LOW POWER ADVANCED ENCRYPTION STANDARD (AES) IMPLEMENTATION ROBUST AGAINST SIDE CHANNEL ATTACKS

by

Serdar Unal

B.S., Electrical & Electronics Engineering, Boğaziçi University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical & Electronics Engineering
Boğaziçi University
2022

ACKNOWLEDGEMENTS

I would like to thank Assist. Prof. Faik Başkaya for his support and guidance throughout the thesis period. His experience helped me to correct my path and prepare a professional thesis that also conforms to the academic guidelines.

This thesis is a cumulative result of all I have learned during my undergraduate and graduate periods. Therefore, I also would like to thank Electrical & Electronics Engineering department faculty members for their teaching.

My company, TUBITAK BILGEM, allowed me to use ASIC tools and equipped me with the skills that made this thesis more professional. I hail my colleagues from TUTEL whose friendship is very valuable for me. Special thanks to my friend Muhammed Said Seferbey.

I would like to express my gratitude to my father, my mother, and my sister for their all-life support. Finally, I thank my dear wife and companion, Şerife, for her understanding and assistance.

ABSTRACT

LOW POWER ADVANCED ENCRYPTION STANDARD (AES) IMPLEMENTATION ROBUST AGAINST SIDE CHANNEL ATTACKS

As the people around the globe become increasingly connected to each other, the amount of information that flows becomes huge. Unfortunately, this vast information network is vulnerable to harmful attacks. Encryption is a strong tool that has been used for ages to act as a shield against these attacks. Among many algorithms utilized for encryption, one of the most popular is AES. AES is an approximately 20-year old algorithm that has been adopted by many organizations around the world to protect classified and unclassified data. In line with the trend of low power and secure implementations, the main intent of this thesis is to show a low-power AES implementation that is secure against power side-channel attacks. In the RTL, currently unused registers are kept constant to lower the power consumption. Choosing the LP ASIC process, using clock-gating, and preferring standard cells with higher threshold voltages enable more power saving. For the side-channel attack resistance, obfuscating and pipelining are employed. The obfuscating disguises the relation between the processed bits and the power consumption by modifying the processed information. On the other hand, the pipelining mixes power consumption related to different inputs with each other. The different versions of AES implementations are processed through FPGA and TSMC 65 nm ASIC flow to compare with each other. After the power traces are collected and analyzed by ChipWhisperer the side-channel attack resistance is evaluated. The effects of the obfuscating and pipelining in increasing attack resistance are proven after predicting key bytes from power traces stemming from thousands of random inputs. The area, power overheads in return for increased attack resistance are detected.

ÖZET

YAN KANAL SALDIRILARINA DAYANIKLI, DÜŞÜK GÜÇ TÜKETEN AES UYGULAMASI

Dünyadaki insanların birbirleriyle artan miktarda iletişim kurmalarıyla birbirleri arasında bilgi miktarı devasa olmaktadır. Ne yazık ki bu devasa bilgi ağı zararlı saldırılara açıktır. Şifreleme yüzyıllardır bu saldırılara karşı kullanılan güçlü bir araç olmaktadır. Şifreleme için kullanılan pek çok algoritma içinde en popüler olanlardan biri de AES'dir. AES yaklaşık 20 yıldır dünyadaki birçok kurum tarafından gizli ve gizli olmayan bilgileri korumak için kullanılmaktadır. Düşük güç tüketen ve güvenli uygulama eğilimi doğrultusunda bu tezin temel amacı düşük güç tüketen ve yan-kanal saldırılarına dayanıklı bir AES uygulaması göstermektir. RTL seviyesinde anlık olarak kullanılmayan register'ların sabit tutulması sağlanarak güç tüketimi düşürülmektedir. Bunun yanında ASIC prosesleri içinden LP kullanmak, clock-gating kullanmak ve yüksek eşik gerilimine sahip standart hücreler tercih etmek daha fazla güç tasarrufu sağlamaktadır. Yan-kanal saldırılarına dayanıklılık için karıştırma ve boru hattı kullanılmaktadır. Karıştırma işlenen veriyi değiştirerek işlenen veri ile güç tüketimi arasındaki ilişkiyi gizlemektedir. Buna karşın, boru hattı farklı girdilerden kaynaklanan güç tüketimlerini birbirleriyle karıştırmaktadır. Farkı AES uygulaması versiyonları FPGA ve TSMC 65 nm ASIC akışlarından geçirilerek birbirleriyle karşılaştırılmaktadır. Güç izlerinin toplanmasından sonra ChipWhisperer programına sokularak yan-kanal saldırılarına dayanıklılık değerlendirilmektedir. Karıştırma ve boru hattının saldırılara dayanıklılığı arttırması, binlerce rassal girdiden kaynaklanan güç izleriyle şifreleme anahtarının parçalarının tahmin edilmesiyle ispatlanmıştır. Saldırı dayanıklılığının artmasına karşın artan alan ve güç tüketimi bilgileri tespit edilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathrm{GF}(p^k)$	Galois Field with order p^k
V_T	Threshold voltage
x^{-1}	Multiplicative inverse of x
μ	Population mean
σ	Standard deviation
$\bullet$	Multiplication in Galois Field
$\oplus$	XOR operation

LIST OF ACRONYMS/ABBREVIATIONS

AES	Advanced Encryption Standard
AESAVS	Advanced Encryption Standard Algorithm Validation Suite
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BBL	Bridge Boost Logic
BC	Best Case
BRAM	Block RAM
CBC	Cipher Block Chaining Mode
CFB	Cipher Feedback Mode
CMOS	Complementary Metal Oxide Semiconductor
CPA	Correlation Power Analysis
CTR	Counter Mode
CTS	Clock Tree Synthesis
DES	Data Encryption Standard
DPA	Differential Power Analysis
DRC	Design Rule Check
DSP	Digital Signal Processing
ECB	Electronic Codebook Mode
EDA	Electronic Design Automation
EM	Electromagnetic
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Gate Array
GDSII	Graphic Design System
GF	Galois Field
HLS	High-Level Synthesis
I2R	Input-to-Register
IBUF	Input Buffer

I/O	Input / Output
IoT	Internet of Things
IP	Intellectual Property
IV	Initialization Vector
KAT	Known Answer Test
LEF	Library Exchange Format
LFSR	Linear Feedback Shift Register
LIB	Liberty Timing File
LP	Low Power
LT	Low Temperature
LUT	Look-up Table
MATLAB	Matrix Laboratory
MCT	Monte Carlo Test
MI5	Military Intelligence, Section 5
ML	Maximum Leakage
MMMC	Multi-Mode Multi-Corner
MMT	Multi-block Message Test
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MTD	Measurements-to-Disclosure
NBS	National Bureau of Standards
NIST	National Institute of Standards and Technology
OBUF	Output Buffer
OFB	Output Feedback Mode
PGE	Partial Guessing Entropy
P & R	Place and Route
PVT	Process, Voltage, Temperature
RC4	Rivest Cipher 4
RC6	Rivest Cipher 6
ROM	Read Only Memory
RSA	Rivest-Shamir-Adleman
RSM	Rotating S-box Masking

RTL	Register Transfer Level
S-box	Substitution Box
SCA	Side Channel Attack
SDF	Standard Delay Format
SHA	Secure Hash Algorithm
SOBER	Seventeen Octet Byte Enabled Register
SPA	Simple Power Analysis
SPEF	Standard Parasitic Exchange Format
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory
STA	Static Timing Analysis
TC	Typical Case
TCF	Toggle Count Format
TRNG	True Random Number Generator
TSMC	Taiwan Semiconductor Manufacturing Company
VCD	Value Change Dump
WC	Worst Case
WCL	Worst Case Low Temperature
WCZ	Worst Case Zero Temperature
WHS	Worst Hold Slack
WNS	Worst Negative Slack
WPWS	Worst Pulse Width Slack
US	United States

1. INTRODUCTION

The people are communicating with each other at ever-increasing rate & speed. This communication revolution enables us to get our job done in a second which would take a much longer time compared in the past. The benefits are threatened by malicious people who can damage our lives while living far from us. The importance of communication security shifted far beyond the borders of the military realm. Personal information is disclosed, social media accounts are accessed to spread false information, bank accounts are emptied, and confidential details are compromised. Security is critical to prevent these activities and ensure the health of the communication.

The information is encrypted to prevent intruders from getting the secrets. If an intruder gets access to the data flowing in the communication channel, he only sees garbage-looking data, not the initial secret. there are two main branches of encryption from the key perspective: symmetric encryption, and asymmetric encryption. The main difference is that the same key is used to encrypt and decrypt the message in symmetric encryption. On the other hand, the decryption key is different from the encryption key in asymmetric encryption. The topic of this thesis, Advanced Encryption Standard (AES) belongs to the symmetric encryption branch. There is also the distinction between block ciphers and stream ciphers. Stream ciphers operate on smaller units and therefore faster. On the contrary, the block ciphers deal with large blocks at a time. The AES is an example of the block ciphers.

The AES was chosen after a competition, organized by the National Institute of Standards and Technology (NIST). The goal of the competition was to select a royalty-free, publicly open algorithm which is capable to protect sensitive data well in the new century. The Rijndael algorithm, developed by Belgian cryptographers Vincent Rijmen and Joan Daemen, was selected as AES and officially published as Federal Information Processing Standards (FIPS) 197 in 2001. Since then, AES has been used extensively worldwide and replaced Data Encryption Standard (DES) which

was the previous standard adopted by National Bureau of Standards (NBS) in the past. NBS is the previous name of the NIST.

AES algorithm consists of rounds whose number depends on the key length. In rounds, four essential operations are performed: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The plaintext undergoes a series of operations involving the round keys and the ciphertext is obtained in the end. Round keys are obtained with the Key Expansion routine using the initial cipherkey. Three key length options are available: 128-bit, 192-bit, and 256-bit. There are also different confidentiality operation modes of AES to improve security by associating encryption of independent data blocks.

Numerous papers have been published about improving and breaking AES, [1] but AES was able to be resistant against any practical algebraic attack. However, the implementations turned out to be vulnerable to side-channel attacks. The side-channel attacks break the algorithms by analyzing leaks stemming from implementations instead of attacking directly to the cipher. Thereby, they bypass the mathematical foundations of the algorithms. Analyzing fluctuations in the power consumption, Electromagnetic (EM) radiations, difference in operation time, emitted sounds can give clues about the secret and can result in disclosure of the cipherkey.

