DESIGN OF A LOW POWER AES PROCESSOR

By
Fakir Sharif Hossain

MASTER OF SCIENCE
IN
INFORMATION AND COMMUNICATION TECHNOLOGY

Institute of Information and Communication Technology
Bangladesh University of Engineering and Technology (BUET)
2012
AUTHOR'S DECLARATION

This is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

__________________________

Fakir Sharif Hossain
# TABLE OF CONTENTS

Board of Examiners ii
Author’s Declaration iii
Table of Contents iv
List of Figures v
List of Tables vi
List of Abbreviations of Technical Symbols and Terms vii
Abstract x
Acknowledgements xi

## Chapter 1: Introduction 1
  1.1 Introduction 1
  1.2 Motivation 2
  1.3 Contribution of this Thesis 4
  1.4 Thesis Outline 4

## Chapter 2: Advanced Encryption Standard (AES) 5
  2.1 Basic Mathematics for AES 5
  2.2 Addition 5
  2.3 Multiplication 7
  2.4 Finite Fields and Composite fields 7
    2.4.1 AES Arithmetic over Finite Field $GF(2^8)$ 8
    2.4.2 AES Arithmetic over Composite Field $GF((2^4)^2)$ 8
  2.5 Overview of AES Algorithm 9
    2.5.1 SubBytes 10
    2.5.2 ShiftRows 12
    2.5.3 MixColumns 13
    2.5.4 AddRoundKey 14
    2.5.5 Key Expansion 15
  2.6 Overview of FPGA 16
    2.6.1 FPGA Cyclone II Device 18
    2.6.2 Development Tool Quartus II 19
Chapter 3: **Low Power Design Considerations**  
3.1 Introduction 20  
3.2 Power Analysis 20  
3.3 Design 22  
   3.3.1 Design Considerations 24  
   3.3.2 Composite Field Based S-Box Architecture 35  
   3.3.3 Implementation of the Processor 36  

Chapter 4: **FPGA Implementation** 37  
4.1 Introduction 37  
4.2 Implementation using FPGA 37  
4.3 Components of the Processor 38  

Chapter 5: **Experimental Results and Discussions** 42  
5.1 Introduction 42  
5.2 Resources Used 42  
5.3 The Simulation Results 43  
5.4 Power Analysis and Measurement of Power consumption 48  
5.5 Comparison with Other Related Works 50  

Chapter 6: **Conclusions** 53  
6.1 Conclusion 53  
6.2 Further Works 54  

References 55  

Appendix A Encryption 62  
Appendix B Decryption 72  
Appendix C Verilog HDL code for GF($2^8$) 83  
Appendix D Pin Assignments 84  
Outcomes of this Thesis 90
LIST OF FIGURES

Figure 2.1: Flow chart for AES Algorithm 12
Figure 2.2: Sub Byte Transformation 13
Figure 2.3: Shift Row Transformation 14
Figure 2.4: Inverse Shift Row Transformation 15
Figure 2.5: MixColumn Transformations. 15
Figure 2.6: AddRoundKey Process. 16
Figure 2.7: Key Expansion module. 17
Figure 3.1: Minimum power-mapping for Gates. 26
Figure 3.2: Complete s-box creation in GF(2^4) 27
Figure 3.3: Legends for the building blocks within the s-box creation 27
Figure 3.4: Hardware implementation of squaring in GF(2^4) 28
Figure 3.5: Hardware implementation of constant multiplication, λ in GF(2^4) 29
Figure 3.6: Hardware Implementation of Multiplication in GF(2^4) 29
Figure 3.7: Hardware Implementation of Multiplication in GF(2^2) 30
Figure 3.8: Hardware Implementation of Multiplication in GF(2^4) 31
Figure 3.9: Multiplicative Inverse in GF(2^4) 35
Figure 3.10: Equivalent multiplication block 37
Figure 3.11: Multiplication operation in GF(2^4). 38
Figure 3.12: Pipelined SubBytes in composite field GF(2^4)^2. 39
Figure 4.1: Low power AES processor 41
Figure 4.2: Block Diagram of AES Encryption module 42
Figure 4.3: Block Diagram of OneToneNine Sub module 43
Figure 4.4: Block Diagram of S-BOX Sub module 43
Figure 4.5: Block Diagram of Shift Rows Sub module 44
Figure 4.6: Block Diagram of MixColumn Sub module 44
Figure 5.1: Simulation result window for s-box operation 46
Figure 5.2: Simulation waveform of Shift Row. 46
Figure 5.3: Simulation waveform of Mix Column Transformations. 47
Figure 5.4: Simulation waveform of OneToneNine Rounds 47
Figure 5.5: Simulation wave shape for Full Encryption Module 48
Figure 5.6: Simulation wave shape for Full Decryption Module 49
Figure 5.7: Power consumption of full Encryption module 51
LIST OF TABLES

Table 2.1: Shows the comparison of addition in normal algebra and in GF(2^n). 15
Table 3.1: Multiplicative inversion in GF(2^4) using different modulo polynomials 36
Table 3.2: Pre-computed results of the multiplicative inverse and constant multiplier operation in GF(2^4). 38
Table 5.1: Thermal Power Dissipation by Block Type 52
Table 5.2: Comparison with other research works 54
### LIST OF ABBREVIATIONS OF TECHNICAL SYMBOLS AND TERMS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DES</td>
<td>Triple-DES (Data Encryption Standard)</td>
</tr>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>ALM</td>
<td>Adaptive Logic Module – basic building block of Stratix family</td>
</tr>
<tr>
<td>ALUT</td>
<td>Adaptive Look-Up Table</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>CBC</td>
<td>Cipher Block Chaining</td>
</tr>
<tr>
<td>CFB</td>
<td>Cipher Feedback</td>
</tr>
<tr>
<td>CHES</td>
<td>Cryptographic Hardware and Embedded System</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DPA</td>
<td>Differential Power Analysis</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>Eq.</td>
<td>Equation</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standards</td>
</tr>
<tr>
<td>FIPS PUB</td>
<td>Federal Information Processing Standards Publications</td>
</tr>
<tr>
<td>f_{\text{MAX}}</td>
<td>Maximum Clock Frequency</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPL</td>
<td>Field Programmable Logic</td>
</tr>
<tr>
<td>Gbps</td>
<td>Giga bits per second</td>
</tr>
<tr>
<td>GF</td>
<td>Galois Field</td>
</tr>
<tr>
<td>GX</td>
<td>Gigabit Transceiver</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>InvMixColumns()</td>
<td>Inverse Mix Columns operation</td>
</tr>
<tr>
<td>InvSubBytes()</td>
<td>Inverse Substitute Bytes operation</td>
</tr>
<tr>
<td>INV_{S-Box}</td>
<td>Inverse Substitution byte operation</td>
</tr>
<tr>
<td>ISCAS</td>
<td>International Symposium of Circuits And Systems</td>
</tr>
<tr>
<td>K</td>
<td>Cipher Key</td>
</tr>
<tr>
<td>LAB</td>
<td>Logic Array Block – a physically grouped set of logic cells</td>
</tr>
<tr>
<td>LC</td>
<td>Logic Cell</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>LNCS</td>
<td>Lecture Notes in Computer Science</td>
</tr>
<tr>
<td>LPM</td>
<td>Library of Parameterized Module</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>M4K</td>
<td>Memory block of 4096 bits</td>
</tr>
<tr>
<td>M512</td>
<td>Memory block of 512 bits</td>
</tr>
<tr>
<td>MixColumns()</td>
<td>Forward Mix Columns operation</td>
</tr>
<tr>
<td>M-ROM</td>
<td>Mega RAM</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PLD</td>
<td>Programmable Logic Device</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>Rcon[]</td>
<td>The Round Constant word array</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only memory</td>
</tr>
<tr>
<td>RotWord()</td>
<td>A Function that performs a cyclic permutation</td>
</tr>
<tr>
<td>S-box</td>
<td>A lookup table that holds non-linear substitute byte values</td>
</tr>
<tr>
<td>SubBytes()</td>
<td>Forward Substitute Bytes operation</td>
</tr>
<tr>
<td>Tco</td>
<td>Clock-to-Output Time</td>
</tr>
<tr>
<td>Th</td>
<td>Hold Time</td>
</tr>
<tr>
<td>Tpd</td>
<td>Point-to-Point Delay</td>
</tr>
<tr>
<td>Tsu</td>
<td>Setup Time</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
<tr>
<td>Word</td>
<td>A group of 32 bits</td>
</tr>
<tr>
<td>Xor</td>
<td>Exclusive-OR</td>
</tr>
<tr>
<td>⊕</td>
<td>Exclusive-OR operation</td>
</tr>
<tr>
<td>⋅</td>
<td>Matrix Multiplication</td>
</tr>
</tbody>
</table>
Abstract

The National Institute of Standards and Technology (NIST) announced Rijndael as the new Advanced Encryption Standard (AES) in 2001. After that a lot of researchers are working to improve the performance of AES in three areas; speed, memory size and power. Depending on various applications, the low power consumption criteria has become a great issue in small computing devices like contactless smart card, wireless sensor, small computing devices, RFID etc. This thesis proposes the design of a low power AES processor without sacrificing its security, compactness and throughputs. Novel techniques have been introduced to meet the low power criteria in designing the processor. The proposed design consists of a composite field based design of substitution box (s-box) in the finite field GF \(2^{4^2}\) rather than the original design in Galios Field \(2^8\). An optimized s-box is also considered for this architecture. Look up table based implementation for two operations which are multiplicative inverse and multiplication by constant is performed in s-box. The proposed processor also mitigates the glitches that occur for unequal path lengths in the s-box and inverse s-box. To reduce the unwanted paths or glitches from the datapath pipelined structure of 3-multipliers is used. This design does not hamper the security of the AES algorithm. The processor is simulated in Quartus II simulation software in the Altera Cyclone II FPGA family device and analyzed in terms of power consumption for the processor. The performance of the processor are compared to those of other research works and found the superiority over other research works.
Acknowledgments

At first I would like to thanks Almighty Allah (SWT) for providing his divine blessing, unlimited mercy to me. During the time I was performing my master’s thesis, I came across many people who have supported and assisted me. First, I want to express my heartiest thanks to my supervisor, professor Dr. Md. Liakot Ali, for giving me the opportunity to do my master’s thesis under his supervision. I would like to express many thanks for his invaluable advice and ideas on the research and also for his devotion of time to me during this program. His support and expertise resolved many hurdles that I encountered throughout the research. Without his continuous support, even in my personal matters, this thesis could not have been completed.

I also gratefully acknowledge the valued advice and support from Professor Dr. Saiful Islam, Director, IICT. My special thanks go to all the staffs of IICT, BUET.

Finally I thank my beloved wife Khairun Nahar for her endless support in all respects which helped me devote to the work.
Chapter 1

1.1 Introduction

This is an age of information and communication technology (ICT). The rapid growth in computer systems and their interconnections via networks has increased the dependence of both organizations and individuals on the information stored and its communication using these systems. Security is an important term in ICT. Now a day’s security has become the key aspects of modern civilization. The security involves in many applications such as mobile networks, automated teller machines (ATMs), copy protection (especially protection against reverse engineering and software piracy), internet e-commerce, internet banking, military and government to facilitate secret communication and many more. Cryptography plays an important role in security system. In cryptography the secret data is encrypted by a secret key and the encrypted data can only be deciphered if one has the key. The encryption algorithm is asset of well defined steps to transform data from a readable format to an encoded format using the key. This set of well defined steps is also called cipher. A symmetric-key cryptography algorithm named data encryption standard (DES) has been the encryption standard since 1977. It has been widely used but its 56-bit long key size has been criticized since its inception. Triple DES with triple key size of DES offers higher security but it is inefficient in software due to its large key size [1].

Encryption algorithms are broadly classified as symmetric and asymmetric algorithms based on the type of keys used [2]. Symmetric algorithms have one key for both parties. The key needs to be private or secret to maintain confidentiality. So both the sender and receiver need to have the same key. Asymmetric algorithms have two keys. One is the public key and the other is the private key. The data is encrypted by using the public key and is decrypted by the receiver using the private key. Due to the complexity of asymmetric algorithms, the symmetric ciphers are always preferred for speed and simplicity [3]. AES based on the Rijndael algorithm is one such symmetric algorithm for encryption. In 2001, the national institute of standards and technology (NIST) announced the approval of the federal information processing standard (FIPS) for the AES, FIPS-197 [4] and approved it to be used for government organization to protect sensitive information [5]. As a replacement of DES, AES is presently widely used in both software and hardware implementations.

It has been used for countless different applications ranging in size and scale such as military, e-banking and different data communication purposes. Every application has different
requirements such as the speed at which security operations must be performed, the physical area for embedded hardware, or its power budget. Before the year 2007 researchers mainly focus on speed and memory size of AES processor. Now a day there are a lot of applications coming in the market where an increasing number of battery-powered embedded systems like PDAs, cell phones, networked sensors, smart cards, RFID etc are used to store, access, manipulate, or communicate sensitive data. It makes security an important issue. Since those devices are resource constrained and battery powered, low power and small area are the mandatory requirements. This thesis focuses on the solution of this problem and has presented a novel technique in designing low power AES processor.

1.2 Motivation
Since AES become the no. one security algorithm all over the world in 2001, so AES has become a widely known for privacy and security of information [4]. It has been used for countless different applications ranging in size and scale such as military, e-banking and different data communication purposes. There have been many novel design techniques for AES that focus on obtaining high throughput or low area usage [5-14]. References [5-10] describe the design of AES that focusing on improvement of speed where maximum speed is reported 36.4 Gbps. References [11-14] are about the optimization of area and cost. However there are lot of applications coming out at present where low power is a must such as contactless smart card, wireless sensor network, small computing devices etc. Due to this reason, a number of research works have been proposed and further research is going on focusing on low power [15-18]. Researchers have proposed pipelining technique to increase the throughput and reduce the power consumption of designs [18]. This technique makes the glitch unable to propagate as far and power savings are obtained thereby. Studies show that basic pipelining technique can reduce 31% dynamic power consumption while it can be increased up to 82% by heavy pipelining. Literature [19] proposes the use of embedded functional blocks instead of general purpose logic elements to reduce the dynamic power consumption of the designs. Internal routing of embedded system block is much more power efficient than the routing used for general purpose logic.
Clock gating is another commonly used technique for dynamic power reduction [20]. It involves blocking off or gating a clock when the operation of the components it is driving is not required. This is basically ANDing a clock with a control signal. Blocking off the clock not only prevents clock switching in unneeded portions of the design, but also prevents unnecessary toggling of data signals as well.
Dynamic voltage scaling is another technique that can be used to reduce both static and dynamic power consumption in designs [21]. Reducing the supply voltage will reduce the power dissipation in designs but at the same time a T-box based AES design intended to have high throughput and low power usage is discussed in the literature [22]. The T-box method combines the SubBytes, Shiftrows and MixColumns of a round into four table lookups where each column of the table lookup requires 4 kB of memory storage to implement. This method was originally intended for software applications since it can be very efficiently implemented in systems with a 32 bit datapath and large amount available memory. It also has potential to be power and energy efficient in embedded system since it relies on embedded RAM blocks than general purpose logic.

