University of Cincinnati

Date: 4/19/2012

I, Yogesh Joshi, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Hardware encryption of AES algorithm on Android platform

Student's name: Yogesh Joshi

This work and its defense approved by:

Committee chair: Philip Wilsey, PhD
Committee member: Fred Beyette, PhD
Committee member: Carla Purdy, PhD

2005
Hardware encryption of AES algorithm on Android platform

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the School of
Electric and Computing Systems
of the College of Engineering and Applied Sciences

April 19, 2012

by

Yogesh Joshi
BE ECE, National Institute of Technology, Durgapur, India

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey
Abstract

Mobile devices have revolutionized the world more than any other technology could have ever done. They have become an all purpose device performing numerous functions, thus eliminating the need of carrying multiple devices. In today’s world mobile phones are seen more as a necessity than a luxury. It has changed the way people communicate with one another, allowing almost everyone to speak to anyone else no matter where they are. In 1999, only 8 percent of the world population had mobile phone subscriptions. By 2007, this figure increased to 49 percent and today 77 percent of the world’s population has mobile phone subscription, out of which 10 percent are Smartphone users [1].

Smartphone is the category of mobile phones which has higher processing power and third party application support. In addition to the basic phone functions, Smartphone can be used to surf the web, listen to music, capture images, shoot videos and even play games. Gone are the days when it was necessary to carry cash and check for money transaction. In recent years this medium has been replaced by credit card, which will eventually be replaced by mobile transactions.

The market of Smartphone has evolved over a period of time and data security has now become a major concern. The use of Smartphone for online transactions has made it necessary to provide greater security to these handheld devices. As of now, secure data exchange on these devices is dependent on software implementation of cryptographic algorithms. To make the process robust, it becomes necessary to add this stage of security at hardware level itself.

This research proposes a space and power efficient hardware model of 128-bit Advanced Encryption Standard (AES), which can be distributed as secure digital (SD) card and incorporate plug and play capability. This implementation fits in low cost Spartan 3 XCS1500 Field Programmable Gate Array (FPGA). It operates at 1020 MHz or 710 MHz and produces a throughput of 1099 and 710.9 Mbps. At 1020MHz it
consumes 8.5nJ energy per operation cycle and at 710 MHz it consumes 10.5nJ of energy per cycle.
Acknowledgments

I would like to express my gratitude to my academic advisor, Dr. Philip A. Wilsey who has been a source of knowledge and guidance. His systematic approach to problem is one thing which helped me in my research work and will be with me throughout my life. Whenever I was bogged down with issues, he would ask me to take ‘baby steps’ and everything would start falling in place. I couldn’t have imagined having a better mentor and advisor for my Master’s work.

I would like to thank Dr. Fred Beyette and Dr. Carla Purdy for providing feedback for my thesis and taking time to serve in my Thesis Defense Committee.

I am really thankful to my friends Kuchi, Sandy, Vikash, Natwar and Rohit for making my stay in Cincinnati enjoyable, my friends back home Ashwini and Ujjwal for providing constant motivation and support and my sister Deepa for editing and proof-reading my thesis document.

Last but not the least, my greatest and special thanks to my parents for providing me love and support throughout my Master’s work and without whom I wouldn’t be where I am today.
# Contents

1 Introduction ........................................ 1
   1.1 The AES Encryption Standard ....................... 2
   1.2 Motivation for implementing AES on hardware ..... 3
   1.3 Contribution of this thesis ......................... 4
   1.4 Thesis Outline .................................... 5

2 Background ........................................... 6
   2.1 Notation and Conventions .......................... 6
      2.1.1 Inputs and Outputs ............................ 6
      2.1.2 The State .................................... 7
   2.2 Mathematical Background .......................... 7
      2.2.1 Finite Field .................................. 7
      2.2.2 Multiplicative Inverses ....................... 9
   2.3 Advanced Encryption Standard ..................... 9
      2.3.1 Sub Bytes/Inverse Sub Bytes .................. 11
      2.3.2 Shift Rows/Inverse Shift Rows ................. 13
      2.3.3 Mix Columns/Inverse Mix Columns .............. 15
      2.3.4 Add Round Key ................................ 17
      2.3.5 Key Schedule .................................. 17
   2.4 Characteristics of Hardware Implementation ...... 18
      2.4.1 Throughput .................................... 19
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2 Power consumption</td>
<td>19</td>
</tr>
<tr>
<td>2.4.3 Energy usage</td>
<td>20</td>
</tr>
<tr>
<td>2.4.4 Latency</td>
<td>21</td>
</tr>
<tr>
<td>2.4.5 Cost</td>
<td>21</td>
</tr>
<tr>
<td>2.5 Setting up Android build environment</td>
<td>21</td>
</tr>
<tr>
<td>3 Literature Review</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Implementation trade-off</td>
<td>23</td>
</tr>
<tr>
<td>3.2 High throughput AES implementation</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Low area AES implementation</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Low power AES implementation</td>
<td>26</td>
</tr>
<tr>
<td>4 Hardware Architecture</td>
<td>28</td>
</tr>
<tr>
<td>4.1 Key Expansion</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Shift Rows/Inverse Shift Rows</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Sub Bytes/Inverse Sub Bytes</td>
<td>32</td>
</tr>
<tr>
<td>4.4 Mix Column/Inverse Mix Columns</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Add Round Key</td>
<td>39</td>
</tr>
<tr>
<td>5 Encryption on Android</td>
<td>40</td>
</tr>
<tr>
<td>5.1 The Need for cryptographic encryption</td>
<td>40</td>
</tr>
<tr>
<td>5.2 Salient features of encryption</td>
<td>41</td>
</tr>
<tr>
<td>5.3 AES for hardware encryption</td>
<td>43</td>
</tr>
<tr>
<td>5.4 Android Architecture</td>
<td>44</td>
</tr>
<tr>
<td>5.5 Implementation of AES on Android</td>
<td>45</td>
</tr>
<tr>
<td>5.5.1 SD Card Emulation</td>
<td>46</td>
</tr>
<tr>
<td>5.5.2 Problems faced during AES implementation on Android</td>
<td>47</td>
</tr>
<tr>
<td>5.5.3 Native code library</td>
<td>48</td>
</tr>
<tr>
<td>5.5.4 Java Application</td>
<td>49</td>
</tr>
<tr>
<td>5.6 Results and Comparison</td>
<td>50</td>
</tr>
</tbody>
</table>
CONTENTS

6 Experimental Results 52
   6.1 Simulation of AES .................................................. 52
      6.1.1 AES Encryption .............................................. 52
      6.1.2 AES Decryption .............................................. 53
      6.1.3 AES Encryption/Decryption combined ..................... 54
   6.2 Performance and Efficiency ........................................ 55
   6.3 Design Parameters .................................................. 57
      6.3.1 AES Encryption and Decryption ............................. 57
      6.3.2 Key Sizes ..................................................... 58
      6.3.3 Key Schedule .................................................. 58
      6.3.4 S-box implementation ....................................... 58
      6.3.5 Power Analysis ............................................... 58
   6.4 Comparison .................................................................. 59

7 Conclusion and Future Work 62
   7.1 Conclusion .............................................................. 62
   7.2 Current adaptation .................................................... 63
   7.3 Future work ............................................................. 63
# List of Figures

2.1 AES operation on plain input text ................................................ 10
2.2 AES Encryption and Decryption module ......................................... 11
2.3 S-box substitution values ............................................................... 12
2.4 Inverse S-box substitution values .................................................... 14
2.5 Shift Rows Application on State array .......................................... 14
2.6 Inverse Shift Rows Application on State array ................................... 15
2.7 Mix Column operation on State array ............................................. 16
2.8 Add Round Key operation on State array ....................................... 18
2.9 Throughput diagram of System ...................................................... 19
2.10 Latency diagram of a System ....................................................... 21
3.1 AES Loop-unrolled architecture with pipelining ............................... 25
3.2 AES Loop-unrolled architecture with sub-pipelining ........................... 25
3.3 Looping architectures ................................................................. 26
4.1 Pseudo code for Key Expansion ..................................................... 30
4.2 Block diagram of 128-bit Key Expander Core .................................. 31
4.3 Input of Shift Rows operation ........................................................ 32
4.4 Block diagram for Substitution ....................................................... 33
4.5 Efficient Implementation of Mix Columns ....................................... 36
4.6 Efficient Implementation of Inverse Mix Columns ............................. 38
LIST OF FIGURES

5.1 Three categories of Cryptographic Algorithms ............................................ 43
5.2 Architecture of Android platform [2] .......................................................... 44
5.3 CryptoChip implementation in Android platform .......................................... 46

6.1 AES Encryption module ............................................................................. 53
6.2 AES Key Expander module ........................................................................ 53
6.3 AES Decryption module ............................................................................. 54
6.4 AES Encryption and Decryption module ...................................................... 54
6.5 Energy per operation vs frequency for AES Encryption ............................... 56
6.6 Energy per operation vs frequency for AES Decryption ............................... 57
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Comparison between Software, ASIC and FPGA Implementations</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>AES bit mode specifications</td>
<td>10</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of software and hardware based encryption</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Peak Data rates for different Telecommunication services</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Resource Usage and Performance of AES cipher</td>
<td>55</td>
</tr>
<tr>
<td>6.2</td>
<td>Power Efficiency of AES cipher at 50 MHz</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Power Efficiency of AES cipher at $F_{Max}$</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison AES Resource Usage and Performance</td>
<td>59</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison AES Energy Efficiency</td>
<td>60</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In general, data protection can be achieved by three basic steps: (i) controlling access to the system (via passwords), (ii) employing access control (through profiling), and (iii) restricting physical access. Today, mobile phones are used to store, modify and communicate sensitive information. These devices contain wireless capabilities and often transmit sensitive data wirelessly. Hence introducing new security concerns while transmitting data over this potentially insecure channel. Security protocols need to be employed at each layer of network stack to achieve secure wireless and wired communication.

SSL protocol is a universal standard used for client server authentication and encrypted messages transmission over the wireless. Before establishing a SSL secure connection, SSL enabled client and server confirms each others identity by verifying their certificate and public ID. This SSL protocol is divided into two sub-protocols namely: the SSL record protocol, which defines the format of transmission data and the SSL handshake protocol, which involves client server authentication, identification of common cryptographic algorithm supported by both client and server, generation of shared secret keys using public-key encryption and finally establishment of secured SSL connection.

Cryptography is the science of transforming data so that it is interpretable only by authorized persons. The choice of cryptographic algorithm depends on the security objective that needs to be achieved. Asymmetric and symmetric encryption algorithms are used for message authentication and privacy whereas hash or message digest algorithms are used to ensure message integrity. While security protocols and encryption algorithms provide answers from a functional perspective, meeting the power and processing requirement
on an embedded device becomes unrealistic. One of the major concerns is to bridge the gap between the power and processor requirements of these security protocols and the available battery and processor capabilities on the handheld devices. SSL uses RSA algorithm for symmetric key exchange, AES algorithm for secure data communication and SHA algorithm for message integrity check. As RSA is used only once for key exchange and, SHA1 has insignificant power usage, the bulk of the power consumption of the device is attributed to the AES encryption algorithm.

Hardware encryption is faster in comparison to software encryption because it has dedicated onboard security chip for performing encryption/decryption, key generation and key handling. The hardware encryption does not consume CPU cycles and hence does not affect system performance. In contrast, software encryption speed is dependent on the host computer and it will impact overall system performance. Furthermore, Software encryption also has the drawback that, when installed, the encryption keys are stored in device memory that can be accessed (and therefore potentially compromised) by other components of the mobile device.

