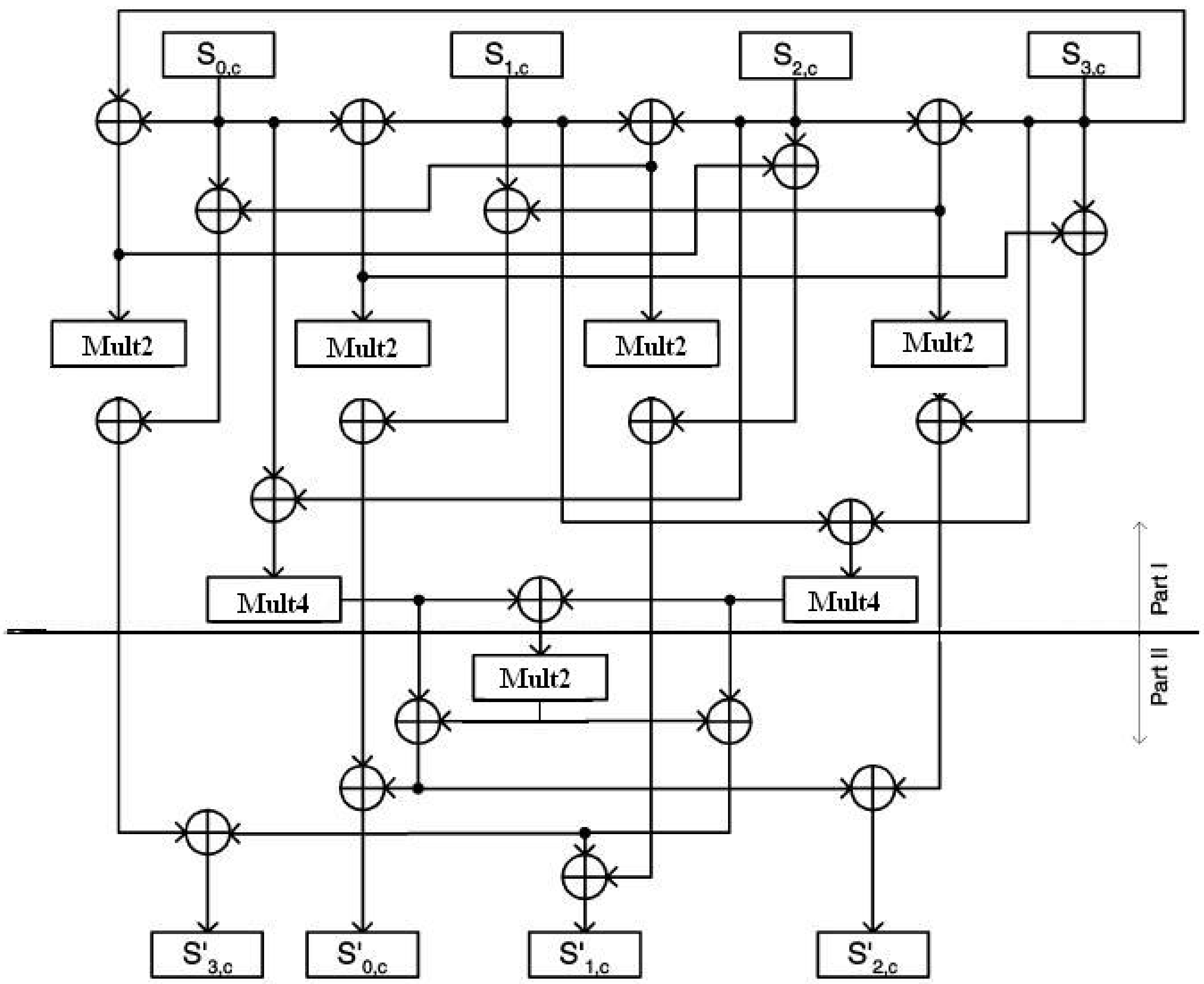
$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{0e\}_{16} & \{0b\}_{16} & \{0d\}_{16} & \{09\}_{16} \\ \{09\}_{16} & \{0e\}_{16} & \{0b\}_{16} & \{0d\}_{16} \\ \{0d\}_{16} & \{09\}_{16} & \{0e\}_{16} & \{0b\}_{16} \\ \{0b\}_{16} & \{0d\}_{16} & \{09\}_{16} & \{0e\}_{16} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