Another technique using low datapath width for AES design is proposed in the literature to reduce the power consumption [23]. It reduces the amount of hardware circuitry in use concurrently. An experiment was conducted on three designs that use one, two and four S-box in parallel and it was observed that they dissipate 137, 158 and 169 mW of dynamic power in Xilinx platform. The use of extra flip flops driven by a phase shifted clock is another power reduction technique for AES design as proposed in the literature [24]. It has been shown that using the proposed power reduction technique an AES design was able to achieve dynamic power consumption of 306.69 mW and 319.17 mW for encryption and decryption respectively while running at 66.66 MHz.

In a Literature [19], it has been commented that a large amount of power consumption in an AES processor is due to its S-box and it is 75% of total power. Generation of glitch in the design is another major reason of dynamic power consumption in the AES design. AES processor based on GF(2^8) was first proposed by Rijandel [1] and from then the researchers were using the same finite field in designing the processor. It is seen that if GF(2^8) is transferred to GF((2^4)^2) using composite field technique and the design of the AES processor is performed using that finite field then complexities of the design is reduced which in turn reduces the power of the AES processor. So in this thesis some basic considerations have been introduced to meet the low power criteria in designing the processor. The design consists of a composite field based architecture of S-box in the finite field GF (2^4)^2 rather than GF (2^8). An optimized S-box is also planned for this architecture by allocating memories dynamically. This design also mitigates the glitches that occur for unequal path lengths from the s-box and inverse s-box. It has been shown that the proposed design consumes much lower power than that of existing ones.
1.3 Contributions of This Thesis

The main objective of this thesis is to design a very low power AES processor. This thesis aims to introduce a new methodology of achieving a very low power processor for AES implementation without compromising the original algorithm so that invulnerability of the AES is maintained.

To meet the goal, the following objectives have been identified:

- To design the S-Box and Inv_S-Box using combinational logic of \( GF(2^4)^2 \).
- To design sub byte transformation by dynamic memory allocation for encryption and decryption respectively.
- To design the proposed architecture of AES processor using Verilog HDL.
- To implement the design using FPGA (field programmable gate array).
- To compare this design with other research works.

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 provides the background information on basic mathematics of AES algorithm which is required for understanding the fundamental operations of some states of AES algorithm. This chapter also presents short overview of the algorithm including its cipher and deciphers parts. Chapter 3 includes the architecture of cipher designs intended to assess the effectiveness of the design strategies. Chapter 4 discusses about the FPGA implementation of the proposed design. The design components are also provided in this chapter. Chapter 5 discusses about the experimental results and discussion on power analysis and measurement of power consumption followed by a comparison with other related works. Finally, Chapter 6 offers suggestions for future work along with concluding remarks.
Chapter 2
Advanced Encryption Standard (AES)

1.1 Introduction
The Rijndael design submitted by Joan Daemen and Vincent Rijmen won the competition and was selected as the new advanced encryption standard due to its robust security properties and simple implementation in both hardware and software [2]. The original Rijndael specification was an iterated block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256 bits. The AES specification is identical to the original except that it limits the block size to 128 bits, retaining the option of specifying a key size of 128, 192 or 256 bits. The use of larger key sizes increases the cryptographic strength of the cipher but requires that a greater number of processing rounds be performed. Currently, 128 bit AES is sufficient for most purposes and is the most commonly used; it will be the focus of the work done in this thesis. Using a key length of 128 bits requires 10 rounds of processing.

2.1 Basic Mathematics for AES
The byte value in AES algorithm is represented as a set of bits (0 or 1) and is represented as the collection of bits separated by comma as \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}. These bytes are interpreted as finite field elements using polynomial representation as:

\[ b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \]

(2.1)

All the operations performed in AES are based on modulo-2 operations. These operations are not as the same operations used in general Number System. The basic operations based on which the entire math of the AES algorithm are Addition and Multiplication. These operations are explained in the subsequent subsections.

2.2 Addition
In modulo-2 additions, two elements are added by adding the coefficients of the corresponding powers in the polynomial [4]. The addition operation here is the XOR operation denoted by the symbol “^”. Subtraction of the polynomials is exactly the same as addition. In the AES algorithm
the finite field algorithm of Galios Field, \( GF(2^n) \) is used for addition of 2 bytes \( \{a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0\} \) and \( \{b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0\} \) is \( \{c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0\} \) where each \( c_i = a_i \land b_i \). For example

If \( A = (179)_{10} \) and \( B = (90)_{10} \) then

\[
A + B = (179)_{10} \land (90)_{10} = (233)_{10}
\]

\[
A - B = (233)_{10}.
\]

Table 2.1 shows the comparison of addition in normal algebra and in \( GF(2^n) \).

<table>
<thead>
<tr>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_1 + p_2 ) (normal algebra)</th>
<th>( p_1 + p_2 ) in ( GF(2^n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x^3 + x + 1 )</td>
<td>( x^3 + x^2 )</td>
<td>( 2x^3 + x^2 + x + 1 )</td>
<td>( x^2 + x + 1 )</td>
</tr>
<tr>
<td>( x^4 + x^2 )</td>
<td>( x^6 + x^2 )</td>
<td>( x^6 + x^4 + 2x^2 )</td>
<td>( x^6 + x^4 )</td>
</tr>
<tr>
<td>( x + 1 )</td>
<td>( x^2 + 1 )</td>
<td>( x^2 + x + 2 )</td>
<td>( x^2 + x )</td>
</tr>
<tr>
<td>( x^3 + x )</td>
<td>( x^7 + x^2 )</td>
<td>( x^3 + x^2 + x + 1 )</td>
<td>( x^3 + x^2 + x + 1 )</td>
</tr>
<tr>
<td>( x^2 + x )</td>
<td>( x^2 + x )</td>
<td>( 2x^2 + 2x )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Numbers can be represented in different forms as shown below:

\[
( x^7 + x^6 + x^4 + x + 1 ) \land ( x^6 + x^4 + x^3 + x )
\]

\[
= x^7 + x^6 + x^5 + x^3 + 1 \hspace{1cm} \text{Polynomial notation}
\]

\[
\{10110011\} \land \{01011010\} = \{11101001\} \hspace{1cm} \text{Binary notation}
\]

\[
\{B3\} \land \{5A\} = \{E9\} \hspace{1cm} \text{Hexadecimal notation}
\]
2.3 Multiplication

In modulo 2 multiplication it is denoted by the symbol „•” and it is the multiplication of the polynomial over an irreducible polynomial of degree-8 based on original AES algorithm. A polynomial is irreducible, if the only divisors of the irreducible polynomial are one and the number itself. The irreducible polynomial is referred to as a Prime Number in Mathematics. For the AES algorithm the irreducible polynomial is of degree 8. It is as follows:

\[ m(x) = x^8 + x^4 + x^3 + x + 1 \]  
\[ ...................................................... (2.2) \]

Equation (2.2) can also be written as \{100011011\}_2 or \{011\}_H in binary and Hexadecimal format respectively.

For example \{72\} •\{18\} = \{dc\}

\[(x^6 + x^5 + x^4 + x) (x^4 + x^3) = x^{10} + x^9 + x^8 + x^5 + x^9 + x^8 + x^7 + x^4 \]

\[= x^{10} + x^7 + x^5 + x^4 \]

\[ x^{10} + x^7 + x^5 + x^4 \text{ modulo } (x^8 + x^4 + x^3 + x + 1) \]

\[= x^7 + x^6 + x^4 + x^3 + x^2 \]

The modular multiplication over the polynomial \(m(x)\) ensures that the resultant product of all multiplication has the degree less than eight and can be represented by a byte. Unlike the addition the multiplication cannot be comprehended by a single operation.

2.4 Finite Field and Composite Field

This section introduces the definition of finite fields and composite fields followed by the basic AES mathematical representations and operations over finite field \(GF(2^8)\) and composite field \(GF(2^4)^2\). Arithmetic in a finite field is different from standard integer arithmetic. There are a limited number of elements in the finite field; all operations performed in the finite field result in an element within that field. While each finite field is itself not infinite, there are infinitely many different finite fields; their number of elements (which is also called cardinal) is necessarily of the form \(p^n\) where \(p\) is a prime number and \(n\) is a positive integer, and two finite fields of the same size are isomorphic. The prime \(p\) is called the characteristic of the field, and the positive integer \(n\)
is called the dimension of the field over its prime field. Finite fields are used in a variety of applications, including in classical coding theory in linear block codes such as BCH and RS and in cryptography algorithms such as the Rijndael encryption algorithm (the main focus of this thesis). The composite fields approach the GF(2ⁿ) is formed by farther subdividing i.e. GF(2ᵐⁿ). So the operational complexity is much reduced and the CPU speed increased significantly. This thesis uses the composite fields approach in the first design of the two designs. The next two sub Sections describes the AES arithmetic over these two fields.

2.4.1 AES Arithmetic over Finite Field $GF(2)^8$

The AES algorithm proposed by Rijndael uses a characteristic ,2” finite field with „8” terms, which can also be called the Galois field GF(2⁸). It employs the following irreducible polynomial for multiplication:

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

For example, \{53\} • \{CA\} = \{01\} in Rijndael's field because

$$(x^6 + x^4 + x + 1) (x^7 + x^6 + x^3 + x) =$$

$$x^{13} + x^{12} + x^9 + x^7 + x^{11} + x^{10} + x^7 + x^5 + x^8 + x^7 + x^4 + x^2 + x^6 + x^3 + x =$$

$$x^{13} + x^{12} + x^9 + x^{11} + x^{10} + x^5 + x^8 + x^4 + x^2 + x^6 + x^3 + x =$$

$$x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x.$$ 

and

$$x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x \mod x^8 + x^4 + x^3 + x + 1 =$$

$$1111101111110 \mod 100011011 = 1.$$ 

A computer program can be initiated for the above polynomial multiplication is given as follows;

1111101111110 (mod) 100011011

^100011011

1110000011110

^100011011
The elements \{53\} and \{CA\} happen to be multiplicative inverses of one another since their product is \(-1\). The verilog HDL code for the above three operations in GF\(2^8\) is provided in the Appendix C.

### 2.4.2 AES Arithmetic over Composite Field GF\((2^4)^2\)

For the AES algorithm the substitution byte called s-box is designed by GF\(2^8\) means 1 byte representation of each elements of 256 byte values. So the multiplicative inverse computation will be done by decomposing the more complex GF\(2^8\) to lower order fields of GF\(2^1\), GF\(2^2\) and GF\((2^3)^2\). In order to accomplish the above, the following irreducible polynomials are used. [25]

\[
\begin{align*}
\text{GF}(2^2) & \in \text{GF}(2) : x^2 + x + 1 \\
\text{GF}(2^2)^2 & \in \text{GF}(2^2) : x^2 + x + \varphi \quad \text{-------------------------(2.3)} \\
\text{GF}(2^2)^3 & \in \text{GF}(2^3)^2 : x^2 + x + \lambda \\
\end{align*}
\]

Where \(\varphi = \{10\}_2\) and \(\lambda = \{1100\}_2\).

The specific composite field used in this thesis is GF\((2^4)^2\) which is isomorphic to field GF\(2^8\). Taking field GF\(2^8\) as a quadratic extension of the field GF\(2^4\) and an element \(a \in \text{GF}(2^8)\) is represented as a linear polynomial with coefficients in GF\(2^4\).

**A. Notation**

Wolkerstorfer j. introduced a two-term polynomial in literature [9], which is the representation of GF\((2^4)^2\) used in the thesis.

\[
a = a_h x + a_l , \ a \in \text{GF}(2^8), \ a_h, \ a_l \in \text{GF}(2^4) \quad \text{-------------------------(2.4))}
\]

The two-term polynomial \(a_h x + a_l\) is an isomorphic representation of \(a\). Hence, all mathematical operations applied to elements of GF\(2^8\) can also be computed in this representation.
B. Addition

Adding the corresponding coefficients in GF(2^4) is as follows.

\[(a_h x+ a_l) \oplus (b_h x+ b_l) = (a_h \oplus b_h)x+(a_i \oplus b_i) \]  \hspace{1cm} (2.5)

C. Multiplication

Two irreducible polynomials are needed for the two-term polynomial multiplication: \(n(x)\) (Equations (2.6)) and \(m(x)\) (Equations (2.7)).

\[n(x) = x^2+{1}x+{E} \]  \hspace{1cm} (\{E\} denotes “1110”)  \hspace{1cm} (2.6)
\[m(x) = x^4+x+1 \]  \hspace{1cm} (2.7)

Equation (2.6) is used to reduce the result to a two-term polynomial. The coefficients of \(n(x)\) are written in hexadecimal notation which are elements in GF(2^4). Multiplication of two-term polynomials is denoted by “\(\otimes\)”. Normal polynomials multiplication is denoted by “\(\times\)”. Multiplying two two-term polynomials, followed by a modular reduction over \(n(x)\), is described by Equation (2.8).

\[(a_h x+ a_l) \otimes (b_h x+ b_l) = ((a_h x+ a_l)\times(b_h x+ b_l)) \mod n(x) \]  \hspace{1cm} (2.8)

Equation (2.7) is used to ensure that, the result of multiplication in subfield GF(2^4) (Equation (2.9)), where \((a'(x), b'(x)) \in GF(2^4)\) is an element of GF(2^4).

\[a'(x) \otimes b'(x) = (a'(x)\times b'(x)) \mod m(x) \]  \hspace{1cm} (2.9)

These two irreducible polynomials \(n(x)\) and \(m(x)\) are chosen by Wolkerstorfer j. to optimize the arithmetic.

D. Multiplicative Inverses

A multiplication of a two-term polynomial with its inverse yields the 1-element of the field GF((2^4)^2)

\[(a_h x+a_l) \otimes (a'_h x+a'_l) = \{0\}x+\{1\} \]  \hspace{1cm} (2.10)

where \(a_h, a_l, a'_h, a'_l \in GF(2^4)\).

\[(a_h x+a_l)-1 = (a'_h x+a'_l) = (a_h \otimes d) x+ (a_h \oplus a_l) \otimes d \]  \hspace{1cm} (2.11)
Where \( d = (a_1^2 \otimes \{E\}) \oplus (a_h \otimes a_l) \oplus a_1^2 - 1 = ((a_1^2 \otimes \{e\}) \oplus ((a_h \oplus a_l) \otimes a_l))^{-1} \)

(\( \oplus \) is addition in GF\((2^4)\); \( \otimes \) is multiplication in GF\((2^4)\)). This multiplicative inversion equation is proposed by Wolkerstorfer j. [9]. Reconfiguration of this equation can provide good quality for sub-pipelining.