1.1 The AES Encryption Standard

Data Encryption Standard (DES) algorithm was accepted as a federal standard in 1977 by the National Bureau of Standards (NIST). It is a symmetric key algorithm which encrypts and decrypts data in 64-bit blocks using a 56-bit key. As DES became vulnerable it was replaced by Triple DES, which was stronger as compared to DES but was slower in hardware. Cryptographers realized triple DES was not a satisfactory long-term solution. So, in 1997 the National Institute of Standards (NIST) put out call for candidates to replace DES entirely. The proposed algorithm should have the ability to allow key sizes of 128, 192 and 256 bit; should work on blocks of 128 bits; should be highly portable and should be equally fast in both hardware and software. From this call, the Rjandael algorithm was developed by John Daemen and Vincent Rijmen. It was approved by NIST as the new AES standard on October 2000 to replace the DES (FIPS 197, 2001) algorithm [3].
1.2 Motivation for implementing AES on hardware

Deploying security mechanisms for handheld devices consumes huge amount of power. While the processing power, memory and network bandwidth of today's mobile and wireless devices are increasing exponentially, their battery power is increasing at a modest pace. These devices can quickly drain their batteries and become useless. The main source of energy consumption during a secure wireless session depends on the frequency of transmission and reception of data packets, and the applied cryptographic algorithm. One method of reducing energy consumption is switching to smaller keys when security concerns are not stringent [4]. Another method is employing hardware acceleration of crypto-mechanisms [5]. FPGA implementation is advantageous as compared to software or ASIC implementation, on grounds of flexibility, scalability, cost-effectiveness and adaptability [6].

The implementation of cryptographic algorithm on reconfigurable platforms offers a host of advantages which include algorithm agility, algorithm upgradability and algorithm modification. Algorithm agility implies the ability of the algorithm to choose from a variety of ciphers on per session basis. This requirement comes from the fact that modern day security protocols are defined to be algorithm independent. Algorithm upgradability refers to the process of upgrading an existing algorithm in the application due to various reasons like expiration of standard and creation of new standard. Algorithm modification refers to changes in the algorithm implementation depending on the requirement and capability of the host devices. Although all these changes can also be accommodated in ASIC implementation but they come at high cost and lack runtime reconfiguration.

The quality of a hardware implementation is measured in terms of hardware cost, throughput, and la-
tency. The implemented hardware should be optimized for area or cost, should have a low latency which minimizes the time to encrypt single block of data, and should offer high throughput by encrypting multiple blocks in parallel.

Numerous FPGA implementations of AES have been presented and evaluated. Most of these implementations focus on high speeds, which are not cost effective and suitable only for high-end applications. A fully unrolled design provides a high throughput by unfolding all the ten rounds on the device. Another approach to this scheme is to implement a single round unit on hardware, which requires a block of data to go through cycles to perform encryption/decryption.

This thesis proposes a dedicated processor for performing hardware encryption on the handheld devices. The design provides a balance between the cost and throughput so as to allow the hardware model to be extended to low cost consumer products, such as PDAs. It provides comprehensive power and time analysis of most popular cryptographic algorithm used in SSL security protocol, the AES algorithm. The power analysis of the study is performed using Xilinx power analyzer on the design implemented on Spartan 3XCS1500 FPGA, which performs encryption/decryption of data stream of 16-bit block size.

1.3 Contribution of this thesis

The goal of this thesis is to research and evaluate hardware architecture design methodologies to propose a model having minimum power consumption and maximum efficiency and still be feasible for implementation on a small FPGA. A practical energy efficient FPGA based AES implementation is presented.

A partial loop unrolling of the AES implementation is proposed which require 10 rounds for AES encryption or decryption. This reduces the hardware cost to $1/10\text{th}$ as compared to fully unrolled hardware design. Encryption/decryption round keys are only generated once and stored in memory as opposed to generating keys on the fly. This result in one time power consumption cost for key generation process on the expense of additional memory required for storage. Also, this thesis states the steps for using the designed implementation model in the android development environment.
1.4 Thesis Outline

The organization of remaining of thesis is as follows: Chapter 2 gives the background and detailed overview of AES algorithm. Chapter 3 provides the literature review. Chapter 4 focuses on hardware architecture of the design. Chapter 5 describes the experimental setup for using this hardware implementation model on a handheld device. Chapter 6 gives a thorough discussion about the implementation results and comparison with other designs implemented on hardware. Chapter 7 provides conclusion of the design.
Chapter 2

Background

This thesis uses open-source logic core, AES (Rijndael) IP Core [7], of Rudolf Usselmann, published in OpenCores, as a starting point. The code is written in verilog and the design has tried to balance the implementation and trade off size and performance. This thesis shares the same design goal as the AES IP core, to be able to fit in a low cost FPGA and still be as fast as possible.

This chapter is intended to provide background information required for understanding of this thesis. It begins with Section 2.1 on Notation and Conventions that is followed by Section 2.2 which elaborates the mathematical background. Section 2.3 reviews the AES algorithm in detail. Finally, Section 2.4 concludes the chapter by stating the parameters considered for checking efficiency of hardware implementation in this thesis.

2.1 Notation and Conventions

2.1.1 Inputs and Outputs

AES algorithm takes 128-bit chunks of data as input and generates 128-bit of encrypted text. These chunks are also referred to as blocks and their size is referred to as their length. The encryption key is a sequence of 128, 192 or 256 bits and is also known as Cipher Key. The input and the output data is known, respectively, as plain-text and cipher-text.
2.1.2 The State

A State, in AES algorithm, is defined as a two dimensional array of bytes. It has four rows and \( N_b \) columns of bytes, where \( N_b \) is the block length divided by 32. The naming convention adopted for referencing individual bytes in the state array \( s \) is either \( s_{row, column} \) or \( s[\text{row}, \text{column}] \), where \( \text{row} \) in \([0, 4)\) and \( \text{column} \) in \([0, N_b)\). The AES standard block length is 128 bits, hence \( N_b = 4 \) and \( \text{column} \) in \([0, 4)\). The encryption process can be stated as follows:

1. The input array of bytes \( \text{input}_0, \text{input}_1, \ldots, \text{input}_{15} \), is copied into the state array \( s \) according to the scheme:

   \[
   s[\text{row}, \text{column}] = \text{input}[\text{row} + 4 * \text{column}] \text{ for } 0 \leq \text{row} < 4, \text{and } 0 \leq \text{column} < N_b. \tag{2.1}
   \]

2. Cipher operations are then performed on the state array \( s \).

3. The final value is copied to an output byte array, \( \text{output}_0, \text{output}_1, \ldots, \text{output}_{15} \) following the scheme:

   \[
   \text{output}[\text{row} + 4 * \text{column}] = s[\text{row}, \text{column}] \text{ for } 0 \leq \text{row} < 4, \text{and } 0 \leq \text{column} < N_b. \tag{2.2}
   \]

2.2 Mathematical Background

2.2.1 Finite Field

A field with limited number of elements is called a Finite field. All operations performed in the finite field results into an element within that field. The finite field with \( k \) elements is denoted \( GF(k) \) and is also called the Galois Field. The order of the finite field is the total number of elements in the field. Galois fields have numerous applications such as error detection and correction codes and cryptography.

Finite fields can be of following types:

- Prime Fields of the form \( GF(p) \), for some \( p > 0 \).
- Extension Fields of the form \( GF(p^k) \), for \( p > 0, k > 0 \).
• Composite Fields of the form $GF(p^n)^m$ where $GP(P^n)$ is the base field.

The key part of field definition is field polynomial which defines how the number space of the field is “wrapped around”, so that the result of any operation remains an element within the field. Reduction is the process of repeated subtraction of the field polynomial such that result of some operation will lie within the field.

The element notation of the finite field can be done in one of following forms:

**Binary notation:** Each element of $GF(2^8)$ is a polynomial of degree 7, where coefficient of each term of the polynomial can either take a value of 0 or 1. The least significant bit is used to represent the constant of the polynomial and the left most bit represents the most significant bit. Binary notation can be depicted as a concatenation of 8-bits, $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ where $a_7$ and $a_0$ represent the most and the least significant bit respectively.

**Polynomial notation:** Each data byte can be represented as a polynomial over the $GF(2^8)$.

$$a(\chi) = \sum_{a_i=0}^{7} a_i\chi^i = a_7\chi^7 + a_6\chi^6 + a_5\chi^5 + a_4\chi^4 + a_3\chi^3 + a_2\chi^2 + a_1\chi^1 + a_0 \quad (2.3)$$

**Hexadecimal notation:** Each data byte can be represented in hexadecimal format $AB$, where $A$ denotes $a_7a_6a_5a_4$ and $B$ denotes $a_3a_2a_1a_0$ in hexadecimal representation.

**Addition in finite field**

Addition in $GF(2^8)$ is obtained by adding the corresponding coefficients modulo 2. This can be accomplished by using eight XOR gates to add corresponding bits. Thus, if $s(\chi)$ denote addition of two finite field elements $a(\chi)$ and $b(\chi) \in GF(2^8)$, we have

$$a(\chi) = a_7\chi^7 + a_6\chi^6 + a_5\chi^5 + a_4\chi^4 + a_3\chi^3 + a_2\chi^2 + a_1\chi^1 + a_0 \quad (2.4)$$

$$b(\chi) = b_7\chi^7 + b_6\chi^6 + b_5\chi^5 + b_4\chi^4 + b_3\chi^3 + b_2\chi^2 + b_1\chi^1 + b_0 \quad (2.5)$$
CHAPTER 2. BACKGROUND  

2.3. ADVANCED ENCRYPTION STANDARD

\[ s(\chi) = a(\chi) + b(\chi) = \sum_{i=0}^{7} a_i \chi^i \oplus \sum_{i=0}^{7} b_i \chi^i = \sum_{i=0}^{7} (a_i \oplus b_i) \chi^i \]  

(2.6)

**Multiplication in Finite Field**

Multiplication of two polynomials in finite field is obtained by performing modulo with an irreducible polynomial to ensure that the final result stays within the used finite field. This can be achieved by first multiplying each term of the second polynomial with all of the terms of the first polynomial. These products are then added and if degree of the new polynomial is greater than 7, then it must be reduced modulo some irreducible polynomial. A polynomial is said to be irreducible if it cannot be factored into nontrivial polynomials over the same field. AES employs the following reducing polynomial for multiplication:

\[ m(\chi) = \chi^8 + \chi^4 + \chi^3 + \chi^1 + 1 \]  

(2.7)

Hence \( q(\chi) \), multiplication result of \( a(\chi) \) and \( b(\chi) \) in \( GF(2^8) \), is obtained by multiplying the two polynomials followed by modular reduction over \( m(\chi) \).

\[ q(\chi) = (a(\chi) \ast b(\chi)) \mod m(\chi) \]  

(2.8)

**2.2.2 Multiplicative Inverses**

In mathematics, a multiplicative inverse for a number, \( \chi \) denoted by \( \frac{1}{\chi} \) or \( \chi^{-1} \), is a number which when multiplied by the number yields the multiplicative identity. The multiplicative inverse for any non-zero binary polynomial \( a(\chi) \) of degree less than the irreducible polynomial in a field is denoted as \( a^{-1}(\chi) \). The extended Euclidean algorithm and the Fermat’s little theorem are two common ways to find the multiplicative inverse in a finite field.

**2.3 Advanced Encryption Standard**

The Advanced Encryption Standard (AES) is a symmetric-key encryption standard adopted by the US government. This standard comprises of three block ciphers, AES-128, AES-192 and AES-256. These ciphers
have 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively. The key length is represented by $N_k = 4, 6, \text{ or } 8$, which reflects the number of 32-bit words (number of columns) in the Cipher Key. In an AES algorithm execution $N_r$ rounds of transformations are performed, where the value of $N_r$ depends on the key size. Table 2.1 below shows the relation between key length, block size and number of rounds.

The AES implementation can be divided in two main components: the Cipher and the Key Schedule. The Cipher is responsible for performing encryption or decryption of the input data blocks and the Key schedule is responsible for generating the cipher keys for each round. In order to transform input data into cipher text or vice-versa, AES-128 requires performing 10 rounds of substitution or permutation operation for both encryption and decryption.