We propose a special AES implementation in this thesis. One of the two main targets is having a low-power design. In general, low power consumption and side-channel resistance are contradictory objectives. Therefore, the low-power objective should be interpreted as relatively low-power. Techniques in the RTL and in the Application Specific Integrated Circuit (ASIC) / Field Programmable Gate Array (FPGA) flows were employed to reduce power consumption and to offset power consumption stemming from the introduction of the side-channel attack countermeasures. The second target is resistance to the power side-channel attacks. Obfuscating and pipelining are introduced to prevent power side-channel attacks. The AES was passed through the Taiwan Semiconductor Manufacturing Company (TSMC) 65 nm ASIC flow, from RTL to GDSII. The efficiency of the countermeasures was analyzed with ChipWhisperer

tool after power traces corresponding to the different inputs were collected from the Cadence Voltus program. To sum up, the contributions from thesis can be summarized as:

- The effect of the pipelining as a countermeasure against power side-channel attacks was examined and compared with regular AES implementations and obfuscating. It has been shown that pipelining improves side-channel attack resistance in addition to performance improvement.
- The side-channel attack resistances of different AES implementations were evaluated using simulation programs after they had gone through the ASIC implementation flow. This enables evaluating side-channel attack resistance before a costly and long tape-out process.

The thesis is constructed as follows. Chapter 2 summarizes the background of the thesis topic. Chapter 3 describes the RTL blocks of the design and some RTL tricks used to reach the objectives of thesis. FPGA design flow steps in Xilinx Vivado and their details are explained in Chapter 4. ASIC counterpart of Chapter 4 is discussed in Chapter 5 using Cadence Electronic Design Automation (EDA) tools. Chapter 6 evaluates the design according to the defined metrics and compares this work with existing literature. Chapter 7 concludes by giving a summary of what has been achieved.

2. BACKGROUND

In this chapter, some important concepts regarding this thesis are explained so that the following chapters will be easier to understand.

2.1. Cryptology

Cryptology is the science of the secret, in basic terms. The term is confused with "Cryptography" but Cryptology is a broader term that includes both Cryptography and Cryptanalysis, which are explained in Section 2.1.1 and Section 2.1.2, respectively. So, Cryptology is concerned with both hiding and revealing. On the other hand, Cryptography is on the "hiding" side. Throughout history, many "hiding procedures" were developed and many people tried to break these procedures to obtain the secret.

2.1.1. Cryptography

Cryptography is the study of techniques for secure communication between different parties. The origin of the word goes back to Greeks, to the word "kryptos" meaning hidden. Cryptography is an old field that people used for ages to hide their secrets. One of the best-known examples in history is the Caesar cipher. It is named after Roman general and statesman Julius Caesar. In Caesar cipher, every character is replaced with another character with a fixed distance between them according to the alphabet. For example, C is replaced with A, D with B, E with C, etc. Caesar cipher can be regarded as a member of cyclical-shift substitution ciphers. It was effective in securing military messages in its era when a limited number of people were literate. As time passed, the ciphers became more and more sophisticated [2]. Today, cryptography is essential in a world through which confidential information flows continuously. The important applications include e-commerce activities, chip-based payment cards, military messages, digital currencies, confidential company data, etc. The following sections present a classification for different types of cryptographic algorithms.

2.1.1.1. Ciphers. There are three essential terms in the definition of a cipher: plaintext, key, and ciphertext. Cipher is the algorithm that converts plaintext to ciphertext using a key. The way the ciphertext is decrypted results in two main branches in two different categories: symmetric ciphers - asymmetric ciphers, block ciphers - stream ciphers. There are also hash algorithms whose purpose is different from the symmetric & asymmetric ciphers.

2.1.1.2. Symmetric vs Asymmetric Ciphers. The same key is used to encrypt and decrypt messages in symmetric algorithms as depicted in Figure 2.1. On the other hand, different keys are used for encryption & decryption in asymmetric algorithms as visible in Figure 2.2. Though asymmetric ciphers have a more complex and slower implementation, they solve the main burden in symmetric ciphers, namely key distribution. In asymmetric ciphers, there are one public key and one private key for each party in the communication. If party A wants to send information to party B, it encrypts using the public key of party B which is freely available on the internet. Party B decrypts it using its own private key. For authentication purposes in asymmetric encryption, the document is encrypted with the private key of party C and anyone with the public key of party C can be sure that the document was signed by party C [3]. In practice, the keys of symmetric ciphers are distributed with asymmetric encryption. One of the most popular symmetric algorithm is the AES is the main focus of this thesis and is explained in detail in Section 2.3.

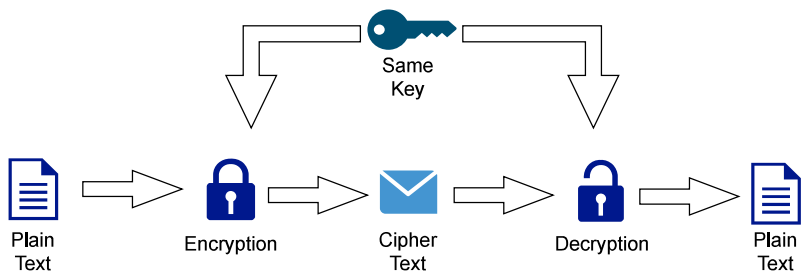


Figure 2.1. Symmetric Encryption.

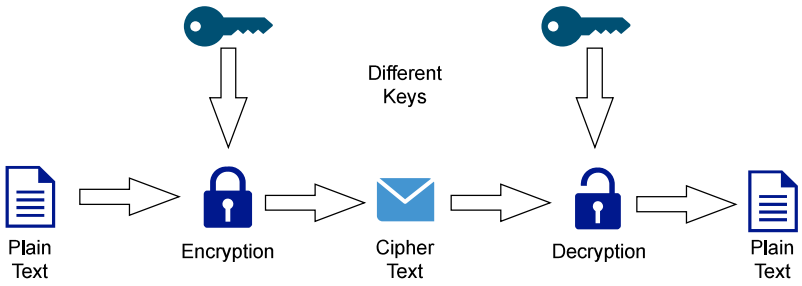


Figure 2.2. Asymmetric Encryption.

2.1.1.3. Block vs Stream Ciphers. The main distinction between block and stream ciphers is the operand size. A large data block consisting of many "unit"s is converted to the ciphertext in block ciphers. On the other hand, in stream ciphers, only one "unit" at a time is converted. The size of the "unit" may vary but it is mostly 8-bit. Processing smaller blocks enables stream ciphers to operate faster and isolate errors between different units. Block ciphers are slower as in general they collect all units to start encryption/decryption. Errors can be propagated to other units. However, in block ciphers, it is harder to infiltrate between units without being detected. Block ciphers are also stronger in encryption as the result is dependent on many units [4]. Examples for stream ciphers include Rivest Cipher 4 (RC4), Seventeen Octet Byte Enabled Register (SOBER), and A5/1. On the other hand, AES, DES, Rivest Cipher 6 (RC6), MISTY-1, and Camellia can be given as instances for block ciphers [5].

2.1.1.4. Hash. Hashing is different from encryption in a sense that a hashed message is not intended to be decrypted. Its simplified block diagram can be seen in Figure 2.3. A famous example is Secure Hash Algorithm (SHA), which digests input plaintext into a small, fixed-sized output. A good hash function returns completely different results for slightly different inputs (avalanche effect) and is infeasible to deduce input value from the hash value. These algorithms are used in application areas such as databases where critical information is stored as hashed values and compromised data give no information about the owner of the plaintext. On the other hand, the users, whose credentials are authenticated with the hash value, can access the system easily.

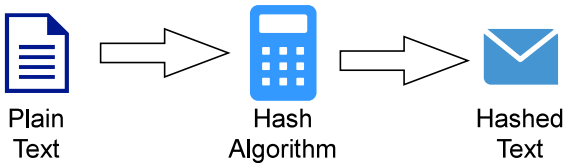


Figure 2.3. Hashing.

2.1.2. Cryptanalysis

Cryptanalysis is a field interested in breaking protections set by Cryptography. Cryptographic algorithms are tested and secrets are compromised by Cryptanalysis. The term encompasses finding the weakness in both the mathematical properties of the algorithm and the implementation of the algorithm. The side-channel attacks stay on the implementation side and they are examined in detail in Section 2.5. There are many different attacks against cryptographic algorithms. The amount of time, storage, and data required to perform an attack are used to characterize the attacks. The attacks that require an infeasible amount of time and storage might have been infeasible in the past but they may be feasible now due to technological advancements. Therefore, a secure algorithm thay may be insecure in the future may need to be replaced by another algorithm. The attacks can be classified into five groups with respect to the amount of information the attacker has.

- Ciphertext-only: The attacker has only access to the collection of ciphertexts.
- Known-plaintext: Collection of arbitrary plaintexts and their corresponding ciphertexts are known.
- Chosen-plaintext (chosen-ciphertext): Similar to the known-plaintext attacks but the collection is chosen by the attacker, i.e not arbitrary.
- Adaptive chosen-plaintext: The collection is again chosen by the attacker but chosen adaptively according to previously obtained information.
- Related-key attacks: The attacker can obtain ciphertexts from the chosen plaintexts under different keys. Keys are not known to the attacker but the relation between keys is known, such as there is only a single bit difference between keys.

2.2. Galois Field

The Galois field is a field with a finite number of elements. The name comes from French mathematician Évariste Galois. The fields support addition, subtraction, multiplication, and division on their finite number of elements. There are different Galois fields with order p^k where p is a prime number and k is a positive integer. p is called the characteristic of the field. The field is used in AES operations, hence it was added as a separate section in Chapter 2. The simplest example of the Galois field is GF (2^1). It has two elements: zero and one. Addition and multiplication in the GF (2^1) are given in Table 2.1 and in Table 2.2, respectively. Addition is basically XOR operation and subtraction is also equal to addition because x = -x by definition in GF (2^1). For multiplication, operands are multiplied and modulo 2 operation is performed on the product to give the final result [6]. In the GF (2^1), multiplication turns out to be AND operation.

The additive inverse of an element is an element where these two elements are added and the addition result modulo p^k becomes 0. The multiplicative inverse of an element is an element where the two elements are multiplied and the multiplication result modulo p^k becomes one. So, these values can be extracted directly from addition and multiplication tables. The Extended Euclidean algorithm can be used to find the multiplicative inverse of an element in the Galois field [6]. Inverses for GF (2^1) are in Table 2.3. There is no multiplicative inverse of zero.