### 2.5 Overview of AES Algorithm

The AES algorithm operates on 128 bits of data and generates 128 bits of output. The length of the key used to encrypt this input data can be 128, 192 or 256 bits. This thesis will deal with only the 128 bits of key. As this is private key cipher it uses just one key of 128 bits which is used both for Encryption and Decryption. \( N_b \) which defines the number of columns of 32 bits of input data is \( N_b = 128/32 = 4 \). Similarly \( N_k \) which defines the number of columns of 32 bits of key is, \( N_k = 128/32 = 4 \). For key length of 192 and 256 the values of \( N_k \) will be 192/32= 6 and 256/32= 8 respectively. The number of rounds \( N_r = 10 \) when \( N_k = 4 \) and changes to 12 and 14 for \( N_k = 6 \) and \( N_k = 8 \). As we will deal with the key length of 128 bits for the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by \( N_r \), where \( N_r = 10 \) when \( N_k = 4 \), \( N_r = 12 \) when \( N_k = 6 \), and \( N_r = 14 \) when \( N_k = 8 \). Therefore, \( N_k = 4, N_r = 10 \). The AES algorithm basically consists of four byte oriented transformation. These transformations are:

- **a)** Byte substitution using s-box table (S-box)
- **b)** Shifting rows of the state array using different offsets (Row transformation)
- **c)** Mixing the data within each column of the state array (Mixing columns)
- **d)** Adding a round key to the state (Add round key)

Figure 2.1 presents the flowchart of the AES Encryption process.
2.5.1 Sub Bytes

Sub byte transformation is the first block of encryption and decryption process where every byte of the input is replaced by the value of substitution table i.e. sub byte. Every round begins with the “SubBytes” transformation. It is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). Figure 2.2 shows the sub bytes transformation. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation.
The affine transformation is performed by the following equation:

\[ b'_i = b_i \oplus b_{(i+4)\text{mod} 8} \oplus b_{(i+5)\text{mod} 8} \oplus b_{(i+6)\text{mod} 8} \oplus b_{(i+7)\text{mod} 8} \oplus c \quad (2.3) \]

Where \( b' \), \( b \), and \( c \) are 8 bit arrays and \( c = \{01100011\} \). The pseudo code for affine transform can be written as follows:

```java
public static boolean[] affineX (boolean[] bprime, boolean[] b, boolean[] c) {
    for (int j=0; j<8; j++) {
        bprime[j]  = b[j] ^ b[(j +4)%8]
        bprime[j] ^= b[(j+5)%8]
        bprime[j] ^= b[(j+6)%8]
        bprime[j] ^= b[(j+7)%8]
        bprime[j] ^= c[j]
    }
    return bprime
}
```

The S-Boxes used in the SubBytes function are created in such a way that they are invertible for use as Inverse S-Boxes in the InvSubBytes function. This is possible because the S-Box maps each input to exactly one output. Like the forward S-box, the inverse S-Box is comprised of two functions, the inverse of the affine transformation is followed by multiplicative inversion in \( GF(2^8) \).
2.5.2 Shift Rows

This transformation circularly shifts each row of the state to the left in encryption or to the right on decryption. The top row of the state is denoted as row (0) and the bottom row is denoted as row (3). The shift offset of each row corresponds to the row number.

A. Shiftrows – Each row of the state is left shifted cyclically a certain number of bytes. Performs i-byte circular left shift to row (i) (i = 0,1,2,3). Figure 2.3 illustrates the shiftrows operation.

![Shift Rows Transformation](image)

Figure 2.3 Shift Rows Transformation

A. Inverse Shiftrows – The inverse of ShiftRows, „InvShiftRows” is used in the decryption function of the cipher. Inverse Shift rows functions exactly the same as ShiftRows, only in the opposite direction. The first row is not rotated, while the second, third and fourth rows are rotated right by one, two and three bytes respectively. InvShiftRows” operation is depicted in Figure 2.4.

![Inverse Shift Rows](image)

Figure 2.4 Inverse Shift Rows
2.5.3 MixColumns

This transformation treats each column of the state as a four-term polynomial over GF($2^8$) and transforms each column to a new one by multiplying it with a constant polynomial $a(x) = \{03\} x^3 + \{01\} x^2 + \{01\} x + \{02\}$ modulo $x^4 + 1$. The inverse mixcolumns operation is a multiplication of each column with $b(x) = a^{-1}(x) = \{0B\} x^3 + \{0D\} x^2 + \{09\} x + \{0E\}$ modulo $x^4 + 1$ [4]. Figure 2.5 shows the MixColumn Operation.

![MixColumn Diagram](image)

**Figure 2.5 MixColumn Transformations**

**A. MixColumns** – Left multiplies the state with a mixcolumns matrix. Mixcolumns transformation gives each byte of a column a new value based on all four bytes in that column. In matrix form, the mixcolumns can be expressed as:

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

**B. InvMixColumns** - is used in the decryption function of the cipher. It works exactly as MixColumns does, except that it uses the inverse of MixColumns transformation matrix in its matrix multiplication, shown in the following equation.

$$
\begin{bmatrix}
S'_{0,c} \\
S'_{1,c} \\
S'_{2,c} \\
S'_{3,c} \\
\end{bmatrix}
= 
\begin{bmatrix}
0E & 0B & 0D & 09 \\
09 & 0E & 0B & 0D \\
0D & 09 & 0E & 0B \\
0B & 0D & 09 & 0E \\
\end{bmatrix}
\begin{bmatrix}
S_{0,c} \\
S_{1,c} \\
S_{2,c} \\
S_{3,c} \\
\end{bmatrix}
$$
2.5.4 AddRoundKey

AddRoundKey is the final cipher function, and is used to mix key information in with the data that is being operated on. It is a straightforward cipher function comprised only of a simple bitwise XOR between the current state and the current round key which was expanded from the input key. Both encryption and decryption function of the cipher use AddRoundKey. Figure 2.6 shows the AddroundKey process.

![AddRoundKey Process Diagram]

Figure 2.6 AddRoundKey Process

2.5.5 Key Expansion

The key expander produces the key schedule one four byte word at time, with each word based on the previous word and the word from four word at a time positions back. Four word positions back happens to be exactly one key length, so stating that the word from four word positions back used is the same as stating that the corresponding word from the previous round key is used. Each word is commonly given an index i such that 0 ≤ i < 44, with words 1 though 4 corresponding to the original input key. For most words, the new word is calculated such that it is simply an XOR between the previous word and the corresponding word from the previous round key, but every word that has an index which is a multiple of 4 is given special treatment. In this case, the word has three additional functions applied to it, RotWord, SubWord, and AddRcon. RotWord is very similar to the ShiftRows function used in the cipher; it is a circular shifting, or rotation, of one byte applied to the word. Similarly, SubWord is nearly identical to the cipher’s SubBytes function. The only difference is that SubWord is only applied to a four byte word, while
SubBytes was used on a 16 byte State. AddRcon is the XORing of the word with a Round Constant. The Rcon Values for each round all have different Most Significant Bytes (MSBs) which are given in hex values.

\[
\begin{align*}
&\text{assign } r_{c}[1] = 32'H01000000; \\
&\text{assign } r_{c}[2] = 32'H02000000; \\
&\text{assign } r_{c}[3] = 32'H04000000; \\
&\text{assign } r_{c}[4] = 32'H08000000; \\
&\text{assign } r_{c}[5] = 32'H10000000; \\
&\text{assign } r_{c}[6] = 32'H20000000; \\
&\text{assign } r_{c}[7] = 32'H40000000; \\
&\text{assign } r_{c}[8] = 32'H80000000; \\
&\text{assign } r_{c}[9] = 32'H1B000000; \\
&\text{assign } r_{c}[10] = 32'H36000000; \\
\end{align*}
\]

The MSB for the Rcon value for round 1 is 01, and every Rcon MSB after that is simply the previous value multiplied by 02 in GF(2^8) using \( m(x) \) as the modulus. The least significant three bytes of each Rcon word are always 00. A diagram depicting the operation of Key Expansion can be seen in Figure 2.7.

![Figure 2.7 Key Expansion modules](image-url)
2.6 Overview of FPGA

Field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. It contains up to thousands of gates. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed. There are various vendor manufacturers for different types of FPGA chip such as Altera, Xilinx, Lattice Semiconductor, Actel, Quick Logic, Cypress Semiconductor, Atmel, Achronix Semiconductor etc. Among them Altera and Xilinx are the most famous FPGA companies since both of the companies have lot of varieties of FPGA device from small number of gate counts to higher number of gate counts. However Altera devices offer the general benefits of PLDs as innovative architectures, advanced process technologies, state-of-the-art development tools, and a wide selection of mega function. The common advantages of Altera devices include: High performance, High-density logic integration, Cost-effectiveness, Short development cycles with the Quartus II software, Mega Core functions, Benefits of in-system programming. In this thesis the used FPGA device is Altera provided EP2C35F672C6 from Cyclone II family.

2.6.1 FPGA Cyclone II Device

This sub section presents some basic information about this device which will help for development of the proposed processors with this cyclone device. Altera’s low-cost CycloneTM II FPGA family is based on a 1.2-V, 90-nm SRAM process with densities over 68K logic elements (LEs) and up to 1.1 Mbits of embedded RAM. With features like embedded 18 × 18 multipliers to support high-performance DSP applications, phase-locked loops (PLLs) for system clock management, and high-speed external memory interface support for SRAM and DRAM devices, Cyclone II devices are a cost-effective solution for high-volume applications. Cyclone II devices support differential and single-ended I/O standards, including LVDS at data rates up to 805 megabits per second (Mbps) for the receiver and 640 Mbps for the transmitter, and 64-bit, 66-MHz PCI and PCI-X for interfacing with processors and ASSP and ASIC devices. Altera also offers low-cost serial configuration devices to configure Cyclone II devices. The Cyclone II FPGA family offers commercial grade, industrial grade, and lead-free devices.
The Cyclone II device family offers the following features:

- High-density architecture with 4,608 to 68,416 LEs
- M4K embedded memory blocks
- Embedded multipliers
- Advanced I/O support
- Flexible clock management circuitry
- Device configuration
- Intellectual property

2.6.2 Development Tool Quartus II

The propose AES processors are designed using Quartus II EDA tool (provided by Altera Company) which provides Graphical User Interface (GUI) to download the digital design AES into the Cyclone II FPGA. Quartus II software provides a simple, automated mechanism to allow designers to obtain the best performance for their designs. This software provides the way to design the solution through Verilog HDL and compile the design to ensure the workability and efficiency logically. The tool Programmer allows using files generated by the Compiler to program and/or configuring all devices supported by the Quartus II software. Programmer and supported programming hardware tool to easily program or configure a working device in minutes. After a successful compilation, download configuration data into a device through the, ByteBlaster or USB-Blaster communications cables, or through the Altera Programming Unit (APU). The program or configure devices can be in Passive Serial mode, Active Serial Programming mode, JTAG mode, or In-Socket Programming mode.

- **Program an Altera Device:** When the design is ready to program or configure a device, it needs to open the Programmer and create a Chain Description File (.cdf) that stores device name, device order, and programming and hardware setup information. CDFs can be used to program or configure one or more devices in a JTAG chain or a Passive Serial chain.

- **Compiling mode:** The Quartus II Compiler consists of a set of independent modules that check the design for errors, synthesize the logic, fit the design into an Altera device, and generate output files for simulation, timing analysis, software building, and device programming. The basic Compiler consists of the Analysis & Synthesis, Fitter, Assembler, and Timing Analyzer modules. Each of the Compiler modules can be run
individually or together from the Quartus II user interface. Alternatively, these modules can be run independently with the appropriate command line executable.

- **Compile the Design:** The Compiler automatically locates and uses all non-design files associated with the design, such as Include Files (.inc) containing AHDL Function Prototype Statements; Memory Initialization Files (.mif) or Hexadecimal Intel-format Files (.hex) containing the initial content of memories; as well as Quartus II Project Files (.qpf) and Quartus II Settings Files (.qsf) containing project and setting information. During compilation, the Compiler generates information, warning, and error messages that appear automatically in the Messages window.

- **Simulation mode:** Simulation allows testing a design thoroughly to ensure that it responds correctly in every possible situation before configuring a device. Depending on the type of information need, functional or timing simulation can be performed with the Simulator. Functional simulation tests only the logical operation of a design by simulating the behavior of flattened netlist extracted from the design files, while timing simulation uses a fully compiled netlist containing timing information to test both the logical operation and the worst-case timing for the design in the target device. Before running a simulation, must be specify input vectors as the stimuli for the Quartus II Simulator. The Simulator uses these input vectors to simulate the output signals that a programmed device would produce under the same conditions. The Simulator supports input vector stimuli in the form of a Vector Waveform File (.vWF), Vector Table Output File (.tbl), Power Input File (.pwf), or a Quartus II generated Vector File (.vec) or Simulator Channel File (.scf).
Chapter 3
Low Power Design Considerations

3.1 Introduction
This chapter presents the design strategies of AES processor to achieve the low power. A vast study on AES shows that the total power consumption of AES processor can be sub divided into two parts: thermal dynamic power consumption and thermal static power consumption. The static power consumption is fixed for a given device but the dynamic power consumption is variable with respect to the transition of each logic element within a clock cycle. It will be discussed in Section 3.2 that the power consumption mainly occurs due to logic elements and memory block of a chip. The AES algorithm comprised of four transformations has fixed power consumption in static mode which depends on voltage level and working temperature of the device where it is implemented. It is shown that the most power consumption part of the algorithm is its sub byte transformation. 75% of total power is consumed by the s-box probably [19]. So the main focus of this thesis is to minimize the power within the s-box. The details are given in the following sub sections of this chapter.

3.2 Power analysis
Now a day’s power is a key design parameter in integrated circuits that could ultimately determine the composition of a chip. But in some cases the sub threshold current will play a more significant role affecting the power budget of future portable devices. The power dissipation model in a typical low power digital SoC (system on chip) is based on logic and memory blocks of a chip. For a chip the total power dissipation is given by Equation 3.1. The Logic blocks are subdivided into CPU/DSP cores and peripheral logic, while memory blocks are categorized as either SRAM and embedded DRAM [28].

The total chip power, $P_{total}$ is:

$$P_{total} = \Sigma P_{logic} + \Sigma P_{memory} \quad --------------------------------- (3.1)$$

Power dissipation in logic blocks, $P_{logic}$, consists of both dynamic (switching) and static (standby) power and can be formulated as:
$$P_{\text{logic}} = P_{\text{dynamic}} + P_{\text{static}} \tag{3.2}$$

Where

$$P_{\text{static}} = V_{\text{DD}} I_{\text{leakage}}$$

Where $I_{\text{leakage}} = \text{leakage current of the chip}$

And

$$P_{\text{dynamic}} = \alpha_{\text{logic}} C_{\text{logic}} V_{\text{DD}}^2 f \tag{3.3}$$

Where

$f = \text{operating clock frequency}$;

$\alpha_{\text{logic}} = \text{activity factor of logic blocks}$;

$C_{\text{logic}} = \text{total capacitance of logic blocks}$;

$V_{\text{DD}} = \text{power supply voltage}.$

The main power dissipation in memory blocks (i.e., SRAM) is due to column bit lines switching, particularly during the write operation where the bit lines are pulled low. The power dissipated in the memory blocks, $P_{\text{memory}}$ can then be written as:

$$P_{\text{memory}} = \alpha_{\text{memory}} C_{\text{memory}} V_{\text{DD}}^2 \tag{3.4}$$

Where

$\alpha_{\text{memory}} = \text{activity of the memory blocks}$;

$C_{\text{memory}} = \text{total capacitance of all memory cells combined}$.