During encryption process, the state array goes through SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations. Similarly, during decryption process the state array goes through InvSubBytes, InvShiftRows, InvMixColumns and AddRoundKey transformations. AddRoundKey requires a separate set of 128-bit keys, based on the input cipher key, for each round. A diagrammatic representation of AES operation is shown in the Figure 2.1.

From the Figure 2.2, we can see that the decryption process is reverse of the encryption process. It uses the same set of cipher keys, but in opposite order, and uses the inverse of SubBytes, ShiftRows, and MixColumn for its transformation.
2.3. ADVANCED ENCRYPTION STANDARD

2.3.1 Sub Bytes/Inverse Sub Bytes

In the Sub Bytes step, each byte in the state array is substituted with one of the bytes from the substitution table (S-box). This step introduces non-linearity in the cipher. S-box generation consists of two functions multiplicative inverse in $GF(2^8)$ and application of affine transformation. The affine transformation is given by the equation:

$$AT(x_i) = x_i \oplus x_{(i+4)\mod 2} \oplus x_{(i+5)\mod 2} \oplus x_{(i+6)\mod 2} \oplus x_{(i+7)\mod 2} \oplus c_i$$

(2.9)

for $0 \leq i < 8$, where $b_i$ and $c_i$ is the $i^{th}$ bit of byte $b$ and $c$, respectively and the byte $c$ has a value $\{63\}$ in decimal or $\{01100011\}$ in binary format.

Figure 2.2: AES Encryption and Decryption module
The affine transformation of the S-box, in matrix form, can be written as:

\[
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
    b_4 \\
    b_5 \\
    b_6 \\
    b_7
\end{bmatrix} = \begin{bmatrix}
    1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
    1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
    1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
    1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
    1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
    0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
    0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
    b_0 \\
    b_1 \\
    b_2 \\
    b_3 \\
    b_4 \\
    b_5 \\
    b_6 \\
    b_7
\end{bmatrix} + \begin{bmatrix}
    0 \\
    0 \\
    0 \\
    0 \\
    0 \\
    0 \\
    0 \\
\end{bmatrix}
\]  

The S-box is a fixed size $16 \times 16$ matrix having 256 bytes of value in range $00_H - FF_H$ and can either be pre-calculated and stored in memory as LUT (Look up table), or implemented manually by calculating multiplicative inverse and performing affine transformation on the input bytes. Both design strategies are discussed in Chapter 3. Figure 2.3 shows the S-box representation where $x$ represent the most significant nibble and $y$ represents the least significant nibble.

The S-box was created in a way so that it is invertible to allow its use as Inverse S-Box. This is possible because all the combinations are unique and each input byte matches to exactly one output byte. The process
of formation of Inverse S-Box is the reverse of construction of S-Box, which is inverse of the affine transform followed by multiplicative inversion in $GF(2^8)$. The inverse affine transformation, in equation and matrix form, can be found in (2.11) and (2.12) respectively.

\[
AT(x_i) = x_{(i+2)\mod 8} \oplus x_{(i+5)\mod 8} \oplus x_{(i+7)\mod 8} \oplus d_i \tag{2.11}
\]

for $0 \leq i < 8$, where $x_i$ is the $i^{th}$ bit of byte $x$, and $d_i$ is the $i^{th}$ bit of a byte $d$ with the value $\{05\}$ or $\{00000101\}$.

\[
\begin{bmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  b_3 \\
  b_4 \\
  b_5 \\
  b_6 \\
  b_7 \\
\end{bmatrix}
= \begin{bmatrix}
  1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
  0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
  1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
  0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
  0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
  1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
  0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
  0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  b_3 \\
  b_4 \\
  b_5 \\
  b_6 \\
  b_7 \\
\end{bmatrix}
+ \begin{bmatrix}
  1 \\
  0 \\
  1 \\
  0 \\
  0 \\
  0 \\
  0 \\
  0 \\
\end{bmatrix}
\tag{2.12}
\]

Figure 2.4 shows the Inverse S-box representation where $x$ represent the most significant nibble and $y$ represents the least significant nibble.

### 2.3.2 Shift Rows/Inverse Shift Rows

The *Shift Rows* step operates on rows of the state; it cyclically shifts the bytes of each row with a certain offset. This offset is dependent on the row number. The first, second, third, and fourth rows are shifted to left by zero, one, two, and three bytes respectively. Figure 2.5 shows the Shift Rows operation.

The *Inverse Shift Rows* function is used in the decryption process of the cipher. Inverse Shift Rows is exactly the same as Shift Rows, only in the opposite direction. The first, second, third, and fourth rows are shifted to right by zero, one, two, and three bytes respectively. Figure 2.6 shows the Inverse Shift Row operation.
CHAPTER 2. BACKGROUND

2.3. ADVANCED ENCRYPTION STANDARD

Figure 2.4: Inverse S-box substitution values

Figure 2.5: Shift Rows Application on State array
2.3.3 Mix Columns/Inverse Mix Columns

The Mix Columns step involves multiplication of a $4 \times 4$ matrix of $GF(2^8)$ values by a column vector of $GF(2^8)$ values. Each column is treated as a 4-byte polynomial over $GF(2^8)$, as shown in Figure 2.4, and then multiplied modulo $\chi^4 + 1$ with the fixed polynomial $a(\chi)$, given by

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

(2.13)

Equation (2.14) shows the matrix multiplication of fixed polynomial $a(\chi)$ with state array $Sin(\chi)$

$$\begin{pmatrix}
Sout'_0,c \\
Sout'_1,c \\
Sout'_2,c \\
Sout'_3,c
\end{pmatrix} =
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{pmatrix}
Sin_0,c \\
Sin_1,c \\
Sin_2,c \\
Sin_3,c
\end{pmatrix}$$

(2.14)

for $0 \leq c < N_b$ where 32-bit input word $Sin = Sin_0,c Sin_1,c Sin_2,c Sin_3,c$ and 32-bit word $Sout' = Sout'_0,c Sout'_1,c Sout'_2,c Sout'_3,c$. Hence we have:
CHAPTER 2. BACKGROUND  2.3. ADVANCED ENCRYPTION STANDARD

Figure 2.7: Mix Column operation on State array

\[
\begin{align*}
S'_{0,0} &= (\{02\} \cdot Sin_{0,c}) \oplus (\{03\} \cdot Sin_{1,c}) \oplus Sin_{2,c} \oplus Sin_{3,c} \\
S'_{1,0} &= Sin_{0,c} \oplus (\{02\} \cdot Sin_{1,c}) \oplus (\{03\} \cdot Sin_{2,c}) \oplus Sin_{3,c} \\
S'_{2,0} &= Sin_{0,c} \oplus Sin_{1,c} \oplus (\{02\} \cdot Sin_{2,c}) \oplus (\{03\} \cdot Sin_{3,c}) \\
S'_{3,0} &= (\{03\} \cdot Sin_{0,c}) \oplus Sin_{1,c} \oplus Sin_{2,c} \oplus (\{02\} \cdot Sin_{3,c})
\end{align*}
\]

where \(\oplus\) is the XOR operation and \(\cdot\) is the multiplication modulo the irreducible polynomial \(m(\chi)\).

The Inverse Mix Columns function is used in the decryption process. It is same as the Mix Columns, except that it uses the inverse of Mix Column transformation matrix in its matrix multiplication. In Mix Columns step, each column is treated as a 4-byte polynomial over and then multiplied modulo \(\chi^4 + 1\) with the fixed \(a^{-1}(\chi)\) polynomial, given by:

\[
a^{-1}(\chi) = \{0B\}\chi^3 + \{0D\}\chi^2 + \{09\}\chi + \{0E\}
\]

Equation (2.17) shows the matrix multiplication of fixed polynomial \(a^{-1}(\chi)\) with state array \(Sin(\chi)\).
\[ S\text{out}'(\chi) = a^{-1}(\chi) \oplus Sin(\chi) \]

\[
\begin{bmatrix}
S\text{out}'_{0,c} \\
S\text{out}'_{1,c} \\
S\text{out}'_{2,c} \\
S\text{out}'_{3,c}
\end{bmatrix} =
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E
\end{bmatrix}
\begin{bmatrix}
Sin_{0,c} \\
Sin_{1,c} \\
Sin_{2,c} \\
Sin_{3,c}
\end{bmatrix}
\]

(2.17)

where 32-bit input word \( Sin = Sin_{0,c} Sin_{1,c} Sin_{2,c} Sin_{3,c} \) and 32-bit output word \( S\text{out}' = S\text{out}'_{0,c} S\text{out}'_{1,c} S\text{out}'_{2,c} S\text{out}'_{3,c} \).

Hence we have:

\[
S\text{out}'_{0,c} = \{0E\} \cdot Sin_{0,c} \oplus \{0B\} \cdot Sin_{1,c} \oplus \{0D\} \cdot Sin_{2,c} \oplus \{09\} \cdot Sin_{3,c}
\]

\[
S\text{out}'_{1,c} = \{09\} \cdot Sin_{0,c} \oplus \{0E\} \cdot Sin_{1,c} \oplus \{0B\} \cdot Sin_{2,c} \oplus \{0D\} \cdot Sin_{3,c}
\]

\[
S\text{out}'_{2,c} = \{0D\} \cdot Sin_{0,c} \oplus \{09\} \cdot Sin_{1,c} \oplus \{0E\} \cdot Sin_{2,c} \oplus \{0B\} \cdot Sin_{3,c}
\]

\[
S\text{out}'_{3,c} = \{0B\} \cdot Sin_{0,c} \oplus \{0D\} \cdot Sin_{1,c} \oplus \{09\} \cdot Sin_{2,c} \oplus \{0E\} \cdot Sin_{3,c}
\]

(2.18)

2.3.4 Add Round Key

The Add Round Key function is used to mix key information with the data. In this step, the Sub Key is combined with the state array. Key schedule derives sub key of size equal to that of state from the main key. Then a bitwise XOR operation is performed between current state and the current round key. This function is used in both encryption and decryption of the cipher.

2.3.5 Key Schedule

Key expansion is used to generate the round keys for each round based on the original input key. The initial round uses the original input key and for the remaining rounds the keys produced by the key expansion are used. The key expansion produces a four byte word at a time, based on the previous word and the word one key length back.
AES goes through $N_r$ round Key steps and one initial round Key step; hence it requires $(N_r + 1)$ unique round keys. The key expansion generates a total of $N_b \cdot (N_r + 1)$ words. For AES-128, $N_b = 4$ and $N_r = 10$; thus key expansion generates 44 words. Each word is indexed by $i$, where $0 \leq c < 44$, and $i \in [0, 3]$ corresponds to the original input key. The process to be followed for generation of these words depends on the index $i$. If $i$ is not a multiple of 4 then the resultant word is produced by a XOR operation between the previous word and the corresponding word from the previous round key, otherwise the resultant word is produced by application of RotWord, SubWord, and AddRcon functions. RotWord is similar to Shift Row function, it takes a word $[a_0, a_1, a_2, a_3]$ as input, performs a cyclic permutation, and returns $[a_1, a_2, a_3, a_0]$ the word. Similarly SubWord is identical to Sub Bytes function, it takes an input word and applies S-box transformation to each of the four bytes to produce an output word. Finally, the AddRcon function performs a XOR operation between the word and the round constant.

Key expansion uses a round constant to break symmetry and introduce non-linearity in the data.