Table 2.1. GF (2^1) Addition.

+	**0**	**1**
0	0	1
1	1	0

Table 2.2. GF (2^1) Multiplication.

•	**0**	**1**
0	0	0
1	0	1

Table 2.3. GF (2^1) Inverses.

Element (x)	**0**	**1**
Additive Inverse (-x)	0	1
Multiplicative Inverse (x^{-1})	-	1

Another example for Galois field is GF (2^8) that is widely used in cryptography as the elements can be represented by 8 bits, i.e bytes. Representing the data as a vector in the Galois field allows easily manipulating the data mathematically. AES SubBytes and MixColumns operations utilize GF (2^8). How they utilize is explained in more detail in Section 2.3.2.2 and in Section 2.3.2.4.

2.3. AES Algorithm

AES is the algorithm made by Belgian cryptographers Vincent Rijmen and Joan Daemen for the competition organized by NIST. After its victory in the competition, it has been used extensively worldwide. AES is a symmetric encryption algorithm and involves sub-algorithms called repeatedly. The key, which is often generated using random number generators (RNG), is utilized after processing by the Key Expansion algorithm. The original version of AES processes data block by block independently. Later, new operation modes were added to improve security. After nearly 20 years, AES with proper implementation is still considered secure according to the cryptography community.

2.3.1. History

National Institute of Standards and Technology (NIST) was seeking an algorithm that would be used by the United States (US) government and voluntarily by others. The algorithm had to be open so that the security would not rely on the secrecy of the operations. Its mathematical background needed to be strong so that the openness of the algorithm would not pose a security threat. In 1997, NIST made an official call to develop Advanced Encryption Standard (AES). Fifteen algorithms were qualified for the first round. After the first round, five algorithms remained: MARS, RC6, Rijndael, Serpent, and Twofish. After the second round, Rijndael was chosen to become AES. The AES was published as Federal Information Processing Standards (FIPS) in 2001. The NIST develops standards and guidelines called FIPS to be used by the US government.

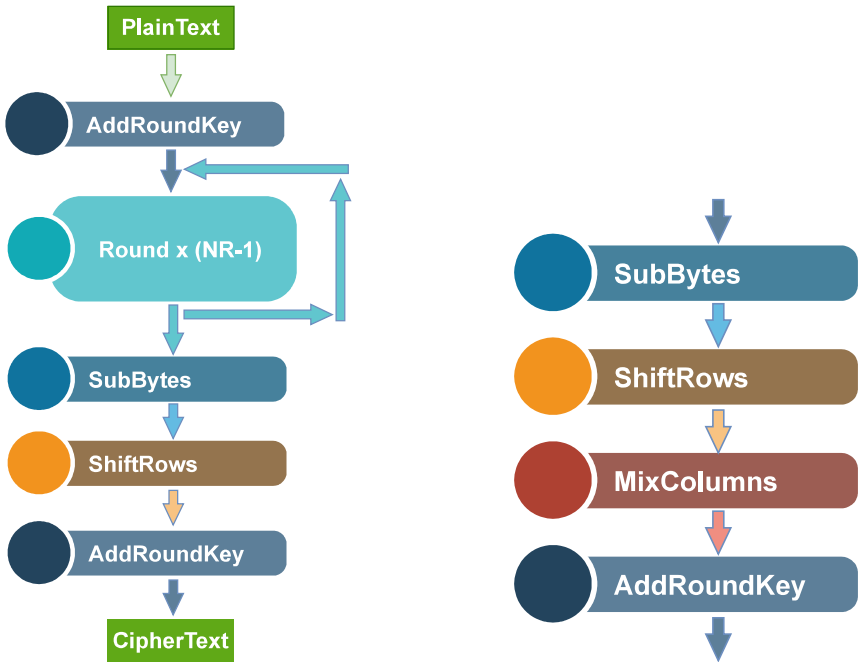
2.3.2. Algorithm Definition

AES has four main operations: AddRoundKey, SubBytes, ShiftRows, and MixColumns. These operations repeatedly applied with an amount determined by the version AES, as given in Table 2.4. This thesis focuses on only AES-128.

Table 2.4. AES Versions.

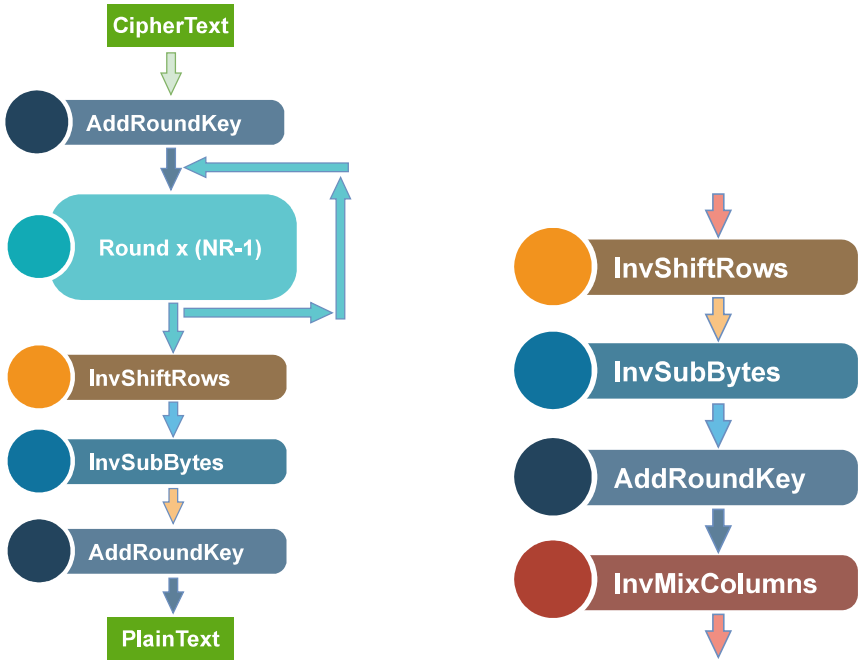
	AES-128	**AES-192**	**AES-256**
Key Length (bits)	128	192	256
Block Size (bits)	128	128	128
Number of Rounds (NR)	10	12	14

Apart from the complete rounds, which contains all of the four operations, there are also individually applied operations as depicted in Figure 2.4. In total, there are 11 AddRoundKey operations, 10 SubBytes operations, 10 ShiftRows operations, and 9 MixColumns operations for AES-128 encryption.



(a) (b)

Figure 2.4. (a) Encryption. (b) Encryption Round.



(a) (b)

Figure 2.5. (a) Decryption. (b) Decryption Round.

Decryption contains the inverse of the mentioned operations as can be seen in Figure 2.5. The inverse of the AddRoundKey is also AddRoundKey since it merely involves XOR. This thesis implements only encryption. Therefore, the remaining inverse operations were not explained here.

2.3.2.1. AddRoundKey. The round key is bitwise XOR'ed with the state array to give the result. Round keys are obtained by the Key Expansion routine from the initial key as explained in Section 2.3.3. The round key and the state array have the same number of bytes. The formula of the operation is given as

$$\mathbf{b}_{i,j} = \mathbf{a}_{i,j} \oplus \mathbf{k}_{i,j} \tag{2.1}$$

whose operands can be seen in Figure 2.6.

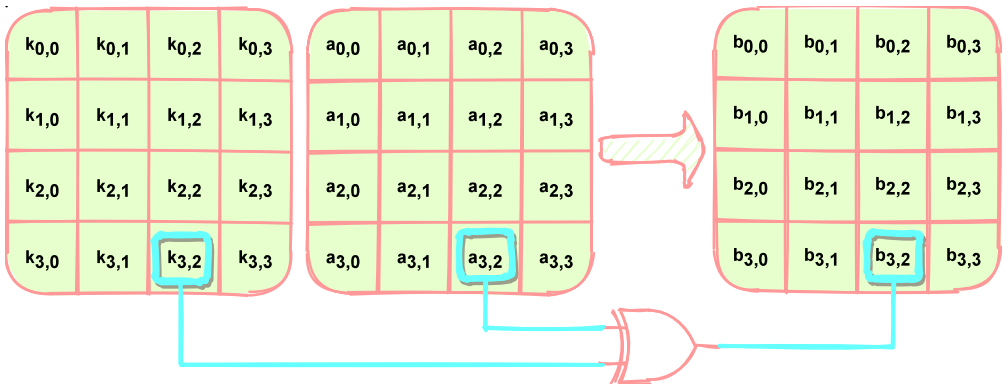


Figure 2.6. AddRoundKey.

2.3.2.2. SubBytes. The SubBytes stage is the main source of nonlinearity in AES. The operation is depicted in Figure 2.7. The bytes from the state array are substituted according to the table called Substitution Box (S-box) as visible in Figure 2.8 with the information from [7]. The least significant part of the input byte determines the column and the most significant part of the input byte determines the row of the S-box. The value at the intersection of the relevant row and column is used. For example, 0x83

is replaced by 0xEC which is at the intersection of the eighth row and third column. S-box can directly be implemented as a Look-up Table. The size of the 16x16 Look-up Table with one-byte entries becomes 2048 bits.

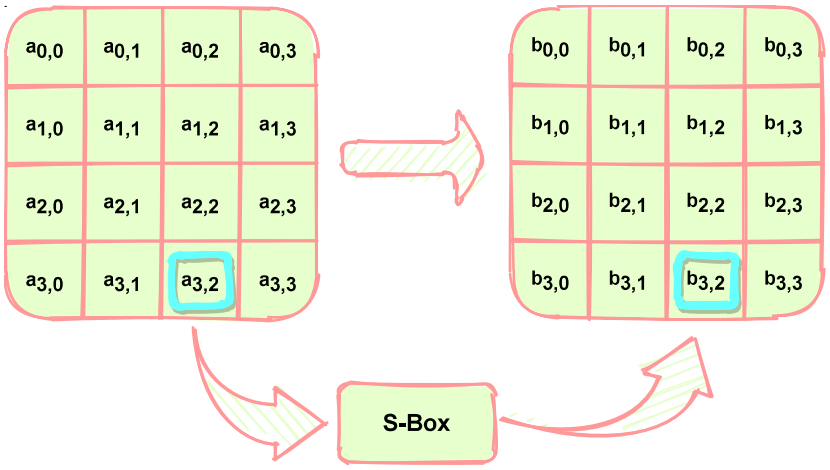


Figure 2.7. SubBytes.