The power of a specific design for a memory depends on the architecture/segmentation of the memory [29]. For example, an SRAM with a 64-bit long output alone with no row or column segmentation of the array will dissipate more power than that of a SRAM with a 4-bit long output alone with a segmentation of row-column. The capacitance ($C_{\text{memory}}$) of a cell can be estimated by considering the diffusion capacitance of a cell, the contact capacitance and the metal line capacitance. This capacitance is referred to as $C_d$ below. The total capacitance of a RAM block is:

$$C_{\text{memory}} = 2^M 2^N C_d$$
Where

\[ M = \text{number of column address bits} \]
\[ N = \text{number of row address bits} \]

However, not all this capacitance is switched on each cycle. In fact, if \( 2M \) bits are written on every other cycle, then \( M \) bits are switched on average. Therefore, \( \alpha_{\text{memory}} \) is approximately \( M/2N \), which is the ratio of accessed bits to total bits in the memory. For example, for a 16Mbit memory, let \( M=16 \) and \( N=12 \); then, \( \alpha = 0.66 \) [29]. Understanding that many factors affecting power consumption allows us to apply the power play power analyzer most effectively. The architecture works only on the construction of s-box and all the other rounds of the AES remain same. The next section: Section 3.3 will discuss about this architecture.

### 3.3 Design

The design is based on composite field based architecture of s-box. The most power consuming operation in s-box [30] is finding the multiplicative inverse of 256 values in GF \( (2^8) \) followed by the affine transform (AT). The proposed design introduces a new technique of finding multiplicative inverse by decomposing of GF\( (2^8) \) into GF\( (2^4)^2 \). The decomposition process and multiplicative inverse process are shown in the following sub sections.

#### 3.3.1 Design Considerations

This design proposes the composite field based design of s-box in GF \( (2^4)^2 \) where the original s-box of AES algorithm is in GF\( (2^8) \). This design also proposes a method of filtering out the glitches that occurs for unequal path lengths in the processor. So this design works for lowering the total power consumption of the AES processor by the decomposition of GF\( (2^8) \) in composite fields. The main power consumption occurs within the s-box operation even though it is used by table look up. Look up table based execution of s-box consumes more power due to extract values from the static RAM. The reason is that the RAM has a fixed time to operate. So this thesis firstly considers the memory allocation of s-box dynamically rather than that of its LUT based realization. Secondly this thesis achieves low power by mitigating the glitches that occurs for unequal path lengths in the processor with the help of balancing path lengths by buffer insertion as a processing step. Some of the researchers show that if the s-box is implemented in GF\( (2^8) \) which require more registers which implies more areas and thus require more memory power. Also this type of design requires more transitions in logic levels which equal to the more dynamic power consumption within the s-box. The detail of this type of power consumptions is explained
in Section 3.2. The s-box construction is performed by the decomposition of GF(2^8) in composite fields. So the design considerations are given below.

- Designing the s-box and inverse s-box of the AES processor in GF (2^3)^2. Here the memory allocations are dynamic.
- Implementing look up table only for two operations which are multiplicative inverse and multiplication by constant.
- Mitigating the glitches that occur for unequal path lengths from the s-box and inverse s-box.
- reducing the unwanted paths or glitches from the datapath pipelined structure of 3-multipliers is designed.

Figure 3.1 describes the concept of mitigation of the glitches. This is one of the considerations of this thesis. Here if we consider the Figure 3.1 which consists of two AND gates and an XOR gate using no buffering then it costs more power than that of buffering. The final output of the logic in Figure 3.1 can be found by 4-inputs to the AND1 and AND2. If we insert buffer after every AND gate than that will be glitch free.

![Figure 3.1 Minimum power-mapping for Gates.](image)

The next Sub Sections explain the above two considerations in details.

3.3.2 Composite field Based S-Box Architecture

This Section illustrates the combinational logic based s-box architectures and the steps involved in constructing the multiplicative inverse module for the s-box. Since both the SubByte and Inverse SubByte transformations are similar other than their operations which involve the Affine
Transformation and its inverse, therefore only the implementation of the SubByte operation will be considered in this thesis. The multiplicative inverse computation will first be covered and the affine transformation will then follow to complete the methodology involved for constructing the s-box for the SubByte operation. The complete s-box operation is shown in the Figure 3.2 and its legends are shown in Figure 3.3.

Figure 3.2 Complete s-box creation in GF(2^4)[25].

Figure 3.3 Legends for the building blocks within the s-box creation

Now this section explains each and every block of Figure 3.2 in details. Let starts with isomorphic mapping to composite fields.
A. Mapping Function $\delta$ and $\delta^{-1}$

Computation of the multiplicative inverse in composite fields cannot be directly applied to an element which is based on $\text{GF}(2^8)$. That element has to be mapped to its composite field representation via an isomorphic mapping, $\delta$. Likewise, after performing the multiplicative inversion, the result will also have to be mapped back from its composite field representation to its equivalent in $\text{GF}(2^8)$ via the inverse isomorphic mapping, $\delta^{-1}$. Both $\delta$ and $\delta^{-1}$ can be represented as an 8x8 matrix. Let $a$ be the element in $\text{GF}(2^8)$, then the isomorphic mappings and its inverse can be written as $\delta*a$ and $\delta^{-1}*a$, which are shown in pseudo code as follows:

$$\delta*a = \begin{align*}
a[0] &= a[0]^a[1]^a[6]; \\
\end{align*}$$

And

$$\delta^{-1}*a = \begin{align*}
\end{align*}$$

Where $\wedge$ denotes the bit wise Xor operation.

B. Squaring in $\text{GF}(2^4)$

From Literature [31], it is stated that any arbitrary polynomial can be represented as $bx + c$, given an irreducible polynomial of $x^2 + Ax + B$. Thus, element in $\text{GF}(2^8)$ may be represented as $bx + c$
where b is the most significant nibble while c is the least significant nibble. For an example, if \( a = \{1011\}_2 \), it can be represented as \( \{10\}_2 x + \{11\}_2 \), where \( a_H = b = \{10\}_2 \) and \( a_L = c = \{11\}_2 \). \( a_H \) and \( a_L \) can be further decomposed to \( \{1\}_2 x + \{0\}_2 \) and \( \{1\}_2 x + \{1\}_2 \) respectively. Let \( k = a \cdot a = a^2 \), where \( \cdot \) means multiplication in \( GF(2^4) \). So \( k \) and \( a \) are the elements of \( GF(2^4) \) and each has 4 bits in binary therefore \( \{k_3, k_2, k_1, k_0\} \) and \( \{a_3, a_2, a_1, a_0\} \).

\[
\begin{align*}
k &= k_Hx + k_L = (a_Hx + a_L) \cdot (a_Hx + a_L) = (a_Hx + a_L)^2 \\
k &= a_H^2 x^2 + (a_Ha_L)x + (a_Ha_L)x + a_L^2 = a_H^2 x^2 + a_L^2 
\end{align*}
\]  
\hspace{1cm} --------------------------(3.5)

From Sub Section 2.4.2 the \( x^2 \) term can be modulo reduction by \( x^2 + x + \varphi \). Putting the value of \( x^2 = x + \varphi \) from Equation 3.5 we get

\[
\begin{align*}
k &= a_H^2 (x + \varphi) + a_L^2 = a_H^2 x + (a_H^2 \varphi + a_L^2) 
\end{align*}
\]  
\hspace{1cm} --------------------------(3.6) therefore

\[
\begin{align*}
k_H &= a_H^2 \\
and \\
k_L &= (a_H^2 \varphi + a_L^2)
\end{align*}
\]

Equation 3.6 clearly shows that the \( GF(2^4) \) decompose to \( GF(2^2) \). Using the irreducible polynomial from Sub Section 2.4.2 the further decomposition is possible in which all the 4-bit values of \( k \) can be determined easily. If we consider \( a=p \) and \( k=N \) then the Equation 3.6 can be written as the following logic function.

\[
\begin{align*}
\end{align*}
\]

Figure 3.4 presents the hardware realization of squaring in \( GF(2^4) \) of the Equation 3.6.
Figure 3.4 Hardware implementation of squaring in GF(2^4).

C. Multiplication with constant, $\lambda$

Consider $k = a\lambda$, where $k$ and $a$ each are 4-bits and $\lambda = \{1100\}_2$ is the elements of GF(2^4).

\[
k = k_Hx + k_L = (a_Hx + a_L) \cdot (1100) = (a_Hx + a_L) \cdot (\lambda_Hx + \lambda_L)
\]

\[
k = a_H\lambda_H x^2 + a_L\lambda_H x \quad \text{as} \quad \lambda_L = \{00\}
\]

For modulo reduction putting $x^2 = x + \varphi$ [25] in the above

\[
k = a_H\lambda_H (x + \varphi) + a_L\lambda_H x
\]

\[
k = ((a_H\lambda_H + a_L\lambda_H) x + \varphi a_L\lambda_H) \in GF(2^2)
\]

therefore

\[
k_H = (a_H\lambda_H + a_L\lambda_H)
\]

and

\[
k_L = \varphi a_L\lambda_H
\]

\[
k_H = (a_H\lambda_H + a_L\lambda_H)
\]

\[
= (a_3 a_2) (\{11\}_2) + (a_1 a_0) (\{11\}_2)
\]

\[
= (a_3 x + a_2) (x + 1) + (a_1 x + a_0) (x + 1)
\]

\[
k_3 x, k_2 = a_3 x^2 + (a_3 + a_2) x + a_2 + a_1 x^2 + (a_1 + a_0) x + a_0 \quad \text{---------(3.7)}
\]

Equation 3.7 can be further reduced to GF(2) by putting $x^2 = x + 1$

\[
k_3 x, k_2 = (a_2 + a_0) x + (a_3 + a_2 + a_1 + a_0) \quad \text{---------(3.8)}
\]

Go after the same procedure like $k_H$, $k_L$ can be determined.

\[
k_1 x + k_0 = (a_3) x + (a_2) \quad \text{-------------------(3.9)}
\]

From Equations (3.8) and (3.9) combined, the formula for computing multiplication with constant $\lambda$ is shown below

\[
k[0] = a[2];
\]

\[
k[1] = a[3];
\]

\[
\]

\[
k[3] = a[0]^a[2];
\]

The hardware realization of constant multiplier $\lambda = \{1100\}_2$ in GF(2^4) is shown below
D. Multiplication in GF($2^4$)

Let $k = ab$ where $k = k_3 k_2 k_1 k_0$ and $a = a_3 a_2 a_1 a_0$, $b = b_3 b_2 b_1 b_0$ this multiplication in GF($2^4$) is presented below:

$$k = (k_H x + k_L) = (a_H x + a_L) \cdot (b_H x + b_L)$$

$$= (a_H b_H) x^2 + (b_H a_L + a_H b_L) x + a_L b_L$$

Substituting the $x^2$ term with $x^2 = x + \varphi$ yields the following.

$$k = (k_H x + k_L) = (a_H b_H) (x + \varphi) + (b_H a_L + a_H b_L) x + a_L b_L$$

$$(k_H x + k_L) = (a_H b_H + b_H a_L + a_H b_L) x + a_H b_H \varphi + a_L b_L \in \text{GF}(2^2) \quad --------(3.10)$$

Equation (3.10) is in the form GF($2^2$). It can be observed that there exists addition and multiplication operations in GF($2^2$). Addition in GF($2^2$) is bitwise XOR operation. Multiplication in GF($2^2$), on the other hand, requires decomposition to GF(2) to be implemented in hardware. Also, the expression would be too complex if Equation (3.10) were to be broken down to GF(2). Thus, the formula for multiplication in GF($2^2$) and constant $\varphi$ will be derived instead in following sub sections. Figure 3.6 below shows the hardware implementation for multiplication in GF($2^4$).
30

Figure 3.6 Hardware Implementation of Multiplication in GF(2^4)

D.1. Multiplication Operation in GF(2^2)

Let \( k = ab \) where \( k = k_1 k_0 \) and \( a = a_1 a_0, b = b_1 b_0 \) this multiplication in GF(2^2) is presented below;

\[
\begin{align*}
   k &= (k_1 \times k_0) = (a_1 x + a_0) \cdot (b_1 x + b_0) \\
   &= (a_1 b_1) x^2 + (b_1 a_0 + a_1 b_0) x + a_0 b_0 \\
\end{align*}
\]

The \( x^2 \) term can be substituted with \( x^2 = x + 1 \) to yield the new expression below.

\[
\begin{align*}
   k &= (k_1 x + k_0) = (a_1 b_1) (x + 1) + (b_1 a_0 + a_1 b_0) x + a_0 b_0 \\
   &= (a_1 b_1 + b_1 a_0 + a_1 b_0) x + a_1 b_1 + a_0 b_0 \in GF(2) \quad (3.11) \\
\end{align*}
\]

The Equation 3.11 can now be implemented in hardware as multiplication in GF(2) involves only the use of AND gates and XOR gates. The formula for computing multiplication in GF(2) is as follows. The hardware implementation of 2-multiplication is shown in Figure 3.7.

\[
\begin{align*}
   k[1] &= (a[1] \& b[1]) \& (a[0] \& b[1]) \& (a[1] \& b[0]); \\
   k[0] &= (a[1] \& b[1]) \& (a[0] \& b[0]); \\
\end{align*}
\]
D.2. Multiplication with constant $\Phi$ in GF($2^2$)

Let $k = a\phi$, where $k = \{k_1 \ k_0\}_2$, $q = \{a_1 \ a_0\}_2$ and $\phi = \{10\}_2$ are elements of GF($2^2$).

\[
k = (k_1 \ k_0) = (a_1 x + a_0) \cdot (x + 0)
= (a_1) x^2 + (a_0) x
\]

Substitute the $x^2$ term with $x^2 = x + 1$, yield the expression below.

\[
k = (k_1 x + k_0) = (a_1) x + (a_0) x
\]

$(k_1 x + k_0) = (a_1 \ a_0) x + a_1 \in$ GF(2) \hspace{1cm} \text{So}

\[
k[1] = a[1] \ ^{\wedge} \ a[0];
\]

\[
k[0] = a[1];
\]

The complete hardware realization of multiplication in GF($2^4$) is shown in Figure 3.8.
Figure 3.8 Hardware Implementation of Multiplication in GF ($2^4$)
E. Multiplicative Inverse in GF(2^4)

The multiplicative inverse in the AES algorithm obtains by the formula $a \cdot a^{-1} = 1$. The mathematical formulae of finding the inverse of $K(x) = k_h + k_l$ modulo $M(x) = x^2 + x + \lambda$ is equivalent to finding polynomials $A(x)$ and $B(x)$ satisfying the following Euclidean equation:

$$A(x) M(x) + B(x) K(x) = 1 \quad \text{(3.12)}$$

Where $B(x)$ in Equation 3.12 is the inverse of $K(x)$ mod $M(x)$ A(X) is the given polynomial. Such $A(x)$ and $B(x)$ can be found by using the extended Euclidean algorithm for an iteration. First, rewrite the Equation 3.12 in the form of

$$M(x) = Q(x) K(x) + R(x) \quad \text{(3.13)}$$

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

$$Q(x) = (k_h^{-1}x + (1 + k_h^{-1}k_l)k_h^{-1} \quad \text{(3.14)}$$

and

$$R(x) = \lambda + (1 + k_h^{-1}k_l)k_h^{-1}k_l \quad \text{(3.15)}$$

Substituting (3.14) and (3.15) into (3.13) and multiplying $k_h^2$ to both sides of the equation, it follows that

$$k_h^2 M(x) = (k_h x + (k_h + k_l)Kx + (k_h^2 \lambda + k_h k_l + k_l) \quad \text{(3.16)}$$

Multiplying both sides of 3.16with $\Phi = (k_h^2 \lambda + k_h k_l + k_l)^{-1}$ we get

$$\Phi k_h^2 M(x = \Phi (k_h x + (k_h + k_l)Kx + 1 \quad \text{(3.17)}$$

Since addition and subtraction are the same in the extended field of GF(2), the first term on the right side of (3.17) can be moved to the left side. Comparing (3.12) and (3.17), it can be observed that

$$K^{-1} x = k_h \Phi x + (k_h + k_l) \Phi$$
To find and count the number of possible representations in $GF(2^8)$, as the extension of $GF(2^4)$, there are many algebraic properties. Reference [31] uses the following properties;

1. There are three polynomial representations of $GF(2^4)$ (over $GF(2)$). These are obtained by using the three irreducible reduction polynomials $1+x^4$, $1+x^3+x^4$, and $1+x+x^2+x^3+x^4$.
2. There are exactly 120 irreducible quadratic polynomials (over $GF(2^4)$) of the form $X^2 + \alpha x + \beta$ (where $\alpha$ and $\beta$ are in $GF(2^4)$). It follows that the field $GF(2^8)$ can be represented as the field extensions of $GF(2^4)$ in 360 ways.