$$0 \le c < 4.$$

This can be rewritten as:

$$\begin{aligned}
S'_{0,c} = {} & (\{02\}_{16}(S_{0,c} + S_{1,c}) + (S_{2,c} + S_{3,c}) + S_{1,c}) \\
& +(\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\
& +\{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{0,c} + S_{2,c})) \\
S'_{1,c} = {} & (\{02\}_{16}(S_{1,c} + S_{2,c}) + (S_{3,c} + S_{0,c}) + S_{2,c}) \\
& +(\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\
& +\{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{1,c} + S_{3,c})) \\
S'_{2,c} = {} & (\{02\}_{16}(S_{2,c} + S_{3,c}) + (S_{0,c} + S_{1,c}) + S_{3,c}) \\
& +(\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\
& +\{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{0,c} + S_{2,c})) \\
S'_{3,c} = {} & (\{02\}_{16}(S_{3,c} + S_{0,c}) + (S_{1,c} + S_{2,c}) + S_{0,c}) \\
& +(\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\
& +\{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{1,c} + S_{3,c})).
\end{aligned}$$

Using substructure sharing, the InvMixColumn can be implemented by the architecture illustrated below. The "mult4" block computes the constant multiplication of {04} transformed by ø, can be implemented by two serially concatenated "mult2" block.



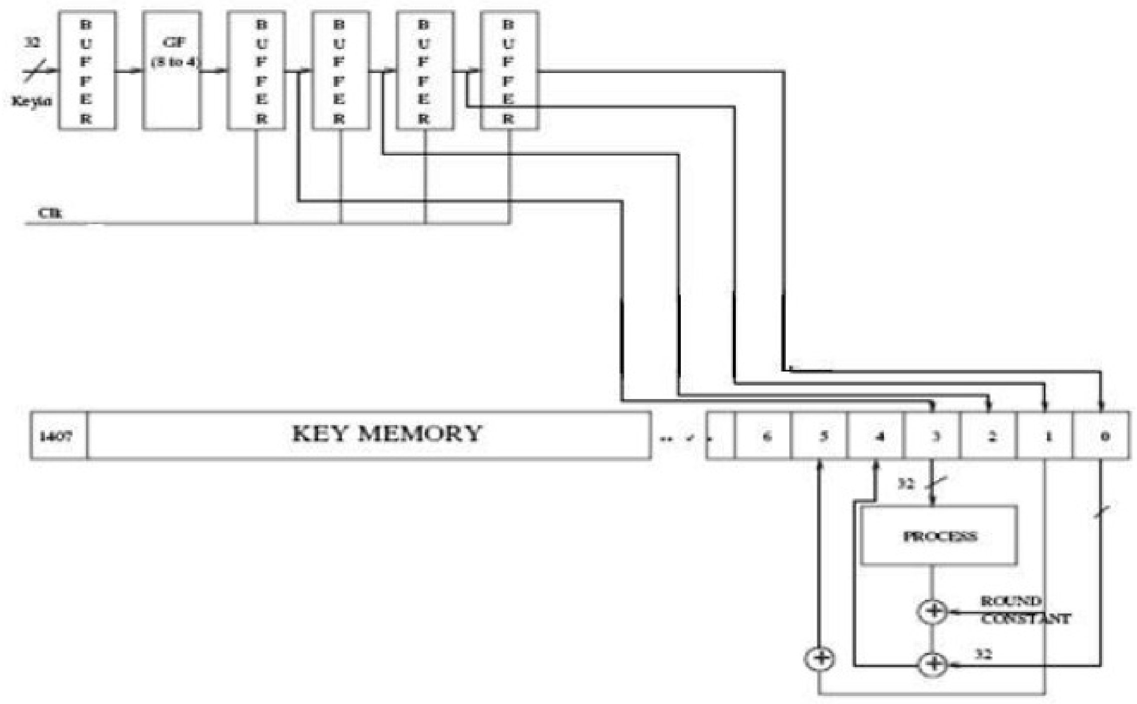
The upper half in the Fig.is exactly the same as the architecture for the implementation of the MixColumns. Therefore in a joint encryptor/decryptor implementation, only the architecture in the above Fig. needs to be implemented for both the MixColumns and the InvMixColumns transformations.