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

Figure 2.8. S-box.

Operations in the Galois field are used to populate the S-box. So, instead of using look-up tables, the conversion operation can be implemented. Implementation is more difficult compared to the look-up tables but there is a room for optimizations. In fact, this is an important research area where S-box implementation is tried to be optimized. Conversion operation consists of two steps:

- Finding multiplicative inverse: Multiplicative inverse of the byte is found according to the GF (2^8) where 00 is mapped to itself as there is no multiplicative inverse of zero in Galois field as mentioned in Section 2.2.
- Applying affine transformation over GF (2^8): Multiplicative inverse of the byte goes through the affine transformation over GF (2^8). The affine transformation,

$$b_i^{'} = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+7)mod8} \oplus c_i \tag{2.2}$$

  is employed to convert byte to transformed byte. $b_i^{'}$ is the i^{th} bit of the output. b_i is the i^{th} bit of the input. c_i is the i^{th} bit of the 8'h63.

  The matrix representation of the affine transformation is as

$$\begin{pmatrix} b_0^{'} \\ b_1^{'} \\ b_2^{'} \\ b_3^{'} \\ b_4^{'} \\ b_5^{'} \\ b_6^{'} \\ b_7^{'} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \tag{2.3}$$

where second added vector is $(01100011)_2 = (99)_{10} = (63)_{16}$.

2.3.2.3. ShiftRows. The bytes in the state array are cyclically shifted in this operation as given in Figure 2.9. The first row of the 4x4 matrix stays the same. The second row is cyclically shifted to the left by one byte, the third row by two bytes, and the fourth row by three bytes. This stage prevents AES from operating on different columns independently.

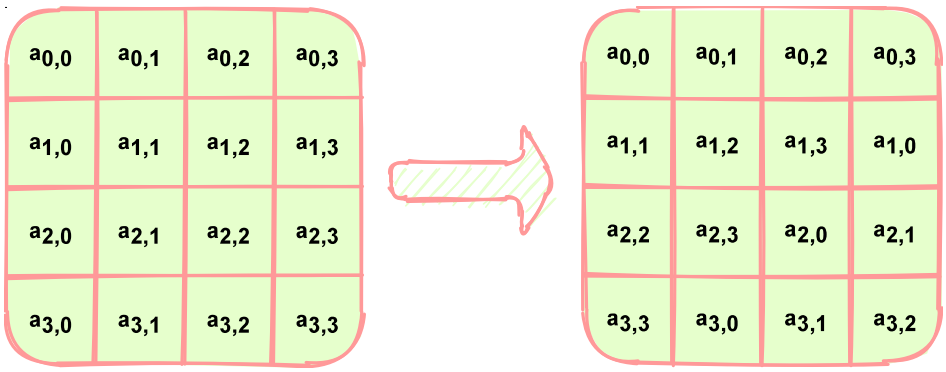


Figure 2.9. ShiftRows.

2.3.2.4. MixColumns. MixColumns is the key source of diffusion in the algorithm. The operation in the state array format is in Figure 2.10. Each of the four columns is multiplied by the array,

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \tag{2.4}$$

to give the output. However, the matrix multiplication is not ordinary multiplication, instead, multiplicands are elements of Galois field GF (2^8). Addition changes to XOR operation in GF (2^8).

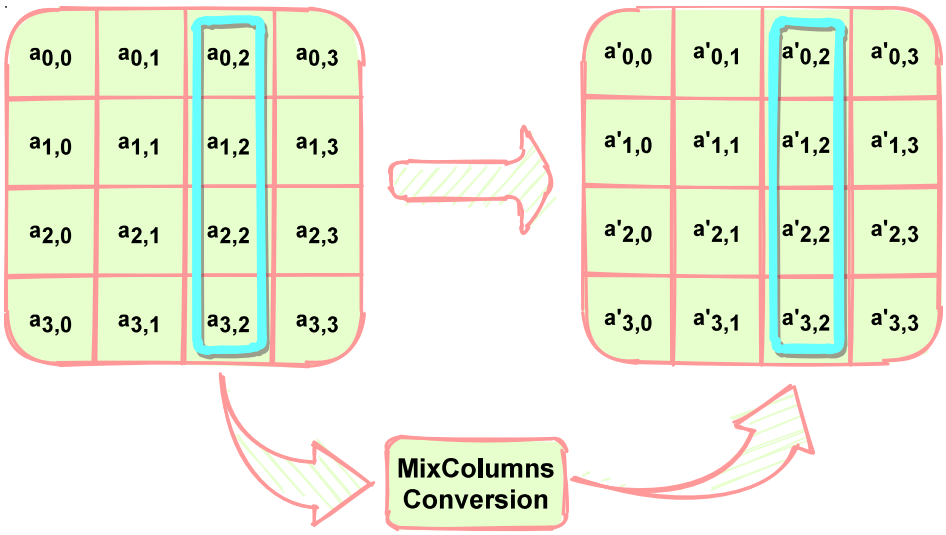


Figure 2.10. MixColumns.

The operations are summarized for one column as follows:

$$
\begin{aligned}
a'_{0,2} &= (\{02\} \bullet a_{0,2}) \oplus (\{03\} \bullet a_{1,2}) \oplus (\{01\} \bullet a_{2,2}) \oplus (\{01\} \bullet a_{3,2}) \\
a'_{1,2} &= (\{01\} \bullet a_{0,2}) \oplus (\{02\} \bullet a_{1,2}) \oplus (\{03\} \bullet a_{2,2}) \oplus (\{01\} \bullet a_{3,2}) \\
a'_{2,2} &= (\{01\} \bullet a_{0,2}) \oplus (\{01\} \bullet a_{1,2}) \oplus (\{02\} \bullet a_{2,2}) \oplus (\{03\} \bullet a_{3,2}) \\
a'_{3,2} &= (\{03\} \bullet a_{0,2}) \oplus (\{01\} \bullet a_{1,2}) \oplus (\{01\} \bullet a_{2,2}) \oplus (\{02\} \bullet a_{3,2}).
\end{aligned}
$$

4x4 matrix is multiplied by 4x1 vector to give 4x1 vector. The first row of the array (2.4) is multiplied by the vector to give the first element of the 4x1 vector. The rest follows the suit. $\oplus$ is XOR operation and $\bullet$ shows multiplication in the Galois field.

An example for MixColumns operation is given as

$$
\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}
\begin{pmatrix} d4 & e0 & b8 & 1e \\ bf & b4 & 41 & 27 \\ 5d & 52 & 11 & 98 \\ 30 & ae & f1 & e5 \end{pmatrix}
=
\begin{pmatrix} 04 & e0 & 48 & 28 \\ 66 & cb & f8 & 06 \\ 81 & 19 & d3 & 26 \\ e5 & 9a & 7a & 4c \end{pmatrix}. \qquad (2.5)
$$

Here, the first element of the output matrix $(04)_{10}$ is calculated for demonstration:

$$
\begin{aligned}
a'_{0,0} &= (\{02\} \bullet a_{0,2}) \oplus (\{03\} \bullet a_{1,2}) \oplus (\{01\} \bullet a_{2,2}) \oplus (\{01\} \bullet a_{3,2}) \\
&= (\{02\} \bullet d4) \oplus (\{03\} \bullet bf) \oplus (\{01\} \bullet 5d) \oplus (\{01\} \bullet 30) \\
&= ((x) \bullet d4) \oplus ((x+1) \bullet bf) \oplus ((1) \bullet 5d) \oplus ((1) \bullet 30) \\
&= ((x) \bullet d4) \oplus ((x+1) \bullet bf) \oplus (5d) \oplus (30) \\
&= (x \bullet (x^7 + x^6 + x^4 + x^2)) \oplus ((x+1) \bullet (x^7 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0)) \oplus \\
&(x^6 + x^4 + x^3 + x^2 + x^0) \oplus (x^5 + x^4) \\
&= (x^8 + x^7 + x^5 + x^3) \oplus (x^8 + x^7 + x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 1) \oplus \\
&(x^6 + x^4 + x^3 + x^2 + 1) \oplus (x^5 + x^4) \\
&= (x^8 + x^7 + x^5 + x^3) \oplus (x^8 + x^7 + x^6 + 1) \oplus (x^6 + x^4 + x^3 + x^2 + 1) \oplus (x^5 + x^4) \\
&= (x^7 + x^5 + x^4 + 2x^3 + x + 1) \oplus (x^7 + x^6 + x^4 + x^3 + x) \\
&\oplus (x^6 + x^4 + x^3 + x^2 + 1) \oplus (x^5 + x^4) \\
&= (x^7 + x^5 + x^4 + x + 1) \oplus (x^7 + x^6 + x^4 + x^3 + x) \oplus (x^6 + x^4 + x^3 + x^2 + 1) \\
&\oplus (x^5 + x^4) \\
&= (10110011) \oplus (11011010) \oplus (01011101) \oplus (00110000) \\
&= (0000100) = (04)_{10}.
\end{aligned}
\tag{2.6}
$$

[02 03 01 01] vector is multiplied by [d4 bf 5d 30] elementwise and the products are XOR'ed. 02 is replaced by x, 03 is replaced by x+1, and 01 is replaced by 1. Elements from [d4 bf 5d 30] are replaced by their polynomial equivalent where ff means $x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0$. The polynomials are multiplied. Multiplying by 02 shifts the element to the left by a bit. 03 shifts by a bit and adds the initial element. 01 does not change the element. Terms with a coefficient of 2 are eliminated and coefficient terms with coefficient -1 are replaced by 1 throughout the process. Then the terms are divided with the special polynomial $x^8 + x^4 + x^3 + x^1 + x^0$ to obtain a remainder, i.e modulo operation. By doing this, products are reduced to bytes. The resulting terms are converted again to binary representation and XOR'ed to obtain the result.

2.3.3. Key Expansion

Key Expansion routine is used to obtain round keys from the initial cipher key. These round keys are required to use in AddRoundKey operations during the encryption process. The process is explained with the help of the pseudo-code formed using the information from [7] for AES-128 and depicted in Figure 2.11. SubWord and RotWord functions can be seen in the following subsections. Rcon is a 32-bit round constant array [rc_i 00_{16} 00_{16} 00_{16}] with the rc_i values given below in Table 2.5.