The inversion process in $GF(2^4)$ using some reduction polynomials are presented in the following Table 3.1.

<table>
<thead>
<tr>
<th>Mod Pol.</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^4 + x + 1$</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>E</td>
<td>D</td>
<td>B</td>
<td>7</td>
<td>6</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>5</td>
<td>A</td>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>$x^4 + x^3 + 1$</td>
<td>0</td>
<td>1</td>
<td>C</td>
<td>8</td>
<td>6</td>
<td>F</td>
<td>4</td>
<td>E</td>
<td>3</td>
<td>D</td>
<td>B</td>
<td>A</td>
<td>2</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>$x^4 + x^3 + x^2 + x + 1$</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>A</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>

In this thesis the multiplicative inverse is a table looks up. This table is pre calculated and stored in memory (RAM). The table is provided in the next section. The implementation of inverse by modulo reduction technique consist of 20 XOR Gates, 9 3-input AND Gates and 6 2-input AND Gates. The hardware realization of this inversion is shown Figure 3.9 below;
F. Affine Transform and Inverse Affine Transform in GF(2^8)

The affine transform is the multiplication of 8-bit data (4-LSBs from one multiplier and 4-MSBs from another multiplier) by an 8×8 matrix consist of element of 0 and 1. There are 35 XORs required to implement the AT in GF(2^8). The software implementation for AT is given below.

\[
\begin{align*}
\end{align*}
\]
The inverse affine transform is similar to AT which is required to implement the inverse s-box for decryption. Inverse AT requires only 18 XORs. The software implementation of Inverse AT is provided below.

\[
\begin{align*}
\end{align*}
\]

### 3.3.3 Implementation of the Design

As mentioned earlier in the Sub Section 3.3.1 the low power is achieved by decomposing the s-box and implementing it in combinational logic in GF(2^4). During this process some other considerations are also considered. One of the key considerations is replacing the single multiplication block with a XOR Gate and two other sub blocks as shown in the following Figure 3.10.

![Figure 3.10 Equivalent multiplication block](image)

In Figure 3.10 the block-1 denotes 4×4 multiplication consist of 14 AND gates and 10 XOR gates. On the other hand block-2 denotes 4×4 multiplication consist of 12 AND gates and 8 XOR gates. Block-3 denotes 4-bit XOR gate. This modification results in reduction of power by balancing out the glitches from the data paths. The detail of the equivalent multiplication block is described by the following block diagram and logic equation.
Figure 3.11 Multiplication operation in GF(2^4).

Figure 3.11 shows the multiplication operation in GF(2^4) which is already discussed in Sub Section 3.3.2. Here three multiplication blocks of GF(2^2) are used to get the multiplication output in GF(2^4). The logic equation of Figure 3.11 is presented below;

Let D, E, and M are the variables of 4-bit long. Also let the multiplication result of M and E is stored in variable D. Then the result of multiplication in GF(2^4) is given below;

\[
D[0]=(E[3]\&M[3])\oplus(E[2]\&M[2])\oplus(E[1]\&M[1])\oplus(E[0]\&M[0]);
\]
\[
D[1]=(E[0]\&M[1])\oplus(E[1]\&M[0])\oplus(E[1]\&M[1])\oplus(E[2]\&M[2])\oplus(E[2]\&M[3])\oplus(E[3]\&M[2]);
\]
\[
D[2]=(E[0]\&M[2])\oplus(E[1]\&M[3])\oplus(E[2]\&M[0])\oplus(E[2]\&M[2])\oplus(E[3]\&M[1])\oplus(E[3]\&M[3]);
\]
\[
D[3]=(E[3]\&M[3])\oplus(E[3]\&M[1])\oplus(E[1]\&M[3])\oplus(E[2]\&M[3])\oplus(E[2]\&M[1])\oplus(E[0]\&M[3])\oplus(E[1]\&M[2])\oplus(E[3]\&M[0])\oplus(E[3]\&M[2]);
\]

Where ‘\(\oplus\)’ denotes the XOR operation and ‘\(\&\)’ denotes AND operation.

Here total 26 AND Gates and 22 XOR Gates are used. To free the glitches due to the unequal path length this thesis inserts buffers which is provided in Figure 3.10. Buffer insertion is performed by dividing the multiplication block into two parts. One part consists of 14 AND gates and 10 XOR gates. Other part consists of 12 AND gates and 8 XOR gates. Finally an XOR Gate is used to sum these two parts. Block-1 performs the following logic operations;
\[ a[1] = (E[0] \& M[1]) \oplus (E[1] \& M[0]) \oplus (E[1] \& M[1]) \]
\[ a[2] = (E[0] \& M[2]) \oplus (E[1] \& M[3]) \oplus (E[2] \& M[0]) \]

Block-2 performs the following logic operations;
\[ b[0] = (E[1] \& M[1]) \oplus (E[0] \& M[0]) \]

Finally the output comes out by Xoring a and b which is denoted by block-3.

\[ D = a \oplus b \]

Here the variable „a” and „b” are the buffers.

The second consideration is implementing the multiplicative inverse and multiplication by constant, \( \lambda \) using look up table. As these two processes are complex and use much more gates which causes delay and more power consumptions in the gate level, so this thesis uses the LUT based implementation. This table occupies only 128 bits long memory which is very compact.

The Table 3.2 presents the pre calculated values of 4-bit inverse and constant multiplier, \( \lambda \) which stored in RAM memory.

<table>
<thead>
<tr>
<th>( X^{-1} )</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>0100</td>
<td>0100</td>
<td>1000</td>
<td>1000</td>
<td>0001</td>
<td>1001</td>
<td>0101</td>
<td>0110</td>
<td>1010</td>
<td>0010</td>
<td>1110</td>
<td>1011</td>
<td>0111</td>
<td>1111</td>
<td>0011</td>
</tr>
</tbody>
</table>
The total process is presented in Figure 3.12 below. The composite field SubByte architecture in $GF(2^4)$ with 4 sub stages is presented in Figure 3.12. To reduce the unwanted paths as well as glitches from the datapath pipelined structure of 3-multipiers is used. Here the dotted line presents the registers.

Figure 3.12 Pipelined SubBytes in composite field $GF((2^4)^2)$. 


Chapter 4
FPGA Implementation

4.1 Introduction
This chapter will discuss about the FPGA implementation of the proposed design and also the intend procedure of the proposed AES processors using Verilog HDL and FPGA Implementation details will be described.

4.2 Implementation using FPGA
The proposed AES processor is implemented in FPGA device of family Altera, Cyclone II, EP2C35F672C6 device. The simulation is performed in Quartus II simulation software. The device specifications are given below;

- Total Logic Elements : 33216
- Total Registers : 3124
- Input Output (I/O) Registers : 33216
- Total Combinational Functions : 33216
- Dedicated Logic Registers : 33216
- Total Memory bits : 483,840

In this work the used FPGA device is Altera provided EP2C35F672C6 from Cyclone II family.

The Cyclone II device family offers the following features:

- High-density architecture with 4,608 to 68,416 LEs
- M4K embedded memory blocks
- Embedded multipliers
- Advanced I/O support
- Flexible clock management circuitry
- Device configuration
- Intellectual property
4.3 Components of the Design

The propose processors are a standalone system. Figure 4.1 shows the main module of the processor and its interred connections and relation.

![Diagram of AES Processor](image)

**Low Power AES Processor**

Figure 4.1 Low Power AES Processor

It consists of an encryptor and a decryptor module. The encryption and the decryption operations cannot be done with the same circuitry. So there are two separate encryptor and decryptor modules. There is another module for key expansion function. These are the top-level modules. The following sections describe these modules and their sub-modules. It is a standard practice to partition a complex design into different modules based on their specific functionality and features. So this thesis is partitioned into modules based on four basic operations and one key expansion operation for Encryption and same for decryption cycle. The four basic operations are substitution byte, shift rows, mix Columns, add round key for this design. Key expansion operation is performed inside the main module which is named as „key Expansion” to generate 10 round keys for next rounds from the supplied symmetric key. Add round key operation is also performed inside the main module „key Expansion” and AES standard module to add key to the
state. AES has nine standard rounds. Each round contains four basic operations; substitution byte, shift rows, mix column and add round key sequentially. If a round contains these 4 basic operations than this round is called standard round. So in AES-128 there are 9 standard rounds. These standard rounds are 1 to 9th round. So a module named onetonine is designed in this thesis to complete the operation of a standard round. The main module of Encryption and sub modules are as follows:-

**AES Encryption Module:** AES encryption module is the main module of encryption which holds the round module onetonine and it holds the other operational module like s-box, shift rows and mixcolumn module. Figure 4.2 shows the block diagram of the AES main module named as aes_encryption. Input data block of 128 bit is given as input which is plain text. Start bit is activated to start the operation. After getting the clock signal output of 128 bit cipher is generated. Beside this output keyout of 128 bit shows the last key of the encryption process, key ready flag indicates the completion of key expansion process and encr_ready flag indicates the ending of the encryption. Quartus II generated block diagram of encryption process is shown in Figure 4.2.

![Figure 4.2: Block Diagram of AES Encryption module](image)

**Onetonine Sub-Module:** This module is treated as standard module in AES which performs the operation of a standard round in AES. There are 9 standard rounds each includes all four operation of AES such as substitution byte, shift row, AddRoundKey, mix column operation. In this module 128 bit long output is generated from the 128 bit long input plaintext and 128 bit long input cipher key. The output is generated with output flag which represent the completion of the process. The block diagram of the round module which is named as onetonine is shown in Figure 4.3.
S-BOX Sub-Module: This module performs the Substitution byte operation of AES algorithm. This thesis uses combinational logic to implement the sub byte. The design involves the computation in GF($2^4$) which makes the proposed processor a novel low power processor. In this work 128 bit data block is given input to the s-box sub module as „boxin‟. After getting the clock „clk‟ the output is generated at „boxout‟. This sub module is named as aes_sub_byte and shown in Figure 4.4.

Shift Rows Sub-Module: This module is used for performing shift row operation of the AES. The block diagram of the module is shown in Figure 4.5. The operation of Shift Rows is to alter the position of the bytes in the 2nd, 3rd and 4th row on the state matrix. The 128 bit data block is given as input to this module and the 128 Bit data is produced at output terminal by the shift operation of the module and output flag represent the output is ready.
**MixColumn Sub-Module:** This is a operation in AES to multiply the present state of AES to a constant matrix by the multiplication rules used in GF(2^8) field. 128 bit input is given to this module. After performing the operation 128 bit output is produced. Output flag represent the ending of this sub module operation. Figure 4.6 shows the generated block diagram of this sub module.

There is another AES operation which is AddRoundKey where key is XORed with the state. There is no separate module for AddRoundKey but this is done inside main module of encryption and standard modules named as onetone module where other AES operation are done simultaneously.
Chapter 5
Experimental Results and Discussions

5.1 Introduction

The proposed AES processor is implemented in FPGA device of family Altera, Cyclone 2, EP2C35F672C6 device. The simulation is performed in quartus II simulation software. The device specifications are given below;

- Total Logic Elements: 33216
- Total Registers: 3124
- Input Output (I/O) Registers: 33216
- Total Combinational Functions: 33216
- Dedicated Logic Registers: 33216
- Total Memory bits: 483,840

5.2 Resources Used

The compilation results for the design are as follows;

- Total Logic Elements: 6733
- Total Registers: 512
- Total pins: 130
- Total Combinational Functions: 6533
- Dedicated Logic Registers: 512
- Total Memory bits: 32768

5.3 The Simulation Results

This thesis only considers the optimization of the s-box to get low power AES processor. During the simulation, the whole encryption module is simulated using Quartus II simulator. And also each block (s-box, shift row, mix column) is simulated independently. The simulation results of each block are provided in the following sub sections.
A.1 Simulation of S-box

Figure 5.1 Simulation result window for s-box operation

Figure 5.1 shows simulation results of combinational function based s-box implementation. Here the boxin takes input as the positive edge clk getting signal and generate the 128 bit output to the boxout.

A.2 Shift row Transformations

Figure 5.2 Simulation vector of Shift Row.

Figure 5.2 presents the shift row transformation of the AES processor. Here the boxin takes 128 bit data from the sub Byte transformation and shifts the 2nd, 3rd and 4th rows by 1, 2 and 3-bit left shift respectively. The first row is kept same as previous state. The compilation result shows that
the shift row transformation is only the bit shifting and that is why it does not occupy any memory blocks and also not any logic elements.

A.3 Mix Column Transformations

The mix column transformation is one of the more power consuming parts of AES processor. Figure 5.3 shows the vector output for the mix column transformation. Here it shows that the input of this transformation takes 128-bit data from the shift row transformation state and sends the output to the boxout by doing a cyclic permutation. The output is produced approximately half cycle later then the input arrived. This half cycle is the gate delay of mix column operation.

A.4 One to Nine Rounds

Figure 5.4 Simulation vector for One to Nine Rounds
Onetonine rounds actually perform the $N-1$ number of rounds which are called the standard rounds of the AES algorithm. Figure 5.4 presents the vector simulation of onetonine rounds which consists of sub byte, shift row and mix column transformations. Output is produced 3 cycles after receiving the input.