2.4 Characteristics of Hardware Implementation

The parameters for measuring efficiency of a hardware implementation are execution speed, and cost. Cryptographic algorithms act on a block of data to transform plain input text to cipher text. Hence, the speed is estimated by throughput, energy efficiency is calculated using static and dynamic consumption, accompanied by another parameter-latency, and cost depends on the implementation area.
2.4. CHARACTERISTICS OF HARDWARE IMPLEMENTATION

2.4.1 Throughput

Throughput indicates data processing capacity. It is given by the amount of data that can be processed and transferred within a unit time and it is usually expressed in Mbps or Gbps. In symmetric-key algorithms, typically the encryption and decryption throughput are equal because the decryption process has to go through the same series of steps as encryption but in reverse order. Hence, the number of rounds a plaintext data has to go through before cipher-text is generated is fixed and

\[
\text{Throughput} = \frac{\text{Number of processed bits}}{\text{Time elapsed}}.
\]

If we consider a system comprising of multiple modules with variable data processing capabilities, the maximum throughput of the system is limited by the maximum throughput of the slowest module.

\[
\text{Throughput}_{\text{system}} = \sum_{i=0}^{n} TP_i
\]  

(2.19)

where \( TP_i \) is the throughput of sub-system \( i \).

The throughput, \( TP_{op} \), of an operation cycle for an AES design with \( N_{\text{bits}_{op}} \) number of output bits per operation and operating at a frequency \( f_{op} \) is given as:

\[
TP_{op} = N_{\text{bits}_{op}} \times f_{op}
\]  

(2.20)

2.4.2 Power consumption

Power consumption is one of the major concerns in the mobile and handheld devices. It drains the battery of the device resulting in some designs to become unfeasible.

Power usage, in FPGAs, can be broken down into static and dynamic components, where the total power usage is the sum of the two:

\[
P_{\text{total}} = P_{\text{Static}} + P_{\text{Dynamic}}
\]  

(2.21)
Essentially static power is the power used when the transistor is not in the process of switching. Static power dissipation is the power used to hold a circuit in one logic state or the other. It is caused due to current leakage, which occurs all times when the device is on. Static power is the product of supply voltage and leakage current.

\[ P_{Static} = V_{Supply} \times I_{Leakage} \]  

(2.22)

In contrast, dynamic power is the power dissipated due to transition of signals. It is caused due to charging and discharging of capacitors. This power is proportional to capacitance, square of supply voltage and frequency of operation.

\[ P_{Dynamic} = C_{node} \times V_{Supply}^2 \times f \]  

(2.23)

### 2.4.3 Energy usage

Energy usage is the average energy consumption which is directly proportional to the average power consumption. Energy consumption, \( E \), of a circuit dissipating power at an average rate of \( P \) and operating for \( t \) seconds is given as:

\[ E = P \times t \]  

(2.24)

However, for our design analysis, the term energy consumption per operation cycle is more meaningful. Once the power consumption of the design is determined, the energy usage per operation, \( E_{PerOp} \), of the design dissipating average power \( P_{op} \) and requiring time \( t_{op} \) for completion of one operation can be calculated as:

\[ E_{PerOp} = P_{op} \times t_{op} \]  

(2.25)

Using Equation (2.20) and (2.25), Energy usage, \( E_{PerOp} \), in terms of throughput of the operation, \( TP_{op} \), can be written as:
2.4.4 Latency

Latency is a measure of time delay experienced in a system between a stimulus and response. In cryptography, latency is defined as the total time required for processing one block of data, which is generally measured in clock cycles. Latency and throughput analyze different features of the system. If we consider a system comprising of multiple modules with variable data processing capabilities, the combined latency of the system, contrary to throughput, is equal to summation of latencies of each individual module.

\[ E_{PerOp} = P_{op} \times \frac{N_{bits_{op}}}{TP_{op}} \]  

(2.26)

where \( Latency_i \) is the latency of module i.

2.4.5 Cost

In an FPGA, the cost of a design is directly proportional to the total area of the implementation. This design implementation area is usually reported in terms of number of configurable logic blocks and block RAMs used. Rather than reporting the number of CLB slices used, some FPGA manufactures report equivalent FPGA circuitry. Another means of reporting efficiency of the design is by mentioning throughput of the design per unit slice. This thesis reports the design area in all three formats.

2.5 Setting up Android build environment

This research uses Googles Android 2.2 operating system, also known as Froyo, as development framework. Among the other mobile operating systems, such as Apples iOS and Blackberry OS, Android OS is an
obvious choice because it is open source. Android source is approximately 2.6 GB and requires 6 GB free
space to complete the build. Android source code build is only supported in Linux and Mac OS. This work
uses Linux OS as building environment. The steps to set-up local work environment, get android source
code and build files on the machine can be found Android open source project website [2].
Chapter 3

Literature Review

This chapter presents some of the previous work done in the area of FPGA based AES implementation. A key size of 128 bits is considered sufficient for most commercial applications. Keys with higher bit size are considered a waste of resources because they require greater implementation area and have longer processing time. Section 3.1 of this chapter focuses on discussion of implementation trade-offs. The work done in the field of AES hardware implementation can be divided into three broad categories: high throughput, low area, and low power. These are each described separately in Sections 3.2, 3.3 and 3.4.

3.1 Implementation trade-off

There are two kinds of implementation trade-offs that can be applied to FPGA based implementation of the design, namely: algorithm optimization and architectural optimization. Algorithm optimization exploits algorithm design trade-offs that can be applied in each round and architecture optimization exploits tradeoffs in design techniques.

Algorithm optimization will be discussed in detail in Chapter 4. Below are some of the design techniques that can be utilized for architecture optimization [8].

1. **Loop architecture**: This architecture employs small number of rounds that are independently implemented in hardware. It results in a reduction of implementation area and hardware cost.

2. **Unrolled architecture**: In this approach, each individual round is implemented independently in the
3. **Pipelining:** This is a basic design strategy used to increase operating frequency and throughput of the design. Multiple blocks of data are processed simultaneously and each round unit constitutes a pipeline stage. Glitches propagating in combination logic and routing resource results in considerable increases in the power consumption. Pipelining the design helps in breaking of large runs of combinational logic; hence dampening in propagation of glitches, directly resulting in power savings.

4. **Inner-Pipelining:** This involves the insertion of registers inside the round unit so that several blocks of data can be processed by the cipher at the same time. As, this breaks down the critical path of the design, the circuit is capable of performing in higher clock frequencies and results in greater throughput.

### 3.2 High throughput AES implementation

Since the introduction of AES many high throughput FPGA based implementation have been developed [9] [10]. To increase throughput these designs employ full unrolling of the iterative structure of AES and introduce pipelining within each round. The throughput can be ideally increased N times by replicating N stages of implementation.

In Loop unrolling, rather than allowing the data to pass repeatedly through a same block of hardware, a separate hardware for each AES round is implemented. The more round units the architecture implements, the higher is the hardware cost. This structure allows new data value to be accepted with each clock cycle as the processed data from the previous round is passed onto the next hardware round.

This approach decreases the number of clock cycles required for encryption, but it results in extremely slow system clock as it maximizes the worst case resister-to-register delay. To fully benefit from a loop unrolled architecture, the pipeline stages need to be decomposed so that each stage of the structure outputs data after every clock cycle. Figure 3.1 shows a loop-unrolled architecture with pipelining where $N = 10$ for 128 bit key size.

Adding pipeline stage increases the throughput by allowing higher clock frequencies. This higher encryption throughput is achieved at higher hardware cost. In such design, the S-box implementation is
achieved using a composite field approach rather than block RAMs (as the latter inserts large delays in the data path). These designs expand keys on the fly and discard them when they are no longer required in order to avoid the storage of the large amounts of key data that is generated during encryption and decryption along the pipeline stages. As large amount of key data is generated and discarded at each pipeline stage on every clock cycle, this increases the power consumption of the structure.

Some designs employ more advanced pipelining technique to increase the throughput by introducing sub-pipelining. In these designs, instead of introduction of pipeline stage at the end of an AES round, the AES round is divided into a certain number of pipeline stages. This approach increases the throughput by the factor of sub-pipelining.

The above design is used in applications free of area constraints and requiring tens of gigabytes of throughput. Such solutions would not be acceptable for use in mobile devices with tight cost and power budgets.

### 3.3 Low area AES implementation

There are many FPGA based AES implementation, focused on using very little area [11] [12]. Such implementations either have 8-bit or 32-bit data path; 8-bit data path aim for low area architectures and 32-bit data path target medium throughput applications. To achieve low area usage, these designs make use of embedded functional blocks and data path width reduction strategies. These designs also make use of looping architectures. Figure 3.3 shows a looping architecture where the number of iterations through the hardware depends on the width of data path.
In AES looping architecture, one round of the cipher is implemented in hardware and the data has to loop Nr times for successful completion of AES encryption/decryption process. This implementation has low register to register delay on the expense of large number of clock cycles required for encryption. This basic round includes the hardware implementation of four AES steps: Shift rows, Byte substitution, Mix columns, and Add round key. This approach reduces the FPGA implementation area usage as the single round unit implementation costs approximately $1/N_r$ area in comparison to area usage in unfolded scheme. This reduction in hardware cost is achieved by sacrificing encryption/decryption speed.

### 3.4 Low power AES implementation

Due to high power consumption in FPGAs, there are few architecture designs targeting low power or energy efficient AES implementation in FPGAs. Most of the designs stating to be low area and energy efficient, do not have their power consumption data available to allow any sort of direct comparison. Combinations of various low power consumption techniques have been used by various researches to provide energy efficient FPGA based AES implementation.

Katashita et al [13] proposed a low power design method by using extra flip-flops driven by phase-shifted clock. The technique was efficiently used in the design to provide power savings. Two different models were used to implement the AES S-boxes; the first one used an asynchronous block RAM and the second one used composite field for performing multiplicative inversion in $GF((2^4)^2)$. Both the proposed models implemented the hardware for single round and cycled the data through it 11 times. The extra flip-flops driven by phase-shifted clock are introduced to be replacements of pipeline registers. It does not increase the cycle latency of the original circuit. In contrast to the pipeline method, it does not require scheduling and the number of extra flip flops are smaller than in pipeline method. This design is capable of
CHAPTER 3. LITERATURE REVIEW 3.4. LOW POWER AES IMPLEMENTATION

performing both encryption and decryption, but uses static pre-expanded key stored in block memory. This design was implemented in Xilinx Virtex II and Xilinx XPower tool was used for power analysis. Using the above power reduction strategies, the block RAM design achieved dynamic power consumption of 309.255 mW and 332.46 mW for encryption and decryption respectively when run at a clock frequency of 66.66 MHz, while the LUT based design achieved dynamic power consumption of 306.69 mW and 319.17 mW for encryption and decryption respectively when run at a clock frequency of 50 MHz.

Rejeb et al [14] proposed low data path width FPGA designs for achieving low power consumptions. The design used several techniques for achieving the objective. First, to reduce the complexity of S-box transformation the elements from $GF(2^8)$ were mapped to elements in $GF((2^4)^2)$ then suitable methods for performing multiplicative inversion were explored. Second, it used low data path width so as to reduce the number of dedicated components working simultaneously. Third, the structure had the capability of disabling the components until the data was available for computation. The designs implemented with one, two, and four S-boxes in parallel have a dynamic power consumption of 137 mW, 153 mW and 169 mW when performing at 40 Mbps, 92.5 Mbps and 142 Mbps respectively. It is capable of performing key expansion, encryption and decryption, and power estimates are performed using Xilinx Power Estimator Spreadsheet.