KeyExpansion (byte key[16], word w[44]) **begin**
word temp
$i = 0$
while $i < 4$ **do**
  w[i] = word(key[$4 \times i$], key[$4 \times i + 1$], key[$4 \times i + 2$], key[$4 \times i + 3$])
  $i = i + 1$
end while
$i = 4$
while $i < 44$ **do**
  temp = w[$i - 1$]
  temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
  w[i] = w[$i - 4$] xor temp
  $i = i + 1$
end while
end

Figure 2.11. Key Expansion.

2.3.3.1. SubWord. Four-byte input is taken and bytes are converted individually to outputs using S-box table in Figure 2.8 similar to the SubBytes operation.

Table 2.5. rc_i Values (hex).

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
rc_i	01	02	04	08	10	20	40	80	1B	36

2.3.3.2. RotWord. RotWord takes four-byte input and performs one-byte cyclical shift on it. The operation is depicted in Figure 2.12.

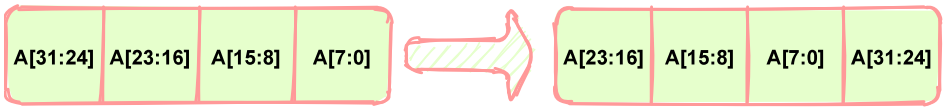


Figure 2.12. RotWord.

2.3.4. Confidentiality Operation Modes

There are different confidentiality operation modes as specified in NIST Special Publication 800-38A. According to the document [8], these are not specific to AES, they can be used with any of the FIPS-approved symmetric key block cipher algorithms.

- Electronic codebook mode (ECB)
- Cipher block chaining mode (CBC)
- Output feedback mode (OFB)
- Cipher feedback mode (CFB)
- Counter mode (CTR)

ECB corresponds to the essential AES implementation. Data blocks are processed independently from each other. Therefore, parallel processing is possible. If the key is the same, the same input always gives the same output. This may be a security problem for particular use cases. It is not mandatory to use other modes but they

are recommended by NIST. CBC, OFB, and CFB relate encryption of a data block to encryption of the previous block. The security is improved but parallel processing, like ECB, is prevented. CBC, OFB, and CFB use unpredictable Initialization Vector (IV) in the encryption process in addition to the key. In the CTR, counter blocks go through the cipher and its output is XOR'ed with the plaintext to give ciphertext. If the counter blocks are known, the data blocks can be processed independently.

2.3.5. Security of AES

AES is fairly secure in terms of algorithmical foundations. It has good linear and differential cryptanalysis resistance [9]. Its minimum 128-bit key makes brute-force attacks infeasible. There have been papers about related-key attacks on AES but they do not undermine the security of AES since a properly implemented algorithm would not allow related keys to be used. The first key-recovery attack reduced needed operations to approximately $(\frac{1}{4})$th but still a huge number of operations, $\approx 2^{126}$ for a 128-bit key, are needed [10]. Therefore, attacks targeting the algorithm itself are infeasible. However, the side-channel attacks pose threat to the security of AES implementations and reduce the number of operations to a much smaller number. The side-channel attacks and countermeasures are presented in Section 2.5

2.4. AES Validation

The AES implementations must be verified before going into products. NIST The Advanced Encryption Standard Algorithm Validation Suite (AESAVS) is an important test suite used for validation purposes. The completion of the tests in the suite indicates that the implementation conforms to the "FIPS 197: Advanced Encryption Standard". There are more than 5700 validated AES algorithm implementations, as of 2020. The list of implementations validated by NIST can be found in [11]. The test suite consists of three main parts: the Known Answer Test (KAT), the Multi-block Message Test (MMT), and the Monte Carlo Test (MCT). For KAT, The document contains hundreds of plaintext-key-ciphertext groups separated into different key lengths. GFSbox,

KeySbox, Variable Key, and Variable Text types are joined to give KAT. The values are selected to test the algorithm at the corner cases. The multi-block input processing capability of the implementation is measured with the MMT but little detail is given in AESAVS for it compared to the other ones. In contrast to the KAT, plaintext-key-ciphertext groups are obtained using an algorithm in MTC. The algorithm starts with an initial plaintext and key values. Then encryption is performed many times using the output of the encryption as input. Periodically the key is updated [12].

2.5. Side Channel Attack (SCA)

In side-channel attacks, the attacker tries to obtain leaked information from the implementation of the algorithm. For cryptographic algorithms like AES mathematical properties are strong enough to make brute-force attacks infeasible due to limitations on time and computational power. In such cases, side-channel attacks are more dangerous and effective on the improper implementations. The attacker can make inferences from the power consumption of the circuit, the radiated electromagnetic waves, emitted sounds, and lights, etc as shown in Figure 2.13. A famous example is the spying on the Egyptian embassy in London by British domestic intelligence agency MI5. MI5 placed a microphone near to rotors used in the crypto device and this additional information coming from the rotor clicks helped them to break cipher used by Egyptians [13]. Power side-channel attacks were examined and executed as a part of this thesis.

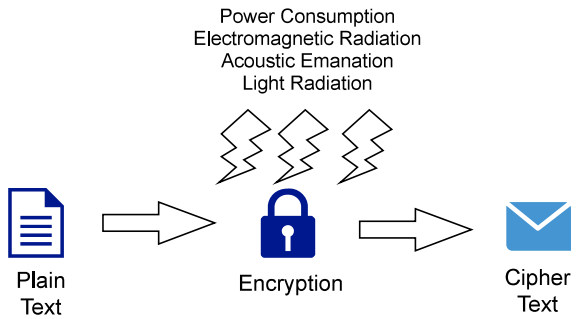


Figure 2.13. Side Channel Attacks.

2.5.1. Power Analysis Attacks

Power analysis attacks are interested in power consumed by the circuit implementing the cryptographic algorithm. The power is proportional to α which is the activity factor as shown in,

$$\mathbf{P}_{static} = \mathbf{I}_{static}\mathbf{V}_{DD} \tag{2.7}$$

$$\mathbf{P}_{dynamic} = \alpha\mathbf{C}\mathbf{V}_{DD}^2\mathbf{f} \tag{2.8}$$

$$\mathbf{P}_{total} = \mathbf{P}_{static} + \mathbf{P}_{dynamic} \tag{2.9}$$

where P = power, I = current, V_DD = supply voltage, α = activity factor, C = load capacitance, f = switching frequency. The activity factor is basically a toggle rate. If the circuit consumes higher power, that means more bits are switching. Comparing between different time instants and traces gives insight into the processed secret information.

2.5.1.1. Simple Power Analysis (SPA). Simple Power Analysis (SPA) primarily deals with major power fluctuations and visual inspection is employed. If the effect of the data is clearly visible on the power trace, then SPA is enough to break the algorithm. The attack does not work well in noisy traces as noise veils difference between time instants. SPA is easy to implement and a low-cost attack but can also be countered easily. Key dependent Rivest-Shamir-Adleman (RSA) multiply and square operations can be an example for to be a victim of SPA [14].

2.5.1.2. Differential Power Analysis (DPA). Differential Power Analysis (DPA) is a more sophisticated attack compared to the SPA and introduced to the literature by Kocher in [15]. The attacker uses hypothetical model of the device to predict the side channel information at the relevant stage. Then, the predictions are compared with the real measurement values by means of statistical methods [16]. When many traces exist, the attack can reveal the secret after analyzing tiny differences between traces.

Signal processing is used to improve traces so that the attack works also for noisy traces in contrast to SPA. There are also many published DPA attacks on AES.

2.5.1.3. Correlation Power Analysis (CPA). Correlation Power Analysis (CPA) is a technique initially developed by Brier [17]. It is the attack type employed for this thesis and utilizes the Pearson correlation function for differentiating traces [18]. The function can be shown as

$$C(T,P) = \frac{\mu(TP) - \mu(T)\mu(P)}{\sqrt{\sigma^2(T)\sigma^2(P)}} \tag{2.10}$$

where T = set of power traces, P = set of estimated power values from the power model, μ = population mean, σ = standard deviation. The power model is used to model power consumption theoretically. One of the simplest power models is the Hamming Distance (HD) model. The number of bit changes, both from zero to one and one to zero, is summed to find HD. Another model is the Hamming Weight (HW) in which the number of one's is calculated. After the model is chosen, the Pearson correlation function is utilized to calculate the correlation coefficient of every point in traces. By doing this, the model and the actual traces are compared. This procedure is repeated for different predicted subkeys. Important point is that not all possible key combinations are tried. Key is predicted as part by part. For example, for AES, key is attacked byte by byte. Because for some parts of the AES, key bytes are processed separately, and hence they can be revealed separately. The total number of combinations is reduced from 2^{128} to $16x2^8$. In the end, the best subkey predictions are collected to give the secret key [18].

2.5.2. Power Analysis Countermeasures

As side-channel attacks proved to be dangerous for the algorithms that are assumed to be secure, countermeasures gained more popularity. Different methods were proposed to make the attacks infeasible. Random operations are added among the normal operations to loose the correlation between power traces and the processed values. Keys are changed periodically in order to prevent the collection of enough traces with

the same key and prevent decryption of the new data even if the old key is exposed. Special analog circuitry shows the power consumption nearly constant by supplying the cryptographic circuit from itself. Algorithm operations are shuffled to cause confusion. Power traces are deceived by obfuscating processed data during the operations [13]. The attacker is forced to implement demanding synchronization in order to correct the clock signal which is messed with random jitter [19].

3. RTL CODING OF DESIGN

Register Transfer Level (RTL) coding is the first stage in both FPGA & ASIC flows. There are two main languages for describing hardware, namely VHDL and Verilog. SystemVerilog can be viewed as an extended version of Verilog, especially for verification purposes. Since it is backward compatible, Verilog files can be used with SystemVerilog after just changing the extension from .v to .sv. In this project, SystemVerilog was used. Columns on which the RTL was constructed are explained in the following section. Subsequently, what was done in RTL to achieve the main objectives of the project are given.

3.1. Structure

The top-level connections in the RTL can be summarized as in Figure 3.1. The key expansion routine takes the CipherKey as an input and calculates the RoundKeys that are used in encryption rounds. KeyReady signal is used to notify the Encryption module so that it can use the incoming RoundKeys in the encryption process. This prevents the generation of unnecessary intermediate results when the CipherKey has been modified but new RoundKeys have not been computed yet.