5.4 IMPLEMENTATION OF KEY EXPANSION:

Roundkeys can be either generated beforehand and stored in memory or generated on the fly. In the former approach, roundkeys can be read out from memory using appropriate addresses, and there is no extra delay for decryption. However, this approach is not suitable for the applications where the key changes constantly. Meanwhile, the delay of memory access is unbreakable, which may offset the speedup achieved by pipelining. Therefore it is more advantageous to generate roundkeys on the fly in a pipelined architecture.

5.5 KEY SCHEDULER:

The KeyScheduling unit has been multiplexed with the databus. The CRYPT signal when high sets the keygeneration mode whereby the input data is streamed into the scheduler and creates all the subkeys. The multiplexing successfully reduce the pin count as no extra pins are required for the input key. Further, when the AES core is used to process (decrypt) the data the keymemory is not accessible from the external world through the IO pins. Thus the round keys are secured from external attack during the normal operation of the device. The basic architecture of the Keyscheduling unit is shown in Figure below. The unit consists of a keygeneration module alongwith a keymemory.

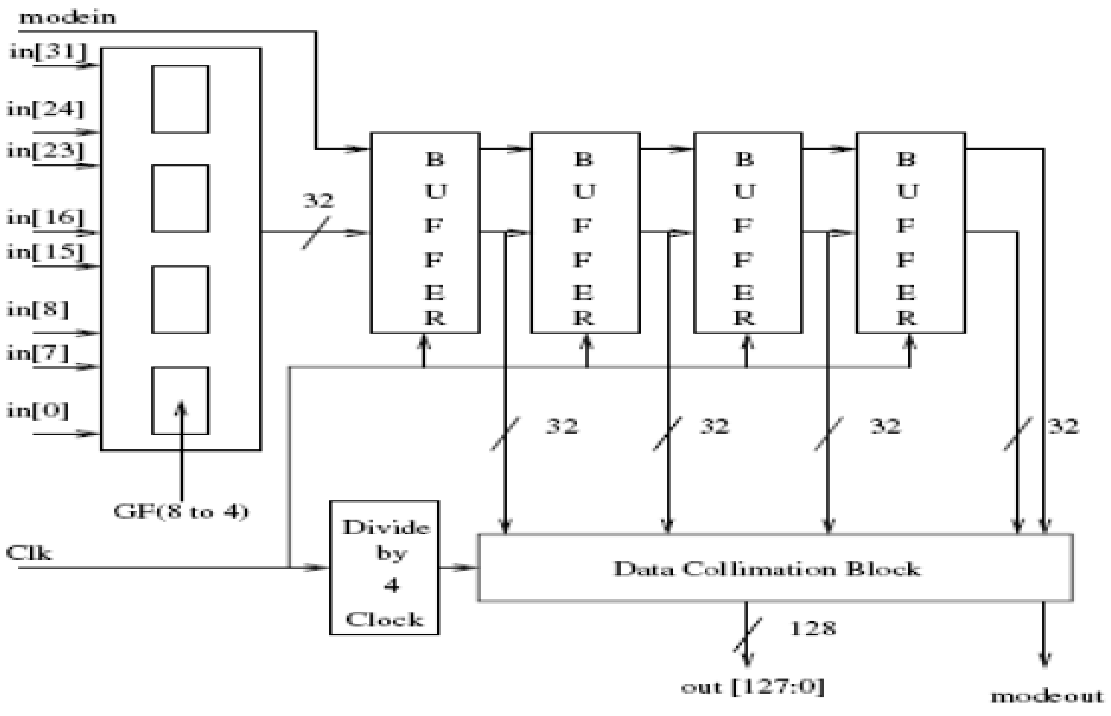


5.5 REDUCTION OF IO LINES:

The present design has a 32 bit I/O to aid in the interface with the conventional data buses of processors. The interfacing units and the control logics have been made inbuilt to the architecture. The important blocks are the Data Scheduler and the Data Dispatch Unit which performs the I/O interface between the data bus and the 128 bit decrypt blocks.