Kenney [15] proposed an energy efficient design by using dynamic power reduction technique. Apart from pipelining, the use of embedded block RAM and register clock enable signals; the structure takes the advantage of unused Look Up Tables inputs to reduce signal activity. The unused portions of the data path are gated off to conserve energy and therefore reduce power consumption. The design is implemented separately for basic, pipelined, operand gating, basic memory and enhanced memory logics which have a total power consumption of 508.92 mW, 158.52 mW, 144.79 mW, 128.32 mW, 117.06 mW and, 516.54 mW, 158.74 mW, 150.14 mW, 128.21 mW, 119.85 mW, respectively when operating at a frequency of 50 MHz. This design was implemented in Altera Cyclone II EP2C35 and the power consumption was evaluated using AES Energy Analyzer, which uses the filter, simulator and PowerPlay Power Analyzer from Altera Quartus II development software.
Chapter 4

Hardware architecture of the design

Most of the low end embedded applications such as mobile phones and portable devices do not require high throughput and do not have the infrastructure to support excessive area required by 128-bit fully unrolled systems. Therefore, we will be concentrating our work on efficient 32-bit AES design for 128-bit key size. As discussed earlier, AES is categorized according to the data path width. Designs targeting very high speed utilize 128-bit fully unrolled pipelined architectures whereas those targeting very low area use 8 bit data path implementations. There are several designs which provide an intermediate throughput implementation by utilizing 32-bit data path. Several open core implementations of AES encryption and decryption exist. This thesis uses the work done by Rudolf Usselmann [7] and modifies it to present a power efficient model. To allow the simultaneous use of AES encryption and decryption functions, independent implementation models for the two operations are proposed. This research provides encryption performance somewhere between the medium and high throughput range, and employs strategies to implement the design in hardware in the smallest area possible. The model achieves a higher FPGA efficiency compared to previously reported 32-bit architectures. Furthermore, it removes the delay in encryption/decryption caused by pre-calculating keys at every key change. This chapter presents the hardware implementation of AES algorithm based on the strategies discussed in chapter 3. Section 4.1 details the structure used in Key Expansion module. Section 4.2, 4.3 and 4.4 describes Shift Rows, Sub Bytes and Mix-Columns architecture, after implementing power reduction strategies, respectively. Finally, Section 4.4 concludes the chapter with listing the hardware design used in implementation of Add Round Key operation.
4.1 Key Expansion

In order to be feasible for use in low-end embedded applications, this design provides moderate throughput, with a primary focus on low area and low power consumption. There are two general strategies for generating all round keys: the first one is pre-calculation of all the round keys and storing them in RAM for future rounds, the second is to generate the keys on the fly, store them only for the ongoing iteration and discard them once they are done. Both the strategies have pros and cons. The former has the disadvantage of set-up time delay every time the encryption/decryption key is changed as it has to pre-calculate the keys for all $N_r$ rounds of operations. Secondly, it results in excessive memory space usage because all the round keys generated for the $N_r$ round of operations must be stored. The latter has the advantage that it does not have any set-up time as the round key is supplied instantly to every round. But, as these round keys are discarded once they are completed; this results in excessive power consumption to generate the round keys every round. The design presented here attempts to make a design trade-off. It employs 128-bit data-path to provide a higher throughput, and generates and stores all the round keys in memory to prevent the excessive power consumption from keys regeneration. To get an advantage from this approach, prior to each run of the AES computations, pre-computing table lookups will result in 20 256-byte tables to be stored in RAM, that require 6KB of memory. As the decryption process is identical to the encryption process, and utilizes the same round keys used in the encryption process but in opposite order, an approach similar to the encryption process has been used. The purpose of the design is to fit in a low cost FPGA and have low power consumption. Hence, a decision to pre-calculate all round keys and store them in memory is taken.

The key expansion unit is responsible for generating the round keys for round of encryption. Different keys are required for each round. This module accepts the initial key and generates a new key for each round. As explained in Chapter 2, the algorithm requires an initial set of words to start the process and each of the rounds require words of key data. Hence the key expansion unit generates a total of words; where a word refers to a column in the state array. The Key Expansion unit pre-calculates all the round keys and stores them in memory for subsequent use in other iterations. This prevents the excessive power consumption from generating round keys on the fly and discarding them at end of each iteration.

The functions used in this unit are Sub Word, RotWord, and XOR operation. Sub Word is similar to Sub Bytes where a byte is taken as input and the result is the S-Box transformation of the inputted byte.
RotWord function creates a circular byte shift in the word and generates \([a_1, a_2, a_3, a_0]\) from \([a_0, a_1, a_2, a_3]\).

The hardware implementation of Key Expansion unit uses 4 S-boxes for the Sub Word operation in processing the first column of each round. The result is then XOR’ed with the direct previous column and the equivalent column from the previous state which is then XOR’ed with Rcon. Rcon right shifts the result by a bit after all iterations and makes sure that the key is always a byte value over the polynomial \(x^8 + x^4 + x^3 + x + 1\). In hardware, Rcon is implemented as a lookup table stored in RAM which supplies values to be XOR’ed with the result depending on the round of encryption. Every time a new Rcon value is requested by the module at the start of the encryption round, the Rcon counter is incremented by one and stored in memory to keep track of the present ongoing round. During the first set of round keys the first Rcon is supplied to the module, a Rcon counter is incremented by one and stored in memory, so that during the second and subsequent rounds a different Rcon value will be supplied to the module.

The initiation of the process is signaled by setting key_load (kld) to 1. At this point, the encryption key

```plaintext
Key Expansion (byte key[4* N_k ], word w[N_b* ( N_r +1)], N_k )
{
    Word temp
    i=0
    while (i<N_k)
        w[i] = to_word(key[4*i], key[4*i + 1], key[4*i + 2], key[4*i + 3])
        increment i
    end while

    i = N_k
    while (i < N_b* (N_r +1))
        temp=w[i - 1]
        if (i mod N_k ==0)
            begin
                temp=Sub-box(Rotword(temp)) Xor rcon [i/ N_k]
            end
        w[i]= w[i - N_k ] xor temp
        increment i
    end while
}
```

Figure 4.1: Pseudo code for Key Expansion
CHAPTER 4. HARDWARE ARCHITECTURE

4.2 Shift Rows/Inverse Shift Rows

As discussed in Chapter 2, Shift Row is a basic row transformation for obtaining a new state by cyclically shifting state row. The Shift Rows operation is performed by shifting the second, third, and fourth rows to one, two and three cyclic places to the left, respectively. The Inverse Shift Rows operation is performed by
shifting the second, third, and fourth rows to three, two and one cyclic places to the right, respectively. The Shift Rows operation can be done by using a counter and depending on the value of the row index the bytes in the rows are shifted places, or by utilizing two 128-bit dual-port RAM, or by using two 128 bit register and simply wiring them. The former approach has a disadvantage that it requires additional logic circuitry and multiplexor for selecting the appropriate row for shift depending on the row index which results in additional power consumption.

4.3 Sub Bytes/Inverse Sub Bytes

Sub Byte is a non-linear byte substitution that operates separately on each of the state bytes. This byte substitution is done with the help of substitution box, also called S-Box, which is invertible. Several techniques have been adopted for implementation of S-Box till date. Most commonly used S-Box design in FPGA based AES implementations is ROM based. As discussed in Chapter 2, the Sub Bytes function takes 8-bit value as input and returns unique 8-bit output. The other implementations make use of composite field arithmetic, where S-Box elements are calculated using combinational logic circuits. The key step of S-Box is calculating multiplicative inverse of each byte. These designs, because of the mathematical operations of AES over finite field $GF(2^8)$, have high hardware complexity. Some designs implement the multiplicative inverse in $GF((2^4)^2)$. This transformation of decomposing the operations from $GF(2^8)$ to $GF(2^4)$ helps in decreasing the hardware complexity. This approach results in small area occupancy but it has the drawback of being computationally extensive which results in high latency and high power consumption.
This research focuses on a memory based implementation of S-Box. The Sub Bytes values are already calculated before hand and no further processing is required. This implementation requires a significant amount of memory space but the latency in retrieving the sub bytes values from memory is significantly lower than the latency involved in calculating Sub Bytes through Affine transformation. In this design, the memory is used as a large lookup table where each of the substitution bytes value is placed in memory addressed by unique input byte values. When an input byte is placed on the address line the corresponding substitution value appears at the output address lines. The number of S-Boxes required in this step depends on the data path used in the AES implementation. The lowest area designs implement 8-bit data path which require one S-Box in its Sub Bytes step. 8-bit data is passed through this S-Box 32 times to perform 128-bit byte substitution. Hence, these designs require 32 clock cycles for one round of AES implementation. Medium throughput designs employing a 32-bit data path require 4 S-Boxes and a total of 4 clock cycles for one round of AES implementation. Instead of calculating the sub bytes sequentially, this approach utilizes 16 S-Boxes to generate the 16 sub bytes in parallel. It allows one round of AES implementation to be completed in one clock cycle, thereby speeding up the process. Parallelism in the design is obtained by duplicating the same S-Box 16 times as shown in Figure 4.4.

Implementation of a Sub Bytes or an Inverse Sub Bytes function requires 8 bit address lines and 256 8-bit memory locations. So, to implement both Sub Bytes and Inverse Sub Bytes in same memory will
require an additional amount of addressing lines and memory of double storage capacity. The decision to implement both the functions in same memory or independently depends on the memory space available in the target FPGA. In this thesis, there are separate implementation of AES encryption and decryption. Hence, Sub Bytes and Inverse Sub Bytes are implemented in independent memory space. This design is a tradeoff between memory usage and speed of operation, and gives higher preference to reduce the time of operation.

### 4.4 Mix Column/Inverse Mix Columns

Mix Column implementation utilizes the knowledge of the matrix addition and multiplication discussed in Chapter 2. As discussed earlier, this implementation operates on each state column by column, treating each column as a 32-bit word. These columns are treated as polynomial over $GF(2^8)$ and multiplied by a fixed polynomial $a(x)$ modulo $X^4 + 1$, where $a(x)$ is given by:

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

(4.1)

Various implementations of Mix Column/Inverse Mix Column functions have been proposed. An efficient implementation architecture can be derived by applying substructure sharing, both to the computation of each byte and between the computation of the four bytes in a column of the state. As stated in X the matrix multiplication is given as follows:

$$\begin{align*}
S_{out}(\chi)' &= a(\chi) \oplus S_{in}(\chi) \\
S_{out}'_0,c &= \langle \{02\} \cdot S_{in}_0,c \rangle \oplus \langle \{03\} \cdot S_{in}_1,c \rangle \oplus S_{in}_{2,c} \oplus S_{in}_{3,c} \\
S_{out}'_1,c &= S_{in}_0,c \oplus \langle \{02\} \cdot S_{in}_1,c \rangle \oplus \langle \{03\} \cdot S_{in}_{2,c} \rangle \oplus S_{in}_{3,c} \\
S_{out}'_2,c &= S_{in}_0,c \oplus S_{in}_1,c \oplus \langle \{02\} \cdot S_{in}_{2,c} \rangle \oplus \langle \{03\} \cdot S_{in}_{3,c} \rangle \\
S_{out}'_3,c &= \langle \{03\} \cdot S_{in}_0,c \rangle \oplus S_{in}_1,c \oplus S_{in}_{2,c} \oplus \langle \{02\} \cdot S_{in}_1,c \rangle
\end{align*}$$

(4.2)

To achieve resource sharing, the Mix Column polynomial in (4.2) is written as (4.3) where the constant $\{02\}$ is taken out as the common factor:
\[ S_{0,c} = \{02\}_{16} (S_{0,c} \oplus S_{1,c}) \oplus (S_{2,c} \oplus S_{3,c}) \oplus S_{1,c} \]
\[ S_{1,c} = \{02\}_{16} (S_{1,c} \oplus S_{2,c}) \oplus (S_{3,c} \oplus S_{0,c}) \oplus S_{2,c} \]
\[ S_{2,c} = \{02\}_{16} (S_{2,c} \oplus S_{3,c}) \oplus (S_{0,c} \oplus S_{1,c}) \oplus S_{3,c} \]
\[ S_{3,c} = \{02\}_{16} (S_{3,c} \oplus S_{0,c}) \oplus (S_{1,c} \oplus S_{2,c}) \oplus S_{0,c} \]

where \( \oplus \) is the XOR operation.

This transformation of the equation makes use of substructure and reduces the hardware repetition. As of now, the components vectors are generated only once and supplied to the system when required. The multiplication of a number modulo 2 operations is done using the \( Xtime \) function.

The multiplication of a polynomial defined in (2.5) with polynomial \( x \) results in

\[ b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \] (4.4)

The function \( Xtime \) responsible for result generation is defined as follows:

If \( b_7 = 0 \), then the result is already in reduced form.