A.4 Full Encryption Module

Figure 5.5 below shows the full encryption module of the AES processor. Here the plaintext and plain key are assigned to the encryption module (using assignment operator of verilog HDL) and found the cipher output as shown in Figure 5.5. The input and plaintext and round key are provided same as NIST standard.

**The Plaintext** : 128'h3243f6a8885a308d313198a2e0370734

**The Plain Key** : 128'h2b7e151628aed2a6abf7158809cf4f3e

**The Cipher Output** : 128'H 3925841D02DC09FBDC118597196A0B32

From Figure 5.5 it is clear that the cipher output is found after 18 cycles i.e. 190 ns from the starting of the encryption process. So the latency of the encryption process is 18-clock cycles/190ns. The maximum clock frequency in the clock domain is found 99.79 MHz for the encryption process. The timing analysis provides the maximum frequency 99.79MHz and the speed performance is determined on the basis of this clock cycle.
The time periods of 99.79 MHz signal = \( \frac{1}{99.79 \times 10^6} = 10 \text{ ns} \).

Again a cycle processes a data sample of 128 bits. So the throughput of the proposed processor is = \( \frac{128}{10\text{ns}} = 12.773 \text{ Gbps} \).

Finally the proposed design-1 AES processor gives the throughput of 12.773 Gbps with the latency 18 cycles.

A.5 Decryption Module

The simulation result of decryption module is provided in Figure 5.6. The input of the decryption process is the outputs of the encryption process i.e. cipher. The round key for the decryption is the last key generated by the key schedule algorithm of encryption process. The cipher text and round key are provided same as NIST standard.

The Cipher text : 128'H3925841D02DC09FBDC118597196A0B32
The Last Key : 128'HD014F9A8c9EE2589E13F0CC8B6630CA6
The Cipher Output : 128'H3243F6A8885A308D313198A2E0370734

![Figure 5.6 Simulation vector wave shape for Full Decryption Module](image)

The decryption process is the same as encryption process. The codes of implementation of decryption are provided at appendix section of this thesis paper.
5.4 Power Analysis and Measurement of Power consumption

In this section the total power consumption of the proposed processor is determined and also power is calculated for each block.

As mentioned earlier in Section 3.5, power consumed in a chip can be broken down in a few different ways:

- Power = static power + dynamic power
- Static Power = gate leakage power + subthreshold leakage power + reverse bias diode leakage power.
- Dynamic Power = Switching Power + Internal short circuit power.

As static power is consumed regardless of the amount of activity in a chip it totally depends on processor’s temperature. Static power does not depend upon the switching activity (No SAIF or VCD file needed) not toggle count involved. So this thesis does not focus on static power. Again though Dynamic Power is power consumed in charging and discharging a capacitive load therefore, dynamic power requires knowledge of switching activity (or toggle rate on a node) to estimate switching power consumption. In this section the dynamic power consumption is analyzed with the results from power play power analizer tool. Dynamic Power can be explained with

\[ P_{\text{Dynamic}} = V \cdot A = V \cdot \left( \frac{Q}{T} \right) = CV^2/T \]  

Where \( C \) = Capacitance of the Gate  
\( V \) = Chip operating Voltage  
\( T \) = Transition time

This equation shows that power consumption is proportional to capacitance and to square of voltage. This thesis presents Toggle Rate based power analysis.

**Toggle Rate based Power Analysis:**

This type of analysis is based on the following two power state [29];

- Internal Power = \((E_{\text{rise}} + E_{\text{fall}})/2\) * Tr\ponsible for\[ (5.2) \]

Where Tr is Toggle Rate. No of times a signal changes state between HI and LO per unit time. Erise and Efall are derived from technology library using input transition and output capacitance.

- Switching Power = \((CV^2/2\) * Tr \[ (5.3) \]
i) At first the .saf and .vcd files are created from the vector.vwf file with the help of vectorless estimation tool.

ii) To calculate the $E_{\text{rise}}$ and $E_{\text{fall}}$ this thesis uses Technology library file that contains the power related information like leakage power (can be state dependent) and energy per transition (can be state dependent and path dependent).

iii) Toggle Rate is number of times that a node changes state between HIGH and LOW per unit time. Toggle Rate information is obtained from SAIF file and VCD file.

iv) To reduce toggle count this thesis uses normal bit shifting and xoring instead of complex multiplication. For an example, let $Z = \frac{1}{4}(A + B) = \frac{1}{4} A + \frac{1}{4} B$. Note that $\frac{1}{4} A$ requires simple shift thrice to right. $\frac{1}{4}A = A >>$.

The following sub sections estimate the power consumption in static and dynamic for this design.

A. Power consumption

The scenario of total power consumption of the proposed design is shown in Figure 5.7. Here only the encryption module’s power consumption is provided. Because the decryption process utilizes approximately the same power as the encryption process does. Here the power calculation is only for the encryption module. If some one doubles the power of encryption module he or she can get the total power (Encryption + Decryption) of the AES processor.

<table>
<thead>
<tr>
<th>Power Play Power Analyzer Status</th>
<th>Successful - Thu Oct 20 02:34:45 2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quartus II Version</td>
<td>7.0 Build 33 02/05/2007 SJ Web Edition</td>
</tr>
<tr>
<td>Revision Name</td>
<td>aes_encryption</td>
</tr>
<tr>
<td>Top-level Entity Name</td>
<td>aes_encryption</td>
</tr>
<tr>
<td>Family</td>
<td>Cyclone II</td>
</tr>
<tr>
<td>Device</td>
<td>EP2C35F672C6</td>
</tr>
<tr>
<td>Power Models</td>
<td>Final</td>
</tr>
<tr>
<td>Total Thermal Power Dissipation</td>
<td>153.41 mW</td>
</tr>
<tr>
<td>Core Dynamic Thermal Power Dissipation</td>
<td>21.17 mW</td>
</tr>
<tr>
<td>Core Static Thermal Power Dissipation</td>
<td>80.00 mW</td>
</tr>
<tr>
<td>I/O Thermal Power Dissipation</td>
<td>52.25 mW</td>
</tr>
<tr>
<td>Power Estimation Confidence</td>
<td>High: user provided sufficient toggle rate data</td>
</tr>
</tbody>
</table>

Figure 5.7 Power consumption of the full Encryption module.
Table 5.1 shows the blockwise thermal power dissipation. This table shows that the dynamic power consumption is more in combinational cell. This effect is mentioned in the previous section of power analysis.

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Total Thermal Power by Block Type</th>
<th>Block Thermal Dynamic power</th>
<th>Block thermal Static power</th>
<th>Routing thermal DP</th>
<th>Average Toggle of Transition/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>22.76 mW</td>
<td>6.23 mW</td>
<td>16.44 mW</td>
<td>0.10 mW</td>
<td>3.587</td>
</tr>
<tr>
<td>Combinational cell</td>
<td>21.00 mW</td>
<td>20.01 mW</td>
<td>--</td>
<td>0.99 mW</td>
<td>7.04</td>
</tr>
<tr>
<td>Register cell</td>
<td>0.17 mW</td>
<td>0.14 mW</td>
<td>--</td>
<td>0.03 mW</td>
<td>0.431</td>
</tr>
</tbody>
</table>

From the Table 5.1 the total dynamic power dissipation are found 21.17 mW for encryption. From the technology library file the following parameters is found.

For I/O:

\[
E_{\text{rise}} = 1.01 \\
E_{\text{fall}} = 0.09 \\
C = 1.686 \text{ and from the Table 5.1} \\
T_r (\text{Toggle rate}) = 3.587 \\
V = 1.2 \text{ V (Cyclone II EP2C35 device operating voltage)}
\]

From Equation 5.2 and 5.3

\[
\text{Internal Power} = \frac{(E_{\text{rise}} + E_{\text{fall}})}{2} \times T_r \\
= \frac{(1.01 + 0.09)}{2} \times 3.587 = 1.972 \text{ mW}
\]

and

\[
\text{Switching Power} = \frac{(CV^2)}{2} \times T_r \\
= 1.68 \times \frac{(1.2 \times 1.2)}{2} \times 3.587 = 4.33 \text{ mW}
\]

so the total I/O dynamic power = Internal power + Switching Power

\[
= 1.972 + 4.33 = 6.302 \text{ mW which is approximately same as Table 5.1 data.}
\]


For Combinational Cell:

\[ E_{\text{rise}} = 2.24 \]
\[ E_{\text{fall}} = 1.012 \]
\[ C = 1.686 \text{ and from the Table 5.1} \]
\[ T_r \text{ (Toggle rate)} = 7.04 \text{ mil Transition/sec} \]
\[ V=1.2 \text{ V (Cyclone II EP2C35 device operating voltage)} \]

From Equation 5.2 and 5.3

\[ \text{Internal Power} = \frac{E_{\text{rise}} + E_{\text{fall}}}{2} \times T_r \]
\[ = \frac{(2.24 + 1.012)}{2} \times 7.04 = 11.44 \text{ mW} \]
again

\[ \text{Switching Power} = \frac{CV^2}{2} \times T_r \]
\[ = 1.68 \times (1.2 \times 1.2)/2 \times 7.04 = 8.51 \text{ mW} \]

So the total I/O dynamic power = Internal power + Switching Power
\[ = 11.44 + 8.51 = 19.95 \text{ mW} \]
Which is approximately same as experimental results of Table 5.1 data.

For Register Cell:

\[ E_{\text{rise}} = 0.01 \]
\[ E_{\text{fall}} = 0.02 \]
\[ C = 1.686 \text{ and from the Table 5.1} \]
\[ T_r \text{ (Toggle rate)} = 0.431 \text{ mil Transition/sec} \]
\[ V=1.2 \text{ V (Cyclone II EP2C35 device operating voltage)} \]

From Equation 5.2 and 5.3

\[ \text{Internal Power} = \frac{E_{\text{rise}} + E_{\text{fall}}}{2} \times T_r \]
\[ = \frac{(0.01 + 0.02)}{2} \times 0.431 = 0.006 \text{ mW} \]
again

\[ \text{Switching Power} = \frac{CV^2}{2} \times T_r \]
\[ = 1.68 \times (1.2 \times 1.2)/2 \times 0.431 = 0.05 \text{ mW} \]

So the total I/O dynamic power = Internal power + Switching Power
\[ = 0.006 + 0.05 = 0.056 \text{ mW} \]
5.5 Comparison with Other Related Works

Table 5.2 shows the comparison of power, throughput and latency of the proposed design with that of other research works. It is observed that the proposed design is superior to all the existing designs in terms of power consumption and throughput. So the proposed design will be nice for low power application which is the vision of this era. The design also offers low latency which makes it very much suitable for real time application.

<table>
<thead>
<tr>
<th>Design</th>
<th>Device</th>
<th>( F_{\text{Max}} ) (MHz)</th>
<th>Throughput (Mbps)</th>
<th>Latency</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Katashita BRAM-1 [24]</td>
<td>Virtex II</td>
<td>66.66</td>
<td>776</td>
<td>11</td>
<td>80</td>
</tr>
<tr>
<td>Katashita LUT-1 [24]</td>
<td>Virtex II</td>
<td>50</td>
<td>582</td>
<td>11</td>
<td>80</td>
</tr>
<tr>
<td>Alam [30]</td>
<td>Virtex II</td>
<td>240</td>
<td>7680</td>
<td>44</td>
<td>80</td>
</tr>
<tr>
<td>Rejeb et al S_box [22]</td>
<td>Virtex 4</td>
<td>123</td>
<td>40</td>
<td>400</td>
<td>80</td>
</tr>
<tr>
<td>Rejeb et al S_box [22]</td>
<td>Virtex 4</td>
<td>145</td>
<td>142</td>
<td>130</td>
<td>80</td>
</tr>
<tr>
<td>Kenny D [26]</td>
<td>Cyclone II</td>
<td>198.93</td>
<td>2546</td>
<td>40</td>
<td>80.03</td>
</tr>
<tr>
<td>This Thesis</td>
<td>Cyclone II</td>
<td>99.79</td>
<td>12773</td>
<td>18</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 6
Conclusions

6.1 Conclusion

At present AES is widely used and the most famous NIST approved security algorithm to face any kind of challenges of hacking in various application. Design of AES processor for resource constrained (battery powered) and low power devices are now an important area of research. With this keeping in mind this thesis introduces a new technology to design a low power AES processor which can support those applications in the future days. A few novel techniques have been incorporated in the design of the AES processor presented in this thesis. Firstly the s-box of the proposed design is based on the composite field GF(2^4)^2 which in turn makes the design simpler and reduces the memory block in a significant amount that lowers the power consumption. Here optimized s-box is also considered for this architecture using the look up table based implementation of only two operations which are multiplicative inverse and multiplication by constant. Secondly the proposed s-box is filtered out the glitches by replacing a 4-bit multiplier with two 4-bit multipliers and introducing an XOR buffer into each stage. Thirdly to reduce the dynamic power to an optimum level a pipelined architecture of 3-multipliers within the s-box is incorporated by putting the registers in between two consecutive operations which decreases the dynamic power of the processor. The simulation is performed in quartus II simulation software. The final design is implemented in FPGA device of family Altera, Cyclone II, EP2C35F672C6 device. Simulation results of the AES processor ensure that the design functions properly. The ciphered output from the design is verified with the NIST provided output with respect to a specific input. The total power of the processor is measured with the PowerPlay power analyzer tool of the simulator. The dynamic power is found 21.17 mW for the encryption process. The power consumption, throughput and latency of the design have been compared with those of other researchers. The comparison shows that the proposed design outperforms all other existing designs in terms of power consumption and throughput. The latency is also low which ensures that the design can operate in the real time.
6.2 Future Works

The research performs in this thesis also exposes several new research areas that can be explored.

- This thesis uses $GF(2^4)^2$ for implementing the s-box using the irreducible polynomial of $x^4+1$. This work suggests that using other irreducible polynomials to implement the s-box can dissipate less power.