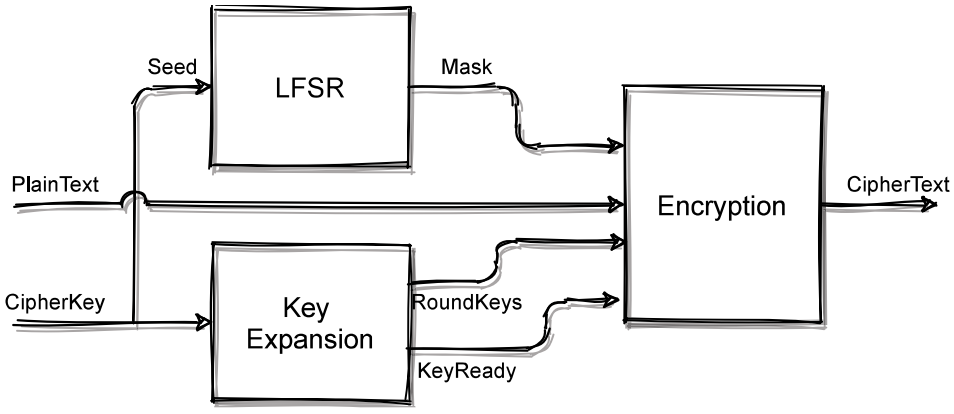


Figure 3.1. Top Level Connections.

Linear Feedback Shift Register (LFSR) is used for obtaining the Mask, which is utilized as a side-channel countermeasure. A 128-bit shift-register was utilized to implement the shift register as can be seen in Figure 3.2. The taps are on 99, 101, 126, and 128. The shift register is fed from the combinational processing of the outputs of the tap registers. The numbers were chosen to maximize the repeat period of the LFSR [20]. Obfuscating using the Mask, is explained in more detail in Section 3.3.

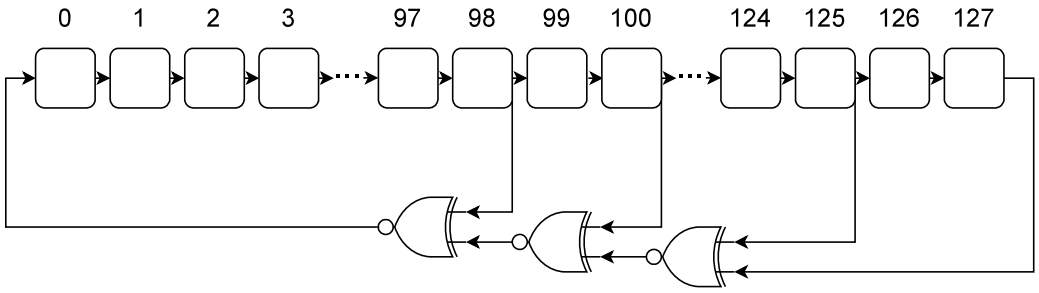


Figure 3.2. LFSR.

There are five different implementation cases in this thesis. The first one is the most compact one, the rolled, which uses hardware of only one round for all middle rounds. The round output is registered and fed back to the input of the round to continue the multi-round process. This version results in little hardware usage but, it takes 17 clock cycles after beginning of the loading of plaintext and before the last bytes of ciphertext become visible. Both the loading of plaintext and the extraction of ciphertext from the 32-bit interface need four cycles, which is the case for all versions except the 64-bit pipelined version. The next version is the obfuscated version of the first case. The latency in terms of clock cycles does not change but hardware resource usage and power consumption increase as additional operations are performed to prevent side-channel attacks. The next two versions are 32-bit and 64-bit pipelined versions. The difference between the pipelined and non-pipelined versions is that the pipelined hardware does not stay idle as new inputs are taken to the processing before the previous one is completed. This increases the throughput. In addition, the pipelin-

ing complicates the relationship between the input and the power trace as multiple different inputs are processed at the same time such that their power consumption mix with each other. The last version is the unrolled version, where again plaintext is received and ciphertext is returned from the 32-bit interface but the encryption process is combinational. In the unrolled version, each round is expanded as a separate block. 17 clock cycle latency of rolled version becomes eight cycles with this version. However, since more operations are completed in the same clock cycle, the period must be larger compared to the other versions. The block diagrams of the rolled, rolled obfuscated, and unrolled versions are:

- AddRoundKey
- Round x (NR-1)
  - SubBytes
  - ShiftRows
  - MixColumns
  - AddRoundKey
- SubBytes
- ShiftRows
- AddRoundKey

The structures of the pipelined versions are different. The structure of the 32-bit pipelined version can be examined in Figure 3.3. The middle nine rounds are combined into three main round hardware blocks. The incoming plaintext passes four times through two of them, and one time through the last one. The remaining AddRoundKey, SubBytes, ShiftRows, and RoundKey operations are performed again at the end.

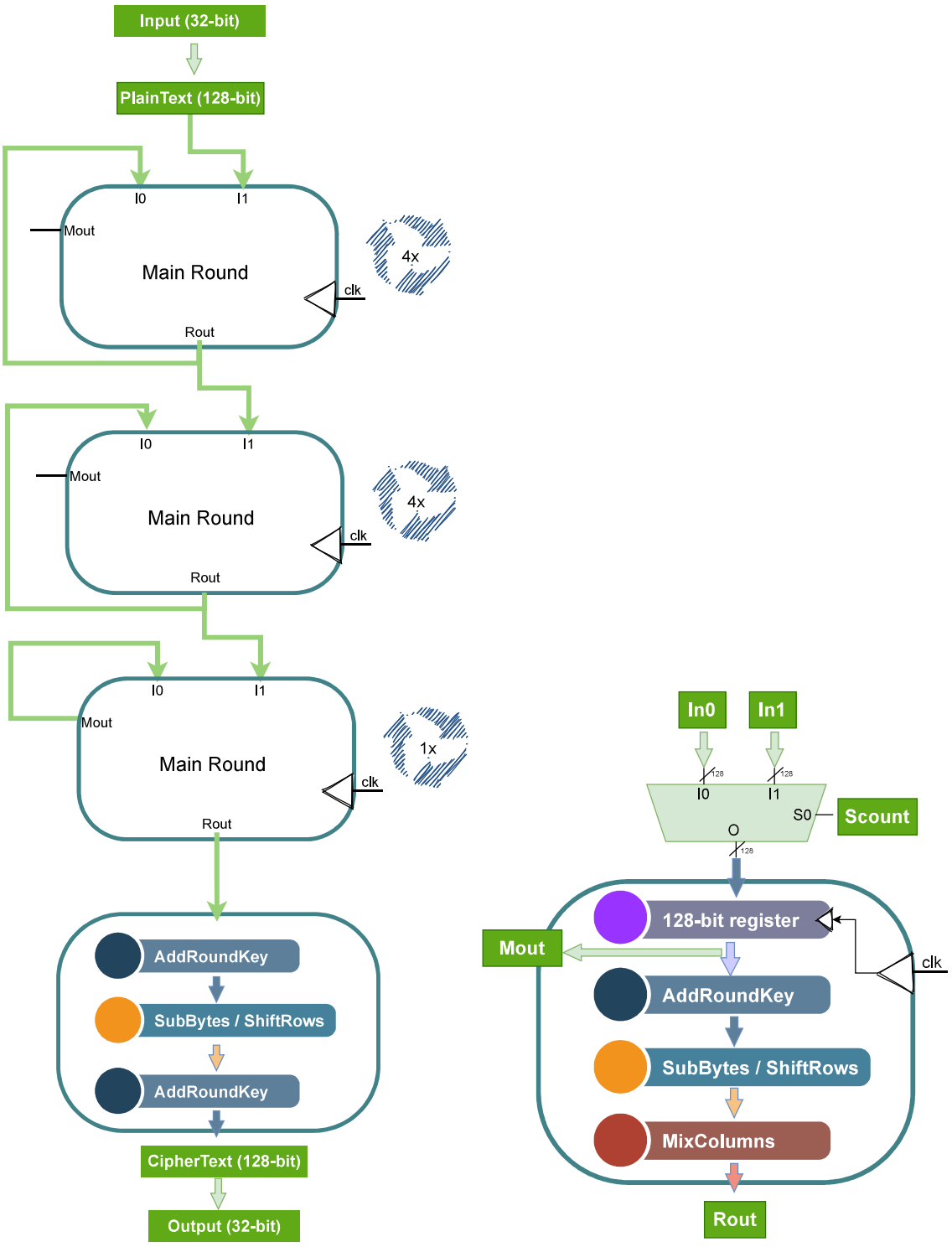


Figure 3.3. (a) Pipeline (32-bit). (b) Pipeline Main Round.

Components of the main round are also visible in Figure 3.3b. This round is slightly different from the original round in a sense that it starts with AddRoundKey and finishes with MixColumns. However, this is just an implementation issue as the first AddRoundKey was incorporated into the first round and the remaining ones shifted due to the initial round. The main round module has a multiplexer in front of it which takes new input from the former stage every four clock cycles, and round output value (Rout) at the remaining cycles. Mout represents the output of the multiplexer and only used in the last main round. Since the last main round block works just once during the four cycles, the last main round keeps its input fixed via Mout. Keeping the 128-bit result of AES fixed for four clock cycles also gives enough time to transfer the 128-bit ciphertext output to the 32-bit bus.

Scount is generated basically by rotating "1000" sequence through four flip-flops to divide the main frequency into four as can be seen in Figure 3.4. There is only one scount module that supplies select bits of the all input MUX modules.

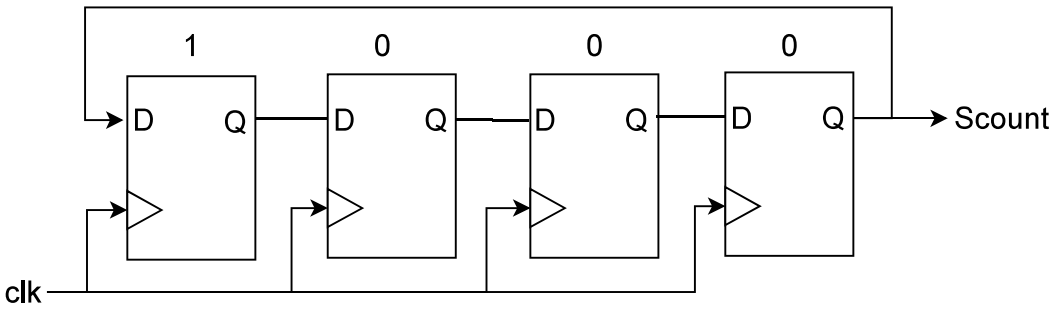


Figure 3.4. Pipeline (32-bit) Scount Module.