If \( b_7 = 1 \), then the reduction is accomplished by XORing with polynomial \( m(x) \).

We obtain multiplication with higher powers of \( X \) by repeated application of \( Xtime() \). Hence we have:

\[ \{02\} \cdot S'_{x,c} = xtime(S'_{x,c}) \] (4.5)
\[ \{03\} \cdot S'_{x,c} = xtime(S'_{x,c}) \oplus S'_{x,c} \] (4.6)

The architecture for implementation of Mix Column function is shown in Figure 4.5 [9].

To apply resource sharing in Inverse Mix Column the polynomial in (2.18) is rewritten as (4.7). Here, the multiplication with the constants \( \{0E\}, \{0B\}, \{0D\} \) and \( \{09\} \) is obtained by re-arranging the equation and multiplying with the constants \( \{02\} \) and \( \{04\} \).
\[ S'_{0,c} = (\{02\}_{16}(S_{0,c} \oplus S_{1,c}) \oplus (S_{2,c} \oplus S_{3,c}) \oplus S_{1,c}) + (\{02\}_{16}\{04\}_{16}(S_{0,c} \oplus S_{2,c}) + \{04\}_{16}(S_{1,c} \oplus S_{3,c})) + \{04\}_{16}(S_{0,c} \oplus S_{2,c})) \]

\[ S'_{1,c} = (\{02\}_{16}(S_{1,c} \oplus S_{2,c}) \oplus (S_{3,c} \oplus S_{0,c}) \oplus S_{2,c}) + (\{02\}_{16}\{04\}_{16}(S_{0,c} \oplus S_{2,c}) + \{04\}_{16}(S_{1,c} \oplus S_{3,c})) + \{04\}_{16}(S_{1,c} \oplus S_{3,c})) \]

\[ S'_{2,c} = (\{02\}_{16}(S_{2,c} \oplus S_{3,c}) \oplus (S_{0,c} \oplus S_{1,c}) \oplus S_{3,c}) + (\{02\}_{16}\{04\}_{16}(S_{0,c} \oplus S_{2,c}) + \{04\}_{16}(S_{1,c} \oplus s_{S_{0,c}})) + \{04\}_{16}(S_{0,c} \oplus S_{2,c})) \]

\[ S'_{3,c} = (\{02\}_{16}(S_{3,c} \oplus S_{0,c}) \oplus (S_{1,c} \oplus S_{2,c}) \oplus S_{0,c}) + (\{02\}_{16}\{04\}_{16}(S_{0,c} \oplus S_{2,c}) + \{04\}_{16}(S_{1,c} \oplus S_{2,c})) \]
The implementation of Inverse Mix Column function, utilizing the resource sharing is illustrated in Figure 4.6. This $X4Time$ block, used for polynomial multiplication with the constant \{04\}, is implemented by concatenating two $Xtime$ blocks serially. We can observe that the upper half of the architecture design of Inverse Mix Column resembles the one used for Mix Column implementation. Hence, if encryption/decryption module is implemented in the same module, only the architecture in Figure 4.6 needs to be implemented for both Mix Column and Inverse Mix Column functions. This sub structure sharing at block level helps in decreasing the implementation area thereby resulting in the decrease in implementation cost.

To reduce the mathematical complexity of the architecture some designs decide that rather than implementing multiplication in $GF(2^8)$ they will simply use pre-calculated lookup tables to perform the byte multiplication for 2, 3, 9, 11, 13, and 14. These tables can be store either in LUTs or in memory. Though resulting in decrease in the power consumption, these tables have been found to increase the area and latency of the design considerably. Hence, the above proposed design is considered better in relation with the implementation objectives.
Figure 4.6: Efficient Implementation of Inverse Mix Columns
4.5 Add Round Key

Add Round key function in hardware is implemented by XOR byte operation between the state array and the generated round key from key scheduler. During the first round the round key is same as the initial key supplied and the state array is the initial supplied text for encryption. This operation is performed column wise. Add round key operation for the decryption is identical to one of the encryption stage as the XOR operation is its own inverse. Therefore, only the correct round keys needs to be supplied depending on whether the module is supposed to perform encryption/decryption.
Chapter 5

AES Encryption on Android

This chapter is dedicated to detailed analysis and implementation of data security measures on hand held devices. Section 5.1 discusses the necessity of cryptographic encryption for portable devices. Section 5.2 provides the salient features of both software and hardware encryption. Section 5.3 justifies the use of AES for hardware encryption, followed by description of Android Architecture in Section 5.4 and implementation details of this cryptographic algorithm on Android platform in Section 5.5. Lastly, the results and comparison of AES implementation on Android platform is performed in Section 5.6.

5.1 The Need for cryptographic encryption

Encryption on a Smartphone is important to business because these devices potentially store sensitive information. Following are some of the sensitive information that might be at risk if the device falls into wrong hands:

- Stored username password
- Calendar entries
- Emails, browsing history and caches

If this is not enough, there are now commercially available specialized forensic tools which can recover deleted phone contacts, messages, emails, and nearly everything that was once stored on the device. This
puts the sensitive information to even greater risk as now you do not need extensive experience of the device operation system and file system to recover the data. To allow the widespread adoption of encryption products in portable devices and mobile business, it should:

- Be easily configurable
- Provide high performance
- Be inexpensive

5.2 Salient features of both software and hardware encryption

Once we have decided that the current world scenario demands encryption techniques to be employed and made available on portable devices, the next step is to decide the medium for implementation of these encryption techniques. Encrypted transmissions are processing intensive functions on client and/or server endpoints and this processing load increases with more complex algorithm and greater key length. Hence, the kind of encryption to be employed on the portable device depends on the system factors such as processing capability supported, storage capacity available, frequency of operation, and security level desired.

Encryption process on the host and/or client system can be implemented by one of the two methods:

- Software-based encryptions provide security to the system by utilizing processor and resources of the client/host system.

- Hardware based encryptions provide security to the system by incorporating a dedicated processor and memory in the client/host system.

Table 5.1 lists the differences between the software and hardware based encryption. The comparison shows that the hardware-based encryption is better than software-based encryption on the basis of permanence, system performance and system security. It is more resistant to brute force attacks and independent of the local security of the system on which it is installed. While software encryptions provide a good first line of defense, they are vulnerable to a number of decryption attacks. Hardware encryptions also offer a stronger line of defense against the same threat models. Hence, Hardware-based encryption, when implemented in a secure manner, is demonstrably superior to software-based encryption.
CHAPTER 5. ENCRYPTION ON ANDROID

5.2. SALIENT FEATURES OF ENCRYPTION

Software based encryption

The sharing of processor/resources with other programs could impact the performance of processes running on the system. Since the processor has to support both the normal and data encryption operation it may cause the entire system to slow down.

As software encryption exists in machines software, it needs to be reinstalled if the operating system is changed.

In software based encryption, the cryptographic keys are stored in system resource making it susceptible to malware attacks.

As software based encryption utilize systems resources, there are more brute force attack as these resources can be utilized to break the cryptographic keys.

This encryption is only as secure as the system it is running on. If someone can get malicious onto the system, it can disable the encryption. Hence, rendering the system void of security.

Hardware based encryption

As it uses separate processing resources and runs on its own hardware, it has no impact on the overall system performance.

This encryption remains in place regardless of changes to the system, as they reside outside of the systems software.

The encryption keys are stored in separate dedicated resource, resulting in isolation of keys and making it inaccessible by the operating system. Thereby protecting these security functions from malware attacks.

As this encryption does its processing on a dedicated chip that cannot be accessed by the system, it is not susceptible to brute force attack.

This encryption is not exposed to these types of security flaws as it runs independent of the system.

<table>
<thead>
<tr>
<th>Software based encryption</th>
<th>Hardware based encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sharing of processor/resources with other programs could impact the performance of</td>
<td>As it uses separate processing resources and runs on its own hardware, it has no impact on</td>
</tr>
<tr>
<td>processes running on the system. Since the processor has to support both the normal and</td>
<td>the overall system performance.</td>
</tr>
<tr>
<td>data encryption operation it may cause the entire system to slow down.</td>
<td></td>
</tr>
<tr>
<td>As software encryption exists in machines software, it needs to be reinstalled if the</td>
<td>This encryption remains in place regardless of changes to the system, as they reside</td>
</tr>
<tr>
<td>operating system is changed.</td>
<td>outside of the systems software.</td>
</tr>
<tr>
<td>In software based encryption, the cryptographic keys are stored in system resource making</td>
<td>The encryption keys are stored in separate dedicated resource, resulting in isolation of</td>
</tr>
<tr>
<td>it susceptible to malware attacks.</td>
<td>keys and making it inaccessible by the operating system. Thereby protecting these</td>
</tr>
<tr>
<td>As software based encryption utilize systems resources, there are more brute force</td>
<td>security functions from malware attacks.</td>
</tr>
<tr>
<td>attack as these resources can be utilized to break the cryptographic keys.</td>
<td></td>
</tr>
<tr>
<td>This encryption is only as secure as the system it is running on. If someone can get</td>
<td>This encryption is not exposed to these types of security flaws as it runs independent of</td>
</tr>
<tr>
<td>malicious onto the system, it can disable the encryption. Hence, rendering the system</td>
<td>the system.</td>
</tr>
<tr>
<td>void of security.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of software and hardware based encryption
5.3 AES for hardware encryption

In general, cryptographic algorithms can be classified into three distinct categories: *Private Key Cryptography*, *Public Key Cryptography*, and *Hash Functions* (Figure 5.1). Algorithms that use the same key for encryption and decryption are also known as a *symmetric cipher* whereas those that use different keys is known as *asymmetric cipher*. Symmetric ciphers are significantly faster as compared to asymmetric ciphers but the requirements for key exchange make them more difficult to use safely.

Because asymmetric ciphers are more computationally intensive, they are often used in combination with symmetric ciphers. During the initial stages of a desired secure communication, a public key cryptography mechanism is used for sharing the symmetric key between the receiver and the sender. Once the connection has been established and the key exchange is complete, the symmetric keys are used for both encryption and decryption. Hash functions are then used to verify message integrity. Asymmetric cipher and symmetric cipher used in this process on cell phones are RSA and AES respectively. The asymmetric cipher is only used during key exchange and the bulk of the encryption/decryption is done using the symmetric cipher. Hence, the focus is on efficient hardware implementation of AES to allow its usage in low end portable devices.

Figure 5.1: Three categories of Cryptographic Algorithms
5.4 Android Architecture

The android platform is an open software platform for mobile devices. It includes everything from an operating system, middleware to key applications. This section provides a brief overview of the architecture of the Android platform.

Figure 5.2 shows the block diagram of the Android architecture. We are going to review this model bottom up. Its architecture is based on Linux 2.6 kernel. The Linux kernel acts as an abstraction layer between the hardware and rest of the software stack. Linux is used because it provides memory management, process management, security model, networking, and a robust set of core operating system infrastructures that are well-tested and that have been proven over time.

Above the O/S layer lie the native libraries. These libraries are written in C/C++ and are used by various components of the Android system. They provide buildin support for important Android-specific applications. For example, they provide support for media applications and data-storage, and are optimized for embedded use. Next we have the Android runtime and main component in the Android runtime is the Dalvik virtual machine. Dalvik has been written specifically for Android and allows processes to run
multiple VMs efficiently. Every application running on an Android platform has its own process and its own instance of Dalvik VM. They provide application portability and runtime consistency. The core libraries, next up in the module, provide most of the functionalities which are available in core libraries of Java programming language. They provide a powerful, yet simple and familiar development platform. Then we have Application framework, which provides the tool kit used by all the applications running on the Android OS. These applications include the native applications supplied in the Android platform as well as the custom applications written by the end users. Finally, we have the Applications layer which includes home applications shipped with the mobile device and the third party applications.

5.5 Implementation of AES on Android

To date, the Android platform only supports software encryption which potentially allows hackers to access data off memory (and in particular encryption keys). This vulnerability is far more difficult to exploit if hardware level encryption is used. Hence the thrust of this thesis.