- The second most power consuming parts of AES processor is Key expansion unit and Mix column unit. It may be the fruitful design if $GF(2^4)^2$ will be used into the both units.
References


[31] Practical Implementation of Rijndael S-Box Using Combinational Logic Edwin NC Mui Custom R & D Engineer Texco Enterprise Ptd. Ltd.
Appendix A

Encryption Codes

Encryption Unit
module aes_encryption(cipher, encr_ready, clk);
output [127:0] cipher;
output encr_ready;
reg encr_ready=0;
//input [127:0] plaintext;
input clk;
integer i, flag, j, k=0;
wire [127:0] shiftout, subout[0:10], modout[0:8], plaintext;
reg [127:0] key[0:10];
//wire [255:0] outkey[0:4];
//reg [7:0] roundkey[0:15], showout;
reg [127:0] newround[0:8];
reg [127:0] shiftin, subin[0:10], newcipher[10:0], cipher=0, modin[0:8];
assign plaintext=128'H3243f6a8885a308d313198a2e0370734;
reg [95:0] dummy;
reg [31:0] re[10:0];
reg [31:0] a[43:0];
reg [7:0] b0, b1, b2, b3;
aes_sub_byte asb1(subout[1], subin[1], clk);
aes_sub_byte asb2(subout[2], subin[2], clk);
aes_sub_byte asb3(subout[3], subin[3], clk);
aes_sub_byte asb4(subout[4], subin[4], clk);
aes_sub_byte asb5(subout[5], subin[5], clk);
aes_sub_byte asb6(subout[6], subin[6], clk);
aes_sub_byte asb7(subout[7], subin[7], clk);
aes_sub_byte asb8(subout[8], subin[8], clk);
aes_sub_byte asb9(subout[9], subin[9], clk);
aes_sub_byte asb10(subout[10], subin[10], clk);
// Round constants value for Key Expansion module
assign
{rc[1],rc[2],rc[3],rc[4],rc[5],rc[6],rc[7],rc[8],rc[9],rc[10]}=32'h010000002000000400000008
000001000000200000040000000800000001B0000036000000;

// Start of key expansion
always @(posedge clk) begin
{a[0],a[1],a[2],a[3]}=128'H2b7e151628aed2a6abf7158809cf4f3c;
//H000102030405060708090a0b0c0d0e0f;
//H2b7e151628aed2a6abf7158809cf4f3c;
begin
for(j=4:j<44;j=j+1)
if(j%4==0)
begin
//{b3,b2,b1,b0}=a[i-1];
subin[j/4]={a[j-1],dummy};
{b3,b2,b1,b0,dummy}=subout[j/4];
a[j]=({b2,b1,b0,b3}^rc[j/4])^a[j-4];
end
else a[j]=a[j-1]^a[j-4];
end
for(k=0;k<11;k=k+1)begin
key[k]={a[4*k],a[4*k+1],a[4*k+2],a[4*k+3]};
end
end
//end of key expansion
aes_sub_byte asb(subout[0],subin[0],clk);
aes_shift_row asr(shiftout,shiftin,clk);
onetonine otn1(modout[0],modin[0],newround[0],clk);
onetonine otn2(modout[1],modin[1],newround[1],clk);
onetonine otn3(modout[2],modin[2],newround[2],clk);
onetonine otn4(modout[3],modin[3],newround[3],clk);
onetonine otn5(modout[4],modin[4],newround[4],clk);
onetonine otn6(modout[5],modin[5],newround[5],clk);
onetonine otn7(modout[6],modin[6],newround[6],clk);
onetonine otn8(modout[7],modin[7],newround[7],clk);
onetonine otn9(modout[8],modin[8],newround[8],clk);
always @(posedge clk) begin
for(i=0;i<11;i=i+1)begin
flag=0;
    case(i)
        0:begin
        newcipher[i]=plaintext^key[0];
        end
        1:begin
        modin[i-1]=newcipher[i-1];
        newround[i-1]=key[i];
        newcipher[i]=modout[i-1];
        end
        2:begin
        modin[i-1]=newcipher[i-1];
        newround[i-1]=key[i];
        newcipher[i]=modout[i-1];
        end
        3:begin
        modin[i-1]=newcipher[i-1];
        newround[i-1]=key[i];
        newcipher[i]=modout[i-1];
        end
        4:begin
        modin[i-1]=newcipher[i-1];
        newround[i-1]=key[i];
        newcipher[i]=modout[i-1];
        end
        5:begin
        modin[i-1]=newcipher[i-1];
        newround[i-1]=key[i];
        newcipher[i]=modout[i-1];
        end
        6:begin

modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
end

7:begin
    modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
end

8:begin
    modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
end

9:begin
    modin[i-1]=newcipher[i-1];
    newround[i-1]=key[i];
    newcipher[i]=modout[i-1];
end

10:begin
    subin[0]=newcipher[i-1];
    shiftin=subout[0];
    newcipher[i]=shiftout;
    cipher=key[i]^newcipher[i];

    if(cipher==128'h3925841d02dc09fbdc118597196a0b32)begin
        encr_ready=1;
    end

    end
endcase
end

end //always
endmodule  // end of encryption
Sub Byte Transformations

module aes_sub_byte(boxout,boxin,clk);
output [127:0] boxout;
input [127:0] boxin;
input clk;
integer i;
reg [7:0] q, p, y, c;
reg [7:0] D;       // total & and ^ = 293 within the for loop
reg [7:0] s[0:255];
reg [7:0] ain, bin, cin, din, ein, gin, hin, iin, jin, kin, lin, min, nin, oin, pin,
            aout, bout, cout, dout, eout, fout, gout, hout, iout, jout, kout,
            lout, mout, nout, oout, pout;
// 00aabbff0b00e015378668899fece00
assign {ain, bin, cin, din, ein, gin, hin, iin, jin, kin, lin, min, nin, oin, pin} = boxin;
assign {s[0]} = 8'h63;
always@* begin
for (i=1; i<256; i=i+1) begin
    q = i;
    //Mapping for GF(2^4)
    p[0] = q[0]^q[1]^q[6];
    // Squareing
    N[3] = p[7];
    // Xoring of Ph and Pl
//M=p[3:0]^p[7:4];
M[0]=p[0]^p[4];

// Multiplication by 4'h1110
H[0]=N[2];
H[1]=N[3];

// multiplication X1
G[0]=(p[3]&M[3])^(p[2]&M[3])^(p[3]&M[2])^(p[1]&M[1])^(p[0]&M[0]);
G[1]=(p[0]&M[1])^(p[1]&M[0])^(p[1]&M[1])^(p[2]&M[2])^(p[2]&M[3])^(p[3]&M[2]);
G[2]=(p[0]&M[2])^(p[1]&M[3])^(p[2]&M[0])^(p[2]&M[2])^(p[3]&M[1])^(p[3]&M[3]);
G[3]=(p[3]&M[3])^(p[3]&M[1])^(p[1]&M[3])^(p[2]&M[3])^(p[2]&M[1])^(p[0]&M[3])^(p[1]&M[2])^(p[3]&M[0])^(p[3]&M[2]);

// xORING
//F=H^G;
F[0]=H[0]^G[0];

//Making Inverse

// mULTIPLICATION X2
D[0]=(E[3]&M[3])^(E[2]&M[3])^(E[1]&M[2])^(E[1]&M[1])^(E[0]&M[0]);
D[1]=(E[0]&M[1])^(E[1]&M[0])^(E[1]&M[1])^(E[2]&M[2])^(E[2]&M[3])^(E[3]&M[2]);
D[2]=(E[0]&M[2])^(E[1]&M[3])^(E[2]&M[0])^(E[2]&M[2])^(E[3]&M[1])^(E[3]&M[3]);
D[3]=(E[3]&M[3])^(E[3]&M[1])^(E[1]&M[3])^(E[2]&M[3])^(E[2]&M[1])^(E[0]&M[3])^(E[1]&M[2])^(E[3]&M[0])^(E[3]&M[2]);
//MULTIPLICATION X3

//INVERSE MAPPING

//AFFINE TRANSFORM
Y[7] = C[7]&C[6]&C[5]&C[4]&C[3];
Y[4] = C[4]&&C[3]&&C[2]&&C[1]&&C[0];
Y[3] = C[7]&&C[3]&&C[2]&&C[1]&&C[0];
Y[2] = C[7]&&C[6]&&C[2]&&C[1]&&C[0];
Y[1] = C[7]&&C[6]&&C[5]&&C[1]&&C[0]&&1;
Y[0] = C[7]&&C[6]&&C[5]&&C[4]&&C[0]&&1;
s[i] = y;
end // END OF FOR
//end
//always @(posedge clk) begin
aout = s[ain];
bout = s[bin];
cout = s[cin];
dout = s[din];
module aes_mix_column (mixout, mixin, clk);
output [127:0] mixout;
input clk;
input [127:0] mixin;
integer i;
reg [7:0] s[0:15], s_shift[0:15], sout[0:15];
assign {s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], s[9], s[10], s[11], s[12], s[13], s[14], s[15]} = mixin;
always @(posedge clk) begin
    for (i=0; i<16; i=i+1)
        if (s[i] >= 8'b10000000)
            s_shift[i] = s[i] ^ 8'b00011011;
        else
            s_shift[i] = s[i] << 1;
    for (i=0; i<=12; i=i+4)
        begin
            sout[i] = s_shift[i] ^ s_shift[i+1] ^ s_shift[i+2] ^ s_shift[i+3] ^ s[i];
            sout[i+1] = s[i] ^ s_shift[i+1] ^ s_shift[i+2] ^ s_shift[i+3] ^ s[i];
            sout[i+2] = s[i] ^ s_shift[i+1] ^ s_shift[i+2] ^ s_shift[i+3] ^ s[i];
            sout[i+3] = s[i] ^ s_shift[i+1] ^ s_shift[i+2] ^ s_shift[i+3] ^ s[i];
        end
end
endmodule
shiftout[i+3]=s_shift[i]*s[i]*s[i+1]*s[i+2]*s_shift[i+3];
end
end
assign
mixout={sout[0],sout[1],sout[2],sout[3],sout[4],sout[5],sout[6],sout[7],sout[8],sout[9],sout[10],sout[11],sout[12],sout[13],sout[14],sout[15]};
endmodule

Shift Row Transformation

module aes_shift_row(shiftout,shiftin,clk);
output [127:0] shiftout;
input clk;
input [127:0] shiftin;
reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin;
assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = shiftin;
assign shiftout={ain,fin,kin,pin,ein,jin,oin,din,iin,nin,cin,hin,min,bin,gin,lin};
endmodule

Onetonine Rounds

module onetonine(oneout,onein,roundkey,clk);
output [127:0] oneout;
input [127:0] onein,roundkey;
input clk;
wire [127:0] subout,shiftout,mixout;
reg [127:0]shiftin,subin,mixin,oneout;
aes_sub_byte asb(subout,subin,clk);
aes_shift_row asr(shiftout,shiftin,clk);
aes_mix_column amc(mixout,mixin,clk);
always @(posedge clk ) begin
    subin=onein;
    shiftin=subout;
    mixin=shiftout;
oneout = mixout\textasciicircum roundkey;

end //always
endmodule
Appendix B

Decryption

module decryption(
    finalkey,keyready,decr_ready,cipher,plaintext,clk,start
);

output [127:0]finalkey,plaintext;
output keyready,decr_ready;
input [127:0]cipher;
input clk,start;
integer i,j,k=0;
wire [127:0]shiftout,subout[0:10],modout[0:8];
reg [127:0]key[0:10];
reg [127:0]newround[0:8], keyout;
reg [127:0]shiftin,subin[0:10],newcipher[10:0],plaintext,modin[0:8],finalkey;
reg decr_ready=0,keyready=0;
reg [95:0]dummy;
reg [31:0]rc[10:0];
reg [31:0]a[43:0];
reg [7:0]b0,b1,b2,b3;
reg [31:0]p[0:3];
assign
plaintext=128'h3243f6a8885a308d313198a2e0370734;
aes_sub_byte asb1(subout[1],subin[1],clk);
aes_sub_byte asb2(subout[2],subin[2],clk);
aes_sub_byte asb3(subout[3],subin[3],clk);
aes_sub_byte asb4(subout[4],subin[4],clk);
aes_sub_byte asb5(subout[5],subin[5],clk);
aes_sub_byte asb6(subout[6],subin[6],clk);
aes_sub_byte asb7(subout[7],subin[7],clk);
aes_sub_byte asb8(subout[8],subin[8],clk);
aes_sub_byte asb9(subout[9],subin[9],clk);
aes_sub_byte asb10(subout[10],subin[10],clk);
    //Round constant value for key Expansion module
assign
{rc[1],rc[2],rc[3],rc[4],rc[5],rc[6],rc[7],rc[8],rc[9],rc[10]}=32'h01000000020000000400000008
00000100000002000000040000000800000001B00000036000000;
    // Start of key expansion
always@(posedge clk)
begin
    keyout=0;
    //input of 128bit key
{a[0],a[1],a[2],a[3]}=128'H2b7e151628aed2a6abf7158809cf4f3c;
begin
    for(j=4;j<44;j=j+1)
if(j%4==0)
begin
    subin[j/4]={a[j-1],dummy};
    {b3,b2,b1,b0,dummy}=subout[j/4];
a[j]=({b2,b1,b0,b3}^rc[j/4])^a[j-4];
end
else a[j]=a[j-1]^a[j-4];
end
for(k=0;k<11;k=k+1)
begin
    key[k]={a[4*k],a[4*k+1],a[4*k+2],a[4*k+3]};
    keyout=key[k];
if(keyout==128'Hd014f9a8e9ee2589e13f0cc8b6630ca6)begin
    finalkey=key[10];
    keyready=1;
end
end
end //end of the key expansion
    //start of decryption
onetonine otn1(modout[0],modin[0],newround[0],clk);
onetonine otn2(modout[1],modin[1],newround[1],clk);
onetonine otn3(modout[2],modin[2],newround[2],clk);
onetonine otn4(modout[3],modin[3],newround[3],clk);
onetonine otn5(modout[4],modin[4],newround[4],clk);
onetonine otn6(modout[5],modin[5],newround[5],clk);
onetonine otn7(modout[6],modin[6],newround[6],clk);
onetonine otn8(modout[7],modin[7],newround[7],clk);
onetonine otn9(modout[8],modin[8],newround[8],clk);
aes_shift_row asr(shiftout,shiftin,clk);
aes_sub_byte asb(subout[0],subin[0],clk);
always@* begin
decr_ready=0;
//{p[0],p[1],p[2],p[3]}=128'h3243f6a8885a308d313198a2e0370734;
//{plaintext}={p[0],p[1],p[2],p[3]};
if(start)begin
    for(i=0;i<11;i=i+1)begin
        case(i)
        0:begin
            newcipher[i]=cipher^key[10-i];
        end
        1:begin
            modin[i-1]=newcipher[i-1];
            newround[i-1]=key[10-i];
            newcipher[i]=modout[i-1];
        end
        2:begin
            modin[i-1]=newcipher[i-1];
            newround[i-1]=key[10-i];
            newcipher[i]=modout[i-1];
        end
        3:begin
            modin[i-1]=newcipher[i-1];
        end
    end
end
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
4:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
5:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
6:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
7:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
8:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
9:begin
modin[i-1]=newcipher[i-1];
newround[i-1]=key[10-i];
newcipher[i]=modout[i-1];
end
10:begin
INV Sub Byte Transformations

module INV_s_box24(boxout, boxin, clk);
output [127:0] boxout;
input [127:0] boxin;
input clk;
reg[7:0] s[0:255];
reg [7:0] q, p, y, c;
reg [7:0] D;
wire [7:0] s[0:255];
reg [7:0] a[0:255];
assign {ain, bin, cin, din, ein, fin, gin, hin, iin, jin, kin, lin, min, nin, oin, pin,
aout, bout, cout, dout, eout, fout, gout, hout, iout, jout, kout, lout, mout, nout, oout, pout} = boxin;
always @(posedge clk) begin
for(i=0;i<=8'hff;i=i+1) begin
q = i;
// Mapping for GF(24)
p[0] = q[0]^q[1]^q[6];
p[4]=q[1]*q[2]*q[3]*q[5]*q[7];
p[5]=q[2]*q[3]*q[5]*q[7];
p[6]=q[1]*q[2]*q[3]*q[4]*q[6]*q[7];
p[7]=q[5]*q[7];