5.5.1 THE DATA SCHEDULER

The Data Scheduling Unit converts a GF (2^8) element into a GF $(2^4)^2$ element and buffers in the data at each system clock. At the fourth clock when 128 bit (32 x 4) of data has arrived the unit dumps the data into a 128-bit register . When a valid block is being decrypted, a corresponding signal of high modein is passed in. This signal is synchronized with the data block, so that a high modeout indicates that the result is corresponding to a valid input.



5.5.2 The DISPATCH UNIT

The Dispatch Unit converts the output word is to GF (2^8) and streams out as a 32-bit wide stream of processed data.

Chapter 6

RESULTS OBTAINED AND COMPARSIONS

The present design architecture is suited for 128 bit block cryptosystems. Inner and outer round pipelining has been used to obtain high throughputs. All computations are performed in composite field to reduce the complexity of the design. The key scheduling has been provided on chip without any deterioration in performance.

6.1 SYNTHESIS REPORT

The figure below shows the performance evaluation of the design implemented.

```
=========================================================================
Final Report
=========================================================================

Device utilization summary:
---------------------------
 Number of Slices:                 10155  out of  13696    74%
 Number of Slice Flip Flops:        3383  out of  27392    12%
 Number of 4 input LUTs:           19065  out of  27392    69%
 Number of IOs:                      69
```

6.2 THROUGHPUT COMPARISON:

Design	Device	Throughput	BlockRAMs	Slices
Our design	Virtex-E XCV1000E-7	13.5Gps	0	10155
Weaver's Rijndael	Virtex-E XCV600E-8	1.75 Gbps	10	770
GMU, Pipelined	Virtex-E XCV1000E-8	16.00 Gbps	80	9199
Amphion,	High Speed Virtex-E XCV50E-8	1.06 Gbps	10	573
Amphion,	Ultra High Speed Virtex-E XCV1600E-8	9.88 Gbps	100	2397
Helion, Fast	Virtex-E XCV400E-8	1.19 Gbps	10	450
Helion, Pipelined	Virtex-E XCV????E-8	>10 Gbps	?	?

6.3 FEATURE COMPARISON:

Features	Our Design	Weavers	GMU	Amphion-H	Amphion-U	Helion-F	Helion-P
Key Length 128, 192, 256	128	128	128,192, 256	128	128	128	128
Includes Key Expanion	Yes	Yes	No	Yes	Yes	Yes	Yes
I/O Bits 32, 128	32	128	128	32	128	128	128

6.4 PERFORMANCE COMPARISON

● **The** present design does not require any RAM compared to 80 block RAMs required by the GMU design team, [100].

● The current design has 32-bit key input as well as text data which can be directly interfaced with the 32-bit data bus of conventional processors. The interfacing module and control logics involved is hence inbuilt in our current design. It is evident that the benefit of a smaller number of I/O lines is obvious, since also smaller target devices with a limited number of input/output-pins can be used. As a disadvantage, decryption slows down considerably, [101]. One of the merits of the present design is in obtaining a high throughput with the fewer I/O lines.

● Key-scheduling is performed on-chip in the proposed design in contrast with the off-chip design reported in [102]. On-chip keyscheduling provides an end to end secured cryptosystem.

● The design results obtained in [102] show that the highest throughput is 16.8 Gbps for Serpent. But, an end to end implementation of the AES-candidates requires the removal of RAMs, the inclusion of on-chip keyscheduling and reduction in pin-count. With these additional design features, a throughput of 13.5Gps is notworthy

● **Reduction of Power:** An intelligent clocking strategy has been adopted. The main sys tem clock is divided by four to generate a slower clock which is used for the decryption blocks. The slower clock thus reduces the power consumption. The use of Galois Subfield reduces the area significantly and thus the power consumption is also reduced.