After obtaining a thorough understanding of the Android architecture, the next step is to decide the implementation layer for our hardware encryption model. The hardware implementation of encryption on Android is a two step process. First, we must develop and provide the APIs which will allow the application running on the device to use this feature. Second, we will have to develop the native software libraries that will have the low level hardware access.

Studying the Android software platform, it appears that the Core libraries layer is best suited for the implementation of the application APIs and that the native libraries layer is best suited for the low-level hardware access. The goal of this thesis is to include an encryption library in the android system and make it available for all the applications running on the device. The purpose is not to provide an addition library but to replace the existing javax.crypto library. This method of implementation has following advantages:

1. It prevents breaking of code for the applications currently running on the android platform.

2. The third party application developers do not need to know the underlying architecture and the implemented hardware encryption APIs as they are identical to the software-level encryption APIs.

Thus, when an application running on an Android platform makes a call to the functions in the javax.crypto
encryption library, our system makes a check to determine if the cryptoChip is loaded into the SD card slot. If present, this application is routed to the hardware-based encryption, provided by the cryptoChip. Otherwise, the normal execution continues and the software based cryptographic functions are used.

The check for the presence of the cryptoChip in the SD card slot is achieved with the following code:

```java
if (android.os.Environment.getExternalStorageState().
    equals(android.os.Environment.MEDIA_MOUNTED) {  
    /* This routes the process to pick up hardware 
    based encryption files from cryptoChip. */
}
```

5.5.1 SD Card Emulation

To facilitate the integration of the hardware cryptoChip with the OS, we build the chip in the form factor of an SD card to allow its easy setup and robust access. To simulate the presence of SD card in a device emulator, we needed to create a disk image and then load it to the emulator at start up. Testing this feature on the android emulator is a three step process:
CHAPTER 5. ENCRYPTION ON ANDROID

5.5. IMPLEMENTATION OF AES ON ANDROID

Creating an SD card disk image:  A SD card image can be created using android tool with the command:

`android create avd -n <avd_name> -t <targetID> -c <size> [K | M]`

Copying encryption files to SD card disk image:  Once the disk image has been created, the files can be copied to the image either by using a utility such as mtools or by first mounting the image as a loop device and then copying files to it.

Loading the image to emulator at start up:  This can be done by specifying the name and path of the image file at startup, as shown below:

`emulator -sdcard <filepath>`

5.5.2 Problems faced during AES implementation on Android

Some of the issues that came up during implementation and testing of AES on the Android platform are:

No support for determining SD card insert/eject.  The first condition our model checks on initiation is whether the SD card is present in the memory slot.  This is an important property for our model as depending on this condition, the system makes a decision whether to pick up the encryption libraries from the SD card or from the built-in crypto core.  The issue faced during testing was that the android emulator does not have support for determining SD card insert/eject [16].  To work around with this issue, the model skips the step where it checks for the presence of SD card and assumes that the crypto chip is always present in the memory slot.

Crypto chip library name clash.  If we declare the crypto chip library with same name outside of the default library path `javax.java.crypto`, the android environment reports a name clash.  Two methods can be adopted to bypass this issue:

1. If defined, outside default library path, the Crypto chip libraries names can be the same name and a different directory structure so as to avoid the name clash with the default android environment.
2. If defined, inside the default library path, the Crypto chip libraries can be given a different name and same directory structure so as to avoid a name clash.
The issue with the above two methods is that it still does not allow same names for the crypto chip and built-in crypto libraries. Due to this, a conditional statement has to be included in the code so as to allow the crypto chip libraries to be picked up in place of the default cryptographic libraries, when an application makes a call for the default cryptographic libraries.

Considering the limitations discussed above, the best solution was to add the crypto library as the native library in a different directory structure. The directory structure and the library files of the crypto chip are kept identical to ones of the android crypto library located under /libcore/crypto. This prevents breaking of code and the user of the library does not have to make any changes to adapt to the new library. Most of the time, the user will not realized that the function calls to the android crypto library are being executed by the hardware encryption libraries located in crypto chip.

The crypto chip native libraries are located at /device/sample/framework with the name cryptoChip. The java and jni folder under the cryptoChip contains the java and the native C++ files respectively. The java files make calls to the functions implemented in the C++ files. The C++ implementation generally make the execution of code faster as the code runs directly on top of kernel rather than running on top of java virtual machine as in case of java code.

### 5.5.3 Native code library

As stated above, the native code file, cryptoChip.cpp, is written in C++ and resides under the /cryptoChip/jni folder. The Android.mk makefile to build this is:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := samples
LOCAL_MODULE:= cryptoChip_library_jni
# The source files to be compiled
LOCAL_SRC_FILES:=
    cryptoChip.cpp
# Additional shared libraries
LOCAL_SHARED_LIBRARIES := \n    libandroid_runtime \n    libnativehelper \n    libcutils \
```
To build the shared libraries using the Android.mk file, the following steps need to be executed on command prompt:

```
cd /$ANDROID_HOME/external/myapps/cryptoChip/jni
source envsetup.sh
mm
```

The above commands result in shared library being built from the cryptoChip.cpp source file. The final output is:

```
Install: out/target/product/generic/system/lib/libMult.so (no symbols) (5 KB)
target SharedLib: libMult (out/target/product/generic/obj/
    SHARED_LIBRARIES/libMult_intermediates/LINKED/libMult.so)
```

### 5.5.4 Java Application

In the previous section, we have created and built native source code. Now, in this section, we will be writing a java package that will be using the native code libraries for implementation. The java package file, cryptoChip.java, is located at /cryptoChip/java/com/example/android/cryptoChipPkg/.

The file AndroidManifest.xml must be added to the folder cryptoChip. This file contains:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.cryptoChipPkg">
    <activity android:name="cryptoChip">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</manifest>
```
The Android.mk file is located in folder cryptoChip and contains:

```make
LOCAL_PATH:= $(call my-dir)
   include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := samples
LOCAL_PACKAGE_NAME := cryptoChipPkg
LOCAL_SRC_FILES := com/example/android/cryptoChipPkg/cryptoChip.java
LOCAL_SDK_VERSION := current
include $(BUILD_PACKAGE)
```

To build the cryptoChip.java using the Android.mk file, the following steps need to be executed on command prompt:

```bash
cd $ANDROID_HOME/external/myapps/cryptoChip/
source envsetup.sh
mm
```

This command will result in the cryptoChip package being built from the cryptoChip.java source file. The final output is:

```
Install: out/target/product/generic/system/app/cryptoChipPkg.apk
```

This application package only contains the Dalvik class executable and not the native shared library libcryptoChip.so. So, to get rid of dependencies, we need to push both the files into the device. Then, any application running on the device will be able to successfully use the cryptoChipPkg package.

### 5.6 Results and Comparison

There are two concerns that have to be satisfied to ensure that the hardware cryptoChip implementation is feasible in the android platform, namely:

**We need to ensure that it does not create a performance bottle neck.** In an embedded system, the time taken for encryption/decryption of a message is the sum of the execution time of the encryption/decryption when performed in hardware and the time taken for transfer of data in communication channel between the SD card and the operating system.
TABLE 5.2: Peak Data rates for different Telecommunication services

<table>
<thead>
<tr>
<th>Telecommunication service</th>
<th>Peak Data rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G</td>
<td>NA</td>
</tr>
<tr>
<td>2G</td>
<td>150 kbps</td>
</tr>
<tr>
<td>3G</td>
<td>200 kbps</td>
</tr>
<tr>
<td>4G</td>
<td>100Mbps - 1Gbps</td>
</tr>
</tbody>
</table>

Total Encryption time = Encryption time in hardware + time taken in data transfer

Currently available high speed SD cards are capable of read/write access speed of 720 Mbps [17]. As shown in Table 5.2, the rate of encryption/decryption on the hardware is 1099 Mbps and 710 Mbps. Hence, our design is capable of extracting the best performance from the mobile device. Any bottleneck in the performance would be due to the low data transfer rate of the handheld device.

**The throughput must be comparable to the current data transfer rates.** To be feasible for the cryptoChip libraries to be used by applications running over the internet, the throughput of the module after implementation on android, should be comparable with the data transmission rate over the internet, for handheld devices.

Table 5.2 shows the data rates provided by the telecommunication services. Our design meets the specification mentioned for 4G data rates. Hence, implementation of hardware encryption in handheld devices will not result in any loss in performance.
Chapter 6

Experimental Results and Discussion

This chapter is organized as follows. Section 6.1 presents the behavioral simulation of the AES Encryption and Decryption module. Section 6.2 provides the performance and resource usage table of the AES cipher. Section 6.3 lists the five design parameters to be kept in mind while comparing this cipher to the existing designs and the resulting comparison is given in Section 6.4

6.1 Behavioral simulation of AES Encryption/Decryption model

The AES design detailed in preceding chapters is developed and tested using Xilinx ISE design flow. Xilinx ISE and ModelSim are used for Simulation, Synthesis and Implementation of the model. The AES Encryption/Decryption core is written in Verilog. A software model in C++ has also been developed to verify the results of this hardware model of AES Encryption/Decryption algorithm.

6.1.1 AES Encryption

The AES Encryption cipher consists of four modules namely: key expansion module, an initial permutation module, a round permutation module, and a final permutation module. It takes an encryption key and plain text as input, and produces cipher text after 12 clock cycles. The start of the encryption process is indicated by asserting ld pin high for one cycle and the end of the encryption cycle is indicated by asserting done signal high for one clock cycle.
CHAPTER 6. EXPERIMENTAL RESULTS

6.1. SIMULATION OF AES

Figure 6.1: AES Encryption module

Figure 6.2: AES Key Expander module

Figure 6.1 shows the timing diagram of the AES Encryption process. The length of the encryption key, plain text, and cipher text is 128-bit each.

6.1.2 AES Decryption

The AES Decryption process is essentially the mirror image of encryption process. Supplying AES encrypted cipher text to the AES decryptor will yield the original plain text. The inverse of the function used in the AES encryptor are designed for the decryptor. The same roundkeys are used, but are required in the reverse order. Using the same block for both AES encryption and decryption is not efficient as mirroring makes the implementation quite different. In addition to that, the inverse round key function is significantly complex as compared to that used for encryption, making this hardware bigger and slower. This is the reason we decided to go for independent implementation of both AES encryptor and decryptor.

This model takes 12 clock cycles, which includes one cycle of initial key loading, one cycle for the output stage and 10 cycles for 10 round of decryption, to generate plain text from cipher text. The caveat is that the encryption key needs to expanded and loaded into the memory before the process can start because the last expanded key is used in the first cycle and the first expanded key is used in the last cycle of decryption process. Once the key is loaded and the expanded keys are generated, these keys are stored in the internal buffers and can be reused for subsequent decryption processes for the same key.
6.1. SIMULATION OF AES

Figure 6.3: AES Decryption module

Figure 6.4: AES Encryption and Decryption module

Figure 6.2 shows the timing diagram for the Key expansion process. It starts when the kld signal is asserted high and once the key expansion is complete, the kdone signal is set high for one clock cycle. The Key expansion and the decryption process cannot be done in parallel and the key must always be loaded first.

Figure 6.3 shows the timing diagram for the Decryption process. The initiation of the decrypt process is indicated by asserting ld signal high and when the decryption is complete the done signal will be set high for once clock cycle.