    // Squareing
N[0]=p[4]*p[5]*p[7];
N[1]=p[6]*p[5];
N[2]=p[6]*p[7];
N[3]=p[7];

    // Xoring of Ph and Pl
M=p[0:3]*p[4:7];

    // Multiplication by 4'h1110
H[0]=N[2];
H[1]=N[3];
H[3]=N[0]*N[2];

    // multiplication X1
G[0]=(P[3]&M[3])&(P[2]&M[3])&(P[3]&M[2])&(P[1]&M[1])&(P[0]&M[0]);
    &M[2])&(P[3]&M[0])&(P[3]&M[2]);

    // XORING
F=H^G;

    //Making Inverse
    &(F[2]&F[0]&F[1])&(F[2]&F[0]&F[1])&(F[2]&F[1]&F[1]);
E[0]=(F[3]&F[2]&F[1])&(F[3]&F[2]&F[0])&(F[3]&F[2]&F[0])&(F[3]&F[2]&F[1]);

    // MULTIPLICATION X2
D[0]=(E[3]&M[3])&(E[2]&M[3])&(E[1]&M[1])&(E[0]&M[0]);

// MULTIPLICATION X3

// INVERSE MAPPING

// AFFINE TRANSFORM
\[ s[i] = Y; \]

end

always @(posedge clk) begin
\[ aout <= s[ain]; \]
\[ bout <= s[bin]; \]
\[ cout <= s[cin]; \]
\[ dout <= s[din]; \]
eout<=s[ein];
fout<=s[fin];
gout<=s[gin];
hout<=s[hin];
iout<=s[iin];
jout<=s[jin];
kout<=s[kin];
lout<=s[lin];
mout<=s[min];
nout<=s[nin];
oout<=s[oin];
pout<=s[pin];
end
assign boxout={aout,bout,cout,dout,eout,fout,gout,hout,iout,jout,kout,lout,mout,nout,oout,pout}; endmodule

**INV-Shift Rows Transformations**

module INV_shift_row(shiftout,shiftin,clk);
output [127:0] shiftout;
input clk;
input [127:0] shiftin;
reg [7:0]ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin;
assign {ain,bin,cin,din,ein,fin,gin,hin,iin,jin,kin,lin,min,nin,oin,pin} = shiftin;
assign shiftout={ain,nin,kin,hin,ein,bin,oin,lin,iin,fin,cin,min,jin,gin,din};
endmodule

**INVERSE MIX COLUMN MODULE**

//AES Inverse Mix_Column Module(For Decryption)
module aesi_mix_column(imixout,imixin,clk);
output [127:0] imixout;
input [127:0] imixin;
input clk;
reg [7:0] temp2,temp4,temp8,carry2,carry4,carry8;
reg [7:0] shift9[0:15],shiftB[0:15],shiftD[0:15],shiftE[0:15];
reg [7:0] mixin[0:15],shift2[0:15],shift4[0:15],shift8[0:15],result[0:15];
integer i;
reg [127:0] imixout;
always @ (imixin or temp2 or temp4 or temp8 or carry2 or carry4 or carry8 or shift2 or shift4 or shift8 or shift9 or shiftB or shiftD or mixin or shiftE ) begin
  temp2=0; temp4=0; temp8=0; carry2=0; carry4=0; carry8=0;
  for(i=0;i<16;i=i+1) begin
    mixin[i]=0;
    shift2[i]=0;
    shift4[i]=0;
    shift8[i]=0;
    shift9[i]=0;
    shiftB[i]=0;
    shiftD[i]=0;
    shiftE[i]=0;
  end //for
  {mixin[0],mixin[1],mixin[2],mixin[3],mixin[4],mixin[5],mixin[6],mixin[7],mixin[8],mixin[9],mixin[10],mixin[11],mixin[12],mixin[13],mixin[14],mixin[15]}=imixin;
  for(i=0;i<16;i=i+1)begin
    {carry2,temp2}=mixin[i]<<1;
    if(carry2)
      shift2[i]=temp2^8'b00011011;
    else
      shift2[i]=temp2;
    {carry4,temp4}=shift2[i]<<1;
    if(carry4)
      shift4[i]=temp4^8'b00011011;
    else
      shift4[i]=temp4;
    {carry8,temp8}=shift4[i]<<1;
    if(carry8)
      shift8[i]=temp8^8'b00011011;
    else
      shift8[i]=temp8;
  end //for
{mixin[0],mixin[1],mixin[2],mixin[3],mixin[4],mixin[5],mixin[6],mixin[7],mixin[8],mixin[9],mixin[10],mixin[11],mixin[12],mixin[13],mixin[14],mixin[15]}=imixin;
shift8[i]=temp8^8'b00011011;
else
    shift8[i]=temp8;
end
shift9[i]=shift8[i]^mixin[i];
shiftB[i]=shift9[i]^shift2[i];
shiftD[i]=shift9[i]^shift4[i];
shiftE[i]=shift8[i]^shift4[i]^shift2[i];
end
imixout=[result[0],result[1],result[2],result[3],result[4],result[5],result[6],
result[7],result[8],result[9],result[10],result[11],result[12],result[13],result[14], result[15]];
end //always
endmodule

**INV-onetonine Module**

module onetonine_inv(oneout,onein,roundkey,clk);
output[127:0] oneout;
input [127:0] onein,roundkey;
input clk;
wire [127:0] subout, shiftout, mixout;
reg [127:0] shiftin, subin, mixin, oneout;
aes_sub_byte asb3(subout, subin, clk);
aes_shift_row asr1(shiftout, shiftin, clk);
aes_mix_column amc(mixout, mixin, clk);
always @(posedge clk) begin
  shiftin = onein;
  subin = shiftout;
  mixin = subout ^ roundkey;
  oneout = mixout;
end //always
endmodule
Appendix C

\[ x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x \mod \ x^8 + x^4 + x^3 + x + 1 = (11111101111110 \mod 100011011) = 1. \]

The verilog HDL code for the above three operations in GF(2^8):

```verilog
module Add_Sub_Mul(a,b,out);
input [7:0] a,b;
output [7:0] out;
reg [7:0] out=0;
reg [7:0] hi_bit_set;
integer counter;
always@* begin
    out = a^b;  // Add two numbers in a GF(2^8) finite field */
    out = a^b;  // Subtract two numbers in a GF(2^8) finite field */
    // Multiply two numbers in the GF(2^8) finite field defined by the polynomial x^8 + x^4 + x^3 + x + 1 */
    for(counter = 0; counter < 8; counter=counter+1)begin
        if(b&1)
            out = p^a;
        else
            hi_bit_set = a^8"h80;
        a<<1;
        if(hi_bit_set)
            a =a^9"h11b;
        b>>1;
    end // end of for loop
end // end for always
endmodule
```
Appendix D

Pin Assignment from .csv file for Cyclone II. Just import the assignment. You have to save this text file in .csv extension which will produce the excel file.

# Quartus II Version 7.0 Internal Build 160 09/19/200 TO Full Version,
# File: D:\de2_pins\de2_pins.csv,
# Generated on: Wed Sep 28 09:40:34 2011,

# Note: The column header names should not be changed if you wish to import this .csv file into the Quartus II software.

To,Location
plaintext[127],PIN_AA1
plaintext[126],PIN_AA2
plaintext[125],PIN_AA3
plaintext[124],PIN_AA4
plaintext[123],PIN_AA5
plaintext[122],PIN_AA23
plaintext[121],PIN_AA24
plaintext[120],PIN_AA25
plaintext[119],PIN_AA26
plaintext[118],PIN_AB1
plaintext[117],PIN_AB2
plaintext[116],PIN_AB3
plaintext[115],PIN_AB4
plaintext[114],PIN_AB23
plaintext[113],PIN_AB24
plaintext[112],PIN_AB25
plaintext[111],PIN_AB26
plaintext[110],PIN_AC1
plaintext[109],PIN_AC2
plaintext[108],PIN_D1
plaintext[107],PIN_D2
plaintext[106],PIN_D3
plaintext[105],PIN_E1
plaintext[104],PIN_E2
plaintext[103],PIN_E3
plaintext[102],PIN_E22
plaintext[101],PIN_E23
plaintext[100],PIN_E24
plaintext[99],PIN_E25
plaintext[98],PIN_E26
plaintext[97],PIN_F1
plaintext[96],PIN_F2
plaintext[95],PIN_F3
plaintext[94],PIN_F4
plaintext[93],PIN_F5
plaintext[92],PIN_F6
plaintext[91],PIN_F20
plaintext[90],PIN_F21
plaintext[89],PIN_F22
plaintext[88],PIN_F23
plaintext[87],PIN_F24
plaintext[86],PIN_F25
plaintext[85],PIN_F26
plaintext[84],PIN_G1
plaintext[83],PIN_G2
<table>
<thead>
<tr>
<th>Plaintext</th>
<th>PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>PIN_P7</td>
</tr>
<tr>
<td>[1]</td>
<td>PIN_P6</td>
</tr>
<tr>
<td>[2]</td>
<td>PIN_P4</td>
</tr>
<tr>
<td>[3]</td>
<td>PIN_P3</td>
</tr>
<tr>
<td>[5]</td>
<td>PIN_M24</td>
</tr>
<tr>
<td>[7]</td>
<td>PIN_M22</td>
</tr>
<tr>
<td>[8]</td>
<td>PIN_M21</td>
</tr>
<tr>
<td>[9]</td>
<td>PIN_M20</td>
</tr>
<tr>
<td>[10]</td>
<td>PIN_M19</td>
</tr>
<tr>
<td>[12]</td>
<td>PIN_M17</td>
</tr>
<tr>
<td>[13]</td>
<td>PIN_M16</td>
</tr>
<tr>
<td>[14]</td>
<td>PIN_M15</td>
</tr>
<tr>
<td>[15]</td>
<td>PIN_M14</td>
</tr>
<tr>
<td>[16]</td>
<td>PIN_M13</td>
</tr>
<tr>
<td>[17]</td>
<td>PIN_M12</td>
</tr>
<tr>
<td>[18]</td>
<td>PIN_M11</td>
</tr>
<tr>
<td>[19]</td>
<td>PIN_M10</td>
</tr>
<tr>
<td>[20]</td>
<td>PIN_M9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cipher</th>
<th>PIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>PIN_AC6</td>
</tr>
<tr>
<td>[101]</td>
<td>PIN_AC5</td>
</tr>
<tr>
<td>[102]</td>
<td>PIN_AC4</td>
</tr>
<tr>
<td>[103]</td>
<td>PIN_AC3</td>
</tr>
<tr>
<td>[104]</td>
<td>PIN_AC2</td>
</tr>
<tr>
<td>[105]</td>
<td>PIN_AC1</td>
</tr>
<tr>
<td>[106]</td>
<td>PIN_AC</td>
</tr>
<tr>
<td>[107]</td>
<td>PIN_A</td>
</tr>
<tr>
<td>[108]</td>
<td>PIN_AA</td>
</tr>
<tr>
<td>[109]</td>
<td>PIN_AA2</td>
</tr>
<tr>
<td>[110]</td>
<td>PIN_AA3</td>
</tr>
<tr>
<td>[111]</td>
<td>PIN_AA4</td>
</tr>
<tr>
<td>[112]</td>
<td>PIN_AA5</td>
</tr>
<tr>
<td>[113]</td>
<td>PIN_AA6</td>
</tr>
<tr>
<td>[114]</td>
<td>PIN_AA7</td>
</tr>
<tr>
<td>[115]</td>
<td>PIN_AA8</td>
</tr>
<tr>
<td>[116]</td>
<td>PIN_AA9</td>
</tr>
<tr>
<td>[117]</td>
<td>PIN_A10</td>
</tr>
<tr>
<td>[118]</td>
<td>PIN_A11</td>
</tr>
<tr>
<td>[119]</td>
<td>PIN_A12</td>
</tr>
<tr>
<td>[120]</td>
<td>PIN_A13</td>
</tr>
<tr>
<td>[121]</td>
<td>PIN_A14</td>
</tr>
<tr>
<td>[122]</td>
<td>PIN_A15</td>
</tr>
<tr>
<td>[123]</td>
<td>PIN_A16</td>
</tr>
<tr>
<td>[124]</td>
<td>PIN_A17</td>
</tr>
<tr>
<td>[125]</td>
<td>PIN_A18</td>
</tr>
<tr>
<td>[126]</td>
<td>PIN_A19</td>
</tr>
<tr>
<td>[127]</td>
<td>PIN_A20</td>
</tr>
</tbody>
</table>

start, PIN_P9
keyout[35],PIN_W16
d coworkers' PIN_W17
keyout[33],PIN_W19
d coworkers' PIN_W21
keyout[31],PIN_W23
d coworkers' PIN_W24
keyout[30],PIN_W25
d coworkers' PIN_W26
keyout[28],PIN_Y1
d coworkers' PIN_Y1
keyout[27],PIN_Y3
d coworkers' PIN_Y4
keyout[25],PIN_Y5
d coworkers' PIN_Y10
keyout[24],PIN_Y11
d coworkers' PIN_Y12
keyout[23],PIN_Y13
d coworkers' PIN_Y14
keyout[22],PIN_Y15
d coworkers' PIN_Y16
keyout[21],PIN_Y17
d coworkers' PIN_Y18
keyout[20],PIN_Y20
d coworkers' PIN_Y21
keyout[19],PIN_Y22
d coworkers' PIN_Y23
keyout[18],PIN_Y24
d coworkers' PIN_Y25
keyout[17],PIN_Y26
d coworkers' PIN_Y27
keyout[16],PIN_Y28
d coworkers' PIN_Y29
keyout[15],PIN_Y30
d coworkers' PIN_Y31
keyout[14],PIN_Y32
d coworkers' PIN_Y33
keyout[13],PIN_Y34
d coworkers' PIN_Y35
keyout[12],PIN_Y36
d coworkers' PIN_Y37
keyout[11],PIN_Y38
d coworkers' PIN_Y39
keyout[10],PIN_Y40
d coworkers' PIN_Y41
keyout[9],PIN_Y42
d coworkers' PIN_Y43
keyout[8],PIN_Y44
d coworkers' PIN_Y45
keyout[7],PIN_Y46
d coworkers' PIN_Y47
keyout[6],PIN_Y48
d coworkers' PIN_Y49
keyout[5],PIN_Y50
d coworkers' PIN_Y51
keyout[4],PIN_Y52
d coworkers' PIN_Y53
keyout[3],PIN_Y54
d coworkers' PIN_Y55
keyout[2],PIN_Y56
d coworkers' PIN_Y57
keyout[1],PIN_Y58
d coworkers' PIN_Y59
keyout[0],PIN_Y60
d coworkers' PIN_Y61
keyready,PIN_E8
Outcomes of this Thesis