6.5 CONCLUSION:

In the present document, full outer pipelined 10-round AES-Rijndael have been designed and implemented using FPGA. The design includes on-chip keyscheduling and RAM free design, inspite of obtaining a high throughput of 13.5 Gbps. The performance of the design has been compared with competitive works and has been found to be the most efficient when power, throughput, area and other usablity features are concerned.

Appendix:

In the composite field GF $((2^4)^2)$, an element can be expressed as $s_h x + s_l$ € GF (2^4) and x is a root of $x^2 + x + \lambda$. Using Extended Euclidean algorithm, the multiplicative inverse of $s_h x + s_l$ modulo $x^2 + x + \lambda$ can be computed as

$$(s_h x + s_l)^{-1} = s_h \emptyset x + (s_h + s_l)\emptyset \qquad (7.1)$$

Where $\emptyset = (s_h^2 \lambda + s_h s_l + s_l^2)^{-1}$.

The problem of finding the inverse of $S(x) = (s_h x + s_l)$ modulo $P(x) = x^2 + x + \lambda$ is equivalent to finding polynomials $A(x)$ and $B(x)$ satisfying the following equation:

$$A(x)\,P(x) + B(x)\,S(x) = 1 \qquad (7.2)$$

Then $B(x)$ is the inverse of $S(x)$ modulo $P(x)$. Such $A(x)$ and $B(x)$ can be found by using the Extended Euclidean Algorithm for one iteration. First, we need to rewrite $P(x)$ in the form of

$$P(x) = Q(x)\,S(x) + R(x) \qquad (7.3)$$

Where $Q(x)$ and $R(x)$ are the quotient and remainder polynomials of dividing $P(x)$ by $S(x)$, respectively. By long division, it can be derived that

$$Q(x) = s_h^{-1} x + (1 + s_h^{-1} s_l)\, s_h^{-1}, \qquad (7.4)$$

$$R(x) = \lambda + (1 + s_h^{-1} s_l)\, s_h^{-1} s_l \qquad (7.5)$$

Substituting the equations (7.4) and (7.5) in (7.3) and multiplying s_h^2 to both sides gives

$$s_h^2\,\mathbf{P}(x) = (s_h x + (s_h + s_l))\,\mathbf{S}(\mathrm{x}) + (s_h^2\lambda + s_h\,s_l + s_l^2)$$

Multiplying $\emptyset = (s_h^2\lambda + s_h\,s_l + s_l^2)^{-1}$ to both sides of the equation we get

$$\emptyset\,s_h^2\,\mathbf{P}(x) = \emptyset\,(s_h x + (s_h + s_l))\,\mathbf{S}(\mathrm{x}) + 1 \qquad (7.6)$$

Since addition and subtraction are the same in the extended field of GF (2), the first term on the right side of 7.6 can be moved to the left side. Comparing (7.2) and (7.6), it can be observed that

$$\mathbf{S}^{-1}(x) = s_h\,\emptyset\,x + (s_h + s_l)\,\emptyset$$

REFERENCES:

[1] Altera. APEX II Programmable Logic Device Family Data Sheet.
www.altera.com/literature/ds/ds ap2.pdf.

[2] Amphion. www.amphion.com.

[3] P. Chodowiec, P. Khuon, and K. Gaj. Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining. *Proceedings of the ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays, Monterey, California, USA*, pages 94–102, February 11-13 2001.

[4] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag Berlin Heidelberg, 2002.

[5] A. Dandalis and V. K. Prasama. An Adaptive Cryptographic Engine for IPSec Architectures. *in Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000), Napa Valley,California, USA*, pages 132–131, 2000.

[6] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9:545–557,August 2001.

[7] FIPS. Advanced Encryption Standard (AES). *FIPS PUB 197*, November 26 2001. csrc.nist.gov/publications/fips/ ...fips197/fips-197.pdf.