The 64-bit pipelined version is slightly modified compared to the 32-bit pipelined version. Since it takes the whole plaintext in two clock cycles instead of four, it can process more inputs simultaneously if the hardware allows. Here, there are five main rounds instead of three. Each of the five main rounds are operating on different inputs. When the former input finishes its two turn around, it goes to the latter main round

and its former place is filled with the new input. The last main round block works again once for every different input. In this way, it is possible to have an encrypted text output every two cycles. Scount module of the 64-bit pipelined version and overall block diagram can be seen below in Figure 3.5 and Figure 3.6, respectively. The main round is the same as before.

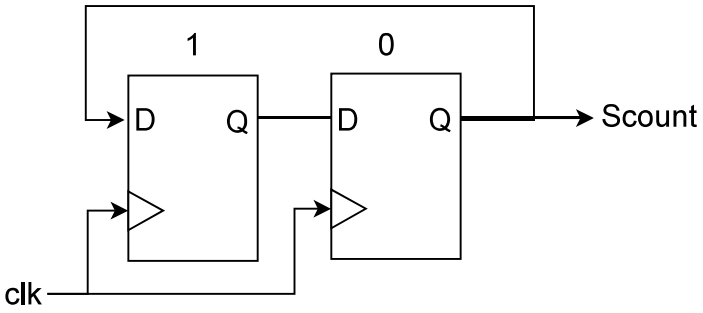
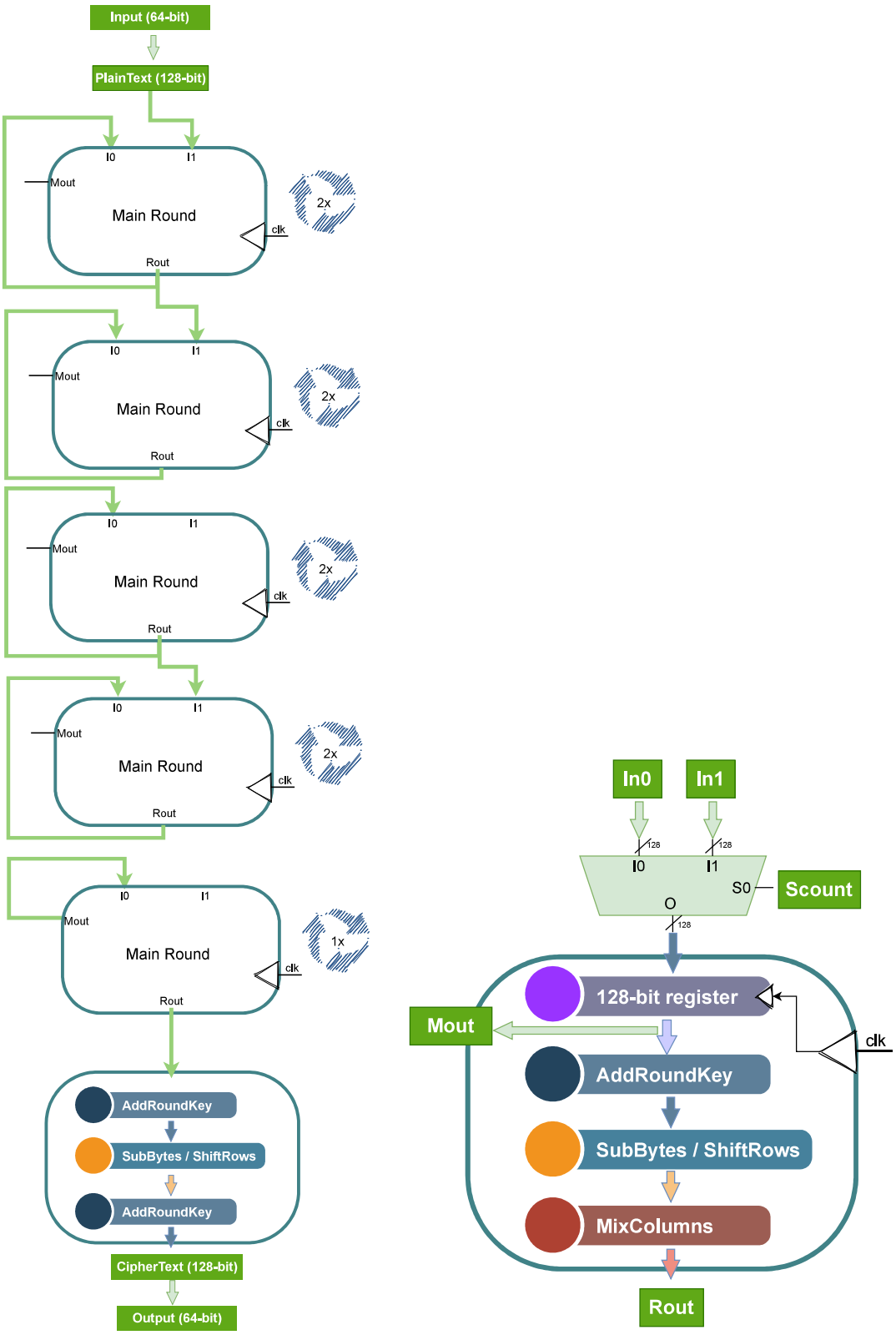


Figure 3.5. Pipeline (64-bit) Scount Module.



(a) (b)

Figure 3.6. (a) Pipeline (64-bit). (b) Pipeline Main Round.

3.2. Low Power Techniques

Several RTL techniques were applied to reduce power consumption. One of them is stopping the data switching when unnecessary. The AES Intellectual Property (IP) encrypts the plain text only when it takes the start flag and the key is ready. Therefore, it does not encrypt the same plain text again and again. With the help of relevant if-else statements, the registers maintain the combinational logic inputs constant resulting in no activity in the combinational circuit. An example from the encryption module is as follows: the state machine works according to the value of the counter. After the encryption operation is completed, the counter value becomes fixed, and the same else-if branch is executed repeatedly. Therefore, the other register values stay the same.

There are also techniques related to the ASIC flow that are explained in detail in Chapter 5 but summarized here to give low-power techniques together with the techniques in the RTL at a glance. The first of them is the selection of the Low-Power (LP) process for the TSMC 65 nm purpose. There is also General Purpose (GP) process available with faster standard cells that consume more power. So, the possible increase in operating frequency and the throughput are sacrificed in favor of lower power consumption. The second technique is the usage of standard cells whose transistors have high threshold voltage. Higher threshold voltage cells need higher voltage to turn on and this results in lower operating current as can be seen from the Metal Oxide Semiconductor Field Effect Transistor (MOSFET) current formula,

$$i_D = \frac{1}{2}k_n^{'}(\frac{W}{L})(V_{GS} - V_t)^2 \tag{3.1}$$

where i_D is the MOSFET current in saturation, $k_n^{'}$ is the process transconductance parameter, $\frac{W}{L}$ is the transistor aspect ratio, V_{GS} is the gate-to-source voltage, and V_t is the threshold voltage [21]. Therefore, they consume less power but they are slower. On the other hand, low threshold voltage cells are faster but consume more power. The ASIC tools are set to use high threshold voltage cells as much as possible. The third technique is using clock gating in ASIC flow. The clock is suppressed when unnecessary.

3.3. Side Channel Attack Countermeasures

In order to prevent or complicate side-channel attacks, obfuscating operation, which utilizes the Boolean Masking depicted in Figure 3.7, was employed. As not whole round is masked including linear and nonlinear operations, the method was called as obfuscation instead of masking. In the RTL the obfuscation was obtained by masking only ShiftRows and MixColumns operations with a mask obtained from LFSR whose 128-bit seed is the cipherkey. Assuming linear function f converts the input to output, XOR'ing two outputs where one comes from f(mask) and the other one comes from f(input $\oplus$ mask) yields the same output as before. The mentioned operation can be explained mathematically as

$$f(input) = f(input \oplus mask \oplus mask) = f(input \oplus mask) \oplus f(mask) = output \quad (3.2)$$

where the associative property of the XOR is used.

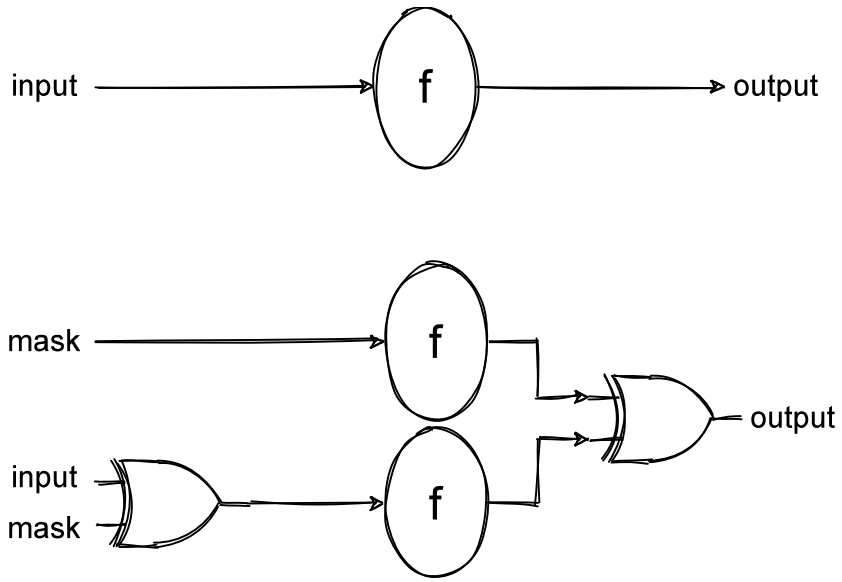


Figure 3.7. Boolean Masking.

The advantage of obfuscating is that it complicates the correlation between input and output. Therefore, the attacker needs more traces to discover the key. It is important to note that the mask value is unimportant and it does not need to be stored. Hence, the mask value can be changed at every cycle which is the case in this thesis. So, every 128-bit input can be processed at every round with a different 128-bit mask. The modules were parametrized so that changing the parameter from the top enables and disables obfuscating.