6.1.3 AES Encryption/Decryption combined

In this section, we perform the encryption and decryption of the input sequence in succession. Figure 6.4 shows timing diagram of the combined process. This takes a total of 24 clock cycles, 12 for encryption and 12 for decryption. It can be verified that the output of the Decryption block is same as the input to the Encryption block.
### Table 6.1: Resource Usage and Performance of AES cipher

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Elements</td>
<td>398</td>
<td>627</td>
</tr>
<tr>
<td>Look Up Tables</td>
<td>722</td>
<td>1190</td>
</tr>
<tr>
<td>BRAMs used</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Usage (%)</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Frequency Max(MHz)</td>
<td>1020.3</td>
<td>666.5</td>
</tr>
<tr>
<td>Throughput Max(Mbps)</td>
<td>1099</td>
<td>710.9</td>
</tr>
<tr>
<td>Latency (Cycles)</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

### Table 6.2: Power Efficiency of AES cipher at 50 MHz

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP&lt;sub&gt;0.50 MHz&lt;/sub&gt;</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td>Ps&lt;sub&gt;0.50 MHz&lt;/sub&gt; (mW)</td>
<td>144</td>
<td>145</td>
</tr>
<tr>
<td>Pd&lt;sub&gt;0.50 MHz&lt;/sub&gt; (mW)</td>
<td>50</td>
<td>82</td>
</tr>
<tr>
<td>P&lt;sub&gt;0.50 MHz&lt;/sub&gt; (mW)</td>
<td>195</td>
<td>226</td>
</tr>
<tr>
<td>E&lt;sub&gt;per OP&lt;/sub&gt; (nJ)</td>
<td>46.8</td>
<td>54.24</td>
</tr>
</tbody>
</table>

### 6.2 Performance, Resource Usage and Energy efficiency of the AES cipher

Table 6.1 provides a design summary of the implemented AES Encryption/Decryption model. It shows the resource usage and performance attained for both design models. The design is implemented on Xilinx Spartan 3 XC3S 1500, which is one of the smallest FPGAs available.

As can be seen in the Table 6.1, both the designs use roughly the same amount of the FPGA space and result in under-utilization of the available resources. We believe that this design can be implemented on smaller FPGAs as well. But, we were not able to confirm this as those FPGAs were not available in the free ISE Webpack Tool [18].

Next, we provide a table for throughput, power used and energy used per operation for the design. Throughput is the number of bits processed per second. Power used is the sum of the static and dynamic power dissipated by the model. Energy used per operation is the product of average power and the time duration for one operation cycle. The detailed description of the above terms can be found in Sections 2.4.1, 2.4.2 and 2.4.3 respectively.

Tables 6.2 and 6.3 provide a summary of the energy usage of the AES Encryptor and Decryptor at 50 MHz and $F_{max}$ respectively. A 50 MHz operating frequency is used for power analysis because most of the
Operation | Encryption | Decryption
---|---|---
\(F_{\text{max}}\) | 1020.3 | 666.5
\(TP_{\text{@}F_{\text{max}}}\) | 1099 | 710.9
\(P_{S_{\text{@}F_{\text{max}}} (\text{mW)}}\) | 153 | 150
\(P_{D_{\text{@}F_{\text{max}}} (\text{mW)}}\) | 574 | 435
\(P_{\text{@}F_{\text{max}}} (\text{mW)}\) | 726 | 585
\(E_{\text{perOP}} (\text{nJ)}\) | 8.5 | 10.5

Table 6.3: Power Efficiency of AES cipher at \(F_{\text{Max}}\)

existing implementation of AES have used frequency in this range. Thus, this will provide a better direct comparison between the results of the different systems.

It is important to include static power consumption in our results as it is a major contributor in the power dissipation, by the design. Disregarding this factor will result in energy efficiency estimates being overly optimistic. By studying the above results, we observe that the static power consumption remains the same and the dynamic power consumption increases as we increase the operating frequency of the design.

Figure 6.5 and 6.6 provides a graph of energy consumed, for both encryption and decryption cycle, when the design was simulated at every frequency from 10 MHz to \(F_{\text{max}}\) in 50 MHz increments. As can be seen in the graph, the energy efficiency is directly proportional to the operating frequency of the design and reducing the clock speed to less than 200 MHz, for AES encryptor, reduces the energy efficiency of the

![Energy per operation vs frequency for AES Encryption](image)

Figure 6.5: Energy per operation vs frequency for AES Encryption
6.3 Design Parameters

Various architectures for hardware implementation of AES algorithm have been published by previous researchers [8] [9] [11] [12] [19]. Unfortunately, there is no easy, proven measure to compare designs of different architectures. In particular, the designs are implemented on FPGAs from different manufacturers. Even the designs implemented on the devices from same manufactures differ in family and different families utilize different technology which makes it even more difficult. For example there is an architecture change in Virtex 5 where it has four LUTs per slice as compared to two LUTs per slice in previous families. In this chapter we will classify design into different categories and then use these categories to compare their performance.

6.3.1 AES Encryption and Decryption

Some architecture provides implementation for both encryption and decryption functions in the same chip. Owing to the symmetric nature of the algorithm it helps in sharing of the units and eventually results in reduction in implementation area. These designs include a parameter which decides which operation needs to be performed on the input text. Some other designs provide standalone implementations for encryption.
and decryption functions.

### 6.3.2 Key Sizes

The AES algorithm provides encryption/decryption function for data of block size 128 bit and supports three key sizes 128, 192 and 256 bits. Although most of the designs target 128-bit key size, as it being the most extensive used key size. However, there are some designs, including ours, that have been implemented to support all three key sizes.

### 6.3.3 Key Schedule

The key schedule needs to be implemented in every AES implementation. The round keys can be previously calculated and stored in memory. This results in initial set up time and requires excessive memory usage. A different approach generates keys on the fly for each round and discards them on round completion. This affects the operating frequency of the design as the key schedule and the data unit shares the same clock.

### 6.3.4 S-box implementation

The performance of the AES algorithm is significantly influenced by the architecture design used in the s-box implementation. A BRAM based approach is preferred for low area designs as it saves the slices required in combinational block based approach. Hence the implementation results of BRAM based approach cannot be compared with those of combinational logic based approach. In our analysis, we will only compare our design with other BRAM based implementations.

### 6.3.5 Power Analysis

Power constraints is one of the main issues in today’s handheld devices. Most of the AES implemented designs just focus on throughput and implementation area, and totally neglecting the architecture’s power requirements. This research tries to make the design as efficient as possible while keeping in mind this additional parameter.
### 6.4 Design comparison with existing models

As stated above, the performance of AES implementation on hardware depends on the various design choices. Hence, the results comparison in this section uses the designs which are capable of both encryption and decryption, which use a 128-bit key for the process, which utilize BRAM approach for s-box implementation and which provide the design’s power usage. The designs are compared with respect to resource usage, performance, and energy efficiency.

Table 6.4 summarizes the resources usage and performance chart for several FPGA design solutions. The first two entries labeled “Encrypt” and Decrypt” report the results from this thesis; the other entries have names/citations to identify their authors. The design proposed by D. Kenney [15] shows better results as compared to other designs. It is implemented on different FPGA, hence the exact comparison of the resource usage is quite difficult. It is impossible to tell exactly how many LEs will be used when implemented on Xilinx FPGAs. For comparison of designs, we have taken a very optimistic approach that one LE of Altera FPGA is equivalent to one LE of Xilinx FPGA.

The design proposed by Katashita et al [13] does not include a hardware for key expansion. The keys have to expanded and loaded into the hardware for the encryption/decryption process to begin. Inclusion of the Key expansion hardware in this model will result in increase in resource utilization.

The low resource utilization of Rejeb et al [14] is expected as the design is intended to be very compact. The design uses lower datapath width to reduce area and power consumption. Unfortunately, this design results in very high latency as smallest design uses one s-box and the biggest one uses 4 s-box. Hence, the data has to be passed again and again through these s-boxes to perform one cycle of encryption or decryption.
Table 6.5: Comparison AES Energy Efficiency

The work in this thesis provides a good balance between the resource utilization, throughput, and latency. The lower throughput reported for the decryption model is because of complex hardware architecture required in inverse round key operation. Still, an attempt has been made to increase throughput as much as possible without increasing the area usage.

The designs considered for comparison do not have data available for static power consumption. As we discussed in Section 6.2, static power consumption is a major contributor in draining device battery. So, if we do not consider this factor, the results that we get will be overly optimistic. Our design’s power consumption results suggest that the static power consumption is around 25% of the dynamic consumption. In order to make other designs comparable with our design, we have added normalized 10% of each designs dynamic power consumption as static power consumption and then proceeded with energy per operation calculation. The final figure might not be totally accurate but that is the closest we can get in comparison.

Among the designs considered, our design provides the smallest energy required per operation. One reason for other design performing poor in comparison with our design is that our design has independent module for both AES encryption and decryption. Hence, we were able to customize both the modules individually to provide better energy efficiency whereas other designs use same module for both the operation. Another reason is that we have used a smaller FPGA whereas other design have been implemented on higher families of FPGA which provide higher speed at the cost of greater power consumption for same area utilization. Lastly, as we have shown in graph 6.5 and 6.6, the energy per operation decreases as frequency
increase and all the selected designs are low frequency implementation. We believe that the designs would performed better, in terms of energy efficiency, if capable of operating at high frequency.

One of the reason for the average performance as compared to other designs is because other designs have been implemented on higher family of FPGAs that have better device architecture for reduction of static and dynamic power consumption. Although the Virtex family is faster, power efficient and better than Spartan family. Unfortunately, this speed and power efficiency comes at a higher cost. The goal of thesis was to provide the implementation on a lost cost FPGA so as to prove that it is feasible to distribute AES cipher in FPGAs. Hence, providing the implementation on a higher family of FPGA would have defeated the purpose.
Chapter 7

Conclusion and Future Work

This chapter presents the conclusion of this thesis, current adaption of AES hardware encryption and the proposed future work.

7.1 Conclusion

In this thesis, space and power efficient hardware implementation of AES is discussed. The focus of research is to make the hardware design to fit in smallest possible FPGA, have low power consumption and produce acceptable throughput. The hardware design, written in Verilog, fits in low cost Spartan 3 FPGA, operates at 1020 MHz and 710 MHz, produces a throughput of 1099 and 710.9 Mbps and consumes 8.5 and 10.5nJ of energy per operation cycle, respectively.

Three main contributions of this research are hardware implementation of energy efficient AES algorithm, analysis of power consumption of FPGA based AES algorithm and first known attempt at implementation of hardware encryption on Android platform. During the research, critical decision were made to include trade-offs between hardware design size and power consumption. An attempt was made to include hardware encryption on Android platform with minimal changes to the OS. Also, this hardware design does not produce any bottleneck in performance at present data transmission rates.
7.2 Current adaptation

Since AES is currently the dominant block cipher and is being used in various protocols, some manufacturers have decided to provide hardware-based AES encryption available in their devices. Below is the list of some of the prominent device manufacturers:

1. Apple is the first one to include hardware-based encryption in their Smartphone model iPhone 3GS and later versions. This extra security feature was long due for iPhones and is aimed mostly at enterprise deployments. iPhone hardware encryption uses AES 256-bit encoding to protect all data on the device.

2. Beginning with the new 2010 Intel Core processor, Intel has made AES instruction available for use. These instructions enable fast and secure data encryption and decryption using AES. They have the flexibility to support all usages of AES, including all standard key lengths and standard modes of operation. In addition to improving performance, this instruction set helps in eliminating major timing and cache based attacks which is a concern in software implementation of AES.

7.3 Future work

This research has opened up new research areas in which possibilities could be explored. Firstly, improvements can be made to the present hardware design to make it more energy efficient. In this research, the add round keys for all the rounds were generated in advance and stored in memory to prevent extra energy consumption from generating keys on the fly and discarding them after use. This on the fly keys generation method can be experimented to evaluate its impact on energy efficiency.

Secondly, this design was implemented on Spartan 3 and used the Xilinx Power Analyzer to analyze the efficiency of the hardware design. There are other design families which have inbuilt capabilities of reducing power consumption. So, implementing this design on FPGA’s of various sizes, complexities and architecture will help in exploring design’s effectiveness and efficiency.

Lastly, this Android implementation did not store the encryption source files in an SD card and rather did it in a different directory structure because the android emulator does not support SD card insert/eject. Other ways of testing this functionality need to be evaluated.
Bibliography