[8] J. B. Fraleigh. *A First Course in Abstract Algebra*.Addison-Wesley Publishing Company, fourth edition, 1989.

[9] J. B. Fraleigh and R. A. Beauregard. *Linear Algebra*.Addison-Wesley Publishing Company, second edition, 1990.

[10] George Mason University. Hardware IP Cores of Advanced
Encryption Standard AES-Rijndael.
ece.gmu.edu/crypto/rijndael.htm.

[11] A. H¨am¨al¨ainen, M. Tommiska, and J. Skytt¨a. 6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm. *in Proceddings of the 12th Conference onField-Programmable Logic and Applications, FPL 2002, La Grande Motte, France*, pages 760–769, September 2002. Manfred Glesner, Peter Zipf and Michel Renovell (eds.).

[12] Helion Technology Limited. www.heliontech.com.

[13] H. Lipmaa. AES implementation speed comparison.
www.tcs.hut.fi/_helger/aes/rijndael.html.

[14] M. McLoone and J. V. McCanny. Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm. *in Proceedings of the 11th Conference on Field-Programmable Logic and Applications, FPL 2001,Belfast, Northern Ireland, UK*, pages 152–161, August 2001.Gordon Brebner and Roger Woods (eds.).

[15] V. Rijmen. Efficient Implementation of Rijndael S-box.
www.esat.kuleuven.ac.be/_rijmen/ ...rijndael/sbox.pdf.

[16] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[17] Virtex-E. Xilinx' Virtex-E Datasheet.
www.xilinx.com/partinfo/ds022.pdf.

[18] Virtex-II. Xilinx' Virtex-II Datasheet.
www.xilinx.com/partinfo/ds031.pdf.

[19] N. Weaver. Rijndael core.
www.cs.berkeley.edu/_nweaver/rijndael
[20]Kris Gaj and Pawel Chodowiec, “AES proposal: Rijndael, submitted as a candidate to the AES,” San Francisco, CA, April 8-12 2001, pp. 149–165, proceedings of RSA Security Conference - Cryptographer’s Track.
[21] C.S.K. Clapp, “Instruction level parallelism in AES candidates,”in *AES candidate conference*, Rome, March 22-23 1999, pp. 68–84.
[22] Charanjit S. Jutla etal. Atri Rudra, Pradeep K. Dubey, “Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic,” in *CHES*, CHES 2001:Paris, France, May 14-16 2001, pp. 171–184, Springer.
[23] National Institute of Standards and Technology (NIST), “Announcing the ADVANCED ENCRYPTION STANDARD (AES),” *Federal Information Processing Standards Publication*, vol. 32, no. 6, 2001.
[24] S.Morioka and A.Satoh, “An Optimized S-Box Circuit Architecture for Low Power AES Design,” in *CHES*, CHES 2002:, San Francisco Bay (Redwood City), USA, August 13-15 2002, Springer. [25] T. Rozylowicz B. Weeks, M.Bean and C. Ficke.,
“NSA’s final report on hardware evaluations,” in *Hardware performance simulations of round 2 Advanced Encryption Standard algorithms*,
http://csrc.nist.gov/encryption/aes/round2/r2anlsys.htm, May 15 2000.
[25] https://www.actapress.com/PaperInfo.aspx?PaperID=30901&reason=500
[26] http://ieeexplore.ieee.org/iel5/4339774/4339775/04340397.pdf
[27] http://delivery.acm.org/10.1145/780000/777450/p240-caltagirone.pdf?key1=777450&key2=8386600121&coll=GUIDE&dl=GUIDE&CFID=66742397&CFTOKEN=39164784
[28]http://delivery.acm.org/10.1145/1270000/1266608/p1116-alam.pdf?key1=1266608&key2=3596600121&coll=GUIDE&dl=GUIDE&CFID=66742591&CFTOKEN=65010994
[29] http://ieeexplore.ieee.org/iel5/10348/32912/01541355.pdf
[30] http://www.martes-itea.org/public/papers/Hamalainen-Design_and_Implementation_2.pdf.