In addition to the obfuscating, pipelining is used. As the operating principle is explained in Section 3.1, the details are not repeated here. Pipelining allows different inputs to be processed at the same time instead of being processed one-by-one. Number of processed blocks inside rounds are the same as the number of main round blocks. Therefore, three different inputs for 32-bit version and five different inputs for 64-bit version are processed simultaneously preventing the hardware from staying idle. The addition of power consumption values due to different inputs give a total consumption value which is dependant on all inputs. From the perspective of the attacker, it becomes harder to analyze inputs individually and more traces become necessary to disclose the cipherkey. Hence, the side-channel attack resistance is improved.

4. FPGA DESIGN FLOW

The RTL code was added to the FPGA flow using Vivado Design Suite (version 2020.2) tool developed by Xilinx. There is RTL at the beginning of the FPGA flow. No IP was needed for this design, RTL was custom designed. Then, the implementation stage starts, and the RTL is converted to the logical gates implemented on the FPGA by resources such as Look-up Table (LUT), Block RAM (BRAM), Digital Signal Processing (DSP) slice, etc. In the end, the bit file becomes ready to be put into FPGA. Various stages of the flow are given below.

4.1. Behavioral Simulation

The RTL code needs to be verified in order to function properly. Behavioral simulation is the first step as it does not consider real delays between the circuit gates. The simulation is used to verify the functionality of the design, but it is less accurate compared to the simulations at the later stages. In this thesis, a test bench that receives 128-bit random inputs from a text file is used in the simulation. The random inputs were generated on MATLAB.

In addition, KAT vectors and Monte Carlo Test of the AESAVS, as mentioned in Section 2.4, are used to validate the results. The ciphertext results are written into the file and compared with the reference values that are generated using American National Standards Institute (ANSI) C reference code in [22].

4.2. Synthesis

In the synthesis, RTL is converted to the standard subcells used by the FPGA such as LUTs, flip-flops, BRAMs, etc. FPGA uses LUTs to implement logical functions. For example, for a three-input logical function, there are eight possible cases. Regardless of the function's complexity, the output of this function can be implemented

with 3-input LUTs. There are preplaced LUTS and the tool maps the functions to these LUTs. Flip-flops can be used for storage units, i.e registers. BRAM's are convenient for replacing large register arrays. The RTL is converted to FPGA primitives explained above, but their places on the FPGA fabric is not known yet.

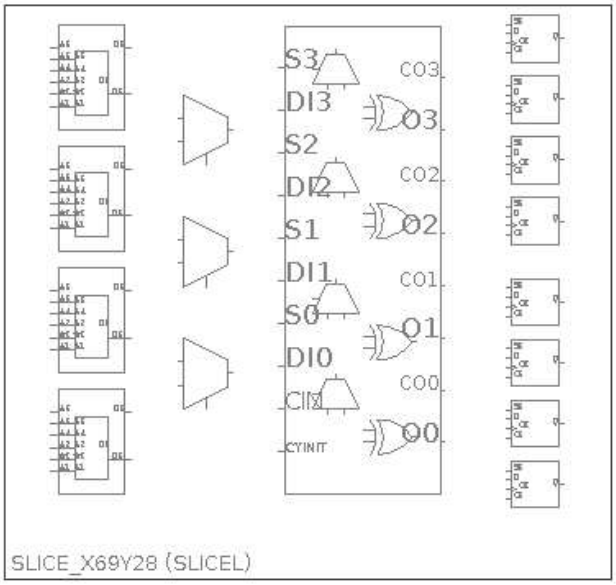


Figure 4.1. FPGA Resources.

In Figure 4.1, inside of one slice can be seen. There are four LUT's, two F7 MUXes, one F8 MUX, eight flip-flops, and one carry chain for the Xilinx 7-series architecture. There are two different types of slices: SLICEL and SLICEM. The difference is that LUT's of SLICEM can be used as distributed RAM or shift registers. The synthesis tool optimizes the design to use fewer resources, consume less power, and work as intended. For these objectives, the tool takes the constraints, which are given by the designer, into account. The clock period was chosen as 50 ns for unrolled version, 12 ns for 64-bit pipelined version, and 11 ns for other versions, which are reasonable values after trial and error. The interface of the input ports (rst_ni, plain_text) was modeled with the minimum 0 ns and maximum 1 ns input and output delays using set_input_delay and set_output_delay. That means the inputs can arrive up to 1 ns

after the clock, hence there is less space in the clock period for Input-to-Register (I2R) paths. Output ports were loaded with 1.4 pF with the set_load command.

4.3. Implementation

The target board for the implementation was chosen as ZedBoard, which contains 7-series programmable logic. The device appearance of the design, unrolled version, is as in Figure 4.2. The aqua cells indicate the elements used by the design.

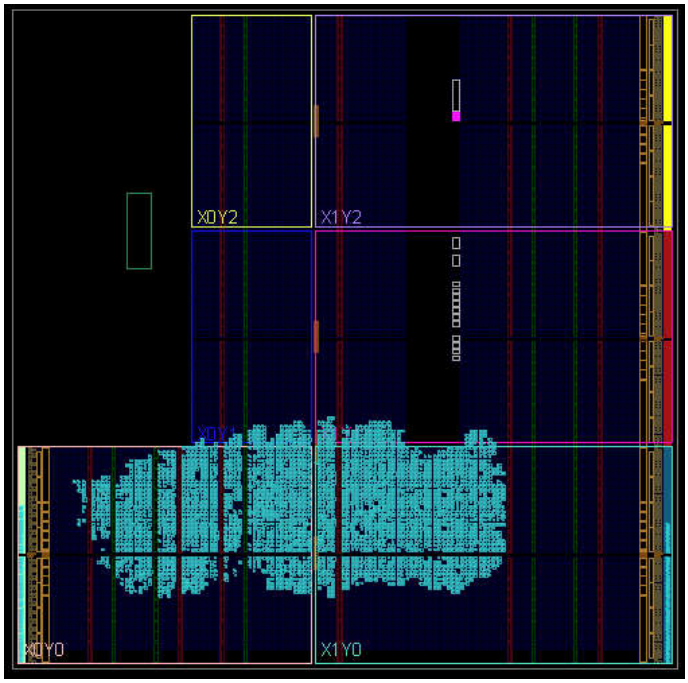


Figure 4.2. FPGA Implementation.

Main utilization parameters can be seen in Table 4.1. LUT's and registers are frequently reported in the literature for the area comparison between different designs. The obfuscated design has 12.5% higher LUT's and 18.13% higher registers than the rolled design. The important part of the additional logic comes from the LFSR. Pipelined versions have higher resource utilization since additional hardware is allocated to allow the processing of different data. The unrolled version has the highest

LUT usage since its rounds are all different hardware blocks. However, it uses fewer registers as the rounds are combinationally connected. For LUTs and registers, the values are increased as we go from the first version to the last version.

Table 4.1. FPGA Utilization (Implementation).

FPGA Resources	**Rolled**	**Obfus.**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
Slice LUTs	4504	5067	5796	7350	9653
Slice Registers	2162	2554	2646	3148	1805
F7 Muxes	608	472	1120	1632	2656
F8 Muxes	288	152	544	800	1312

Detailed utilization results involving primitives can be seen in Table 4.2 . FDRE is a D Flip-Flop with clock enable and synchronous reset. BUFG is a "General Clock Buffer". IBUF is an "Input Buffer", and OBUF is an "Output Buffer". IBUF and OBUF numbers come from the number of input & output ports, respectively [23]. Every external Input / Output (I/O) is connected to these buffers to help them deal with the large off-chip capacitances. IBUF includes clk_i, rst_ni, start_plain_text, and start_key signals in addition to the 32-bit/64-bit input data. On the other hand, OBUF includes only 32-bit/64-bit output data.

Timing and power results at synthesis stage can be seen in Table 4.3 and Table 4.4, respectively. Zeroing Worst Negative Slack (WNS) and Worst Hold Slack (WHS) ensures that every signal flows from its source to its target in the allowed time. If the signal goes too early or too late, the flip-flops would pick up false values and the data would be corrupted. Worst Pulse Width Slack (WPWS) is the worst slack based on collection of checks, namely min low pulse width, min high pulse width, min period, max period, and max skew [24]. That means the clocks are not allowed to go beyond a particular frequency & duty cycle. AC and DC characteristics of the FPGA fabric affect these pulse width checks. As given in Table 4.3, the design have no negative slack values at the implementation stage. Therefore, they are timing clean.

Table 4.2. FPGA Utilization Primitives (Implementation).

FPGA Primitives	Rolled	Obfus.	Pipe. 32	Pipe. 64	Unrolled
FDRE	2162	2554	2646	3148	1805
LUT6	2685	2992	3781	4648	6829
LUT5	1065	1066	892	1043	1659
LUT4	471	476	456	462	308
LUT3	343	519	599	1362	1435
LUT2	322	490	684	904	850
LUT1	10	10	14	10	17
MUXF7	608	472	1120	1632	2656
MUXF8	288	152	544	800	1312
IBUF	36	36	36	68	36
OBUF	32	32	32	64	32
CARRY4	0	11	0	0	0
BUFG	1	1	1	1	1

Table 4.3. FPGA Timing (Implementation).

	Rolled	Obfus.	Pipe. 32	Pipe. 64	Unrolled
Frequency (MHz)	90.909	90.909	90.909	83.333	20
Period (ns)	11	11	11	12	50
WNS (ns)	0.125	0.028	0.028	0.178	1.442
WHS (ns)	0.062	0.074	0.050	0.057	0.060
WPWS (ns)	5.000	5.000	5.000	5.500	24.500

The power consumption values are calculated at commercial temperature grade, typical process, and 25 °C board temperature. Most of the total on-chip power is attributed to the device static power. Device static power is consumed even if there is no consumption in our design as the FPGA uses many components to work properly. As an example, for the rolled version, 70.43% of the total power comes from static power. 31% of the dynamic power is due to the clocks. The IO's cause the highest percentage, 31.5%. The logic results in 14.2% and finally signals use 23.3% of the total dynamic power.

Table 4.4. FPGA Power Consumption (Implementation).

	Device Static Power (mW)	**Dynamic Power (mW)**	**Total On-Chip Power (mW)**	**Junction Temperature**
Rolled	105.005	44.086	149.091	26,7 °C
Obfuscat.	105.095	50.196	155.290	26,8 °C
Pipe. 32	105.753	94.573	200.326	27,3 °C
Pipe. 64	106.122	119.112	225.234	27,6 °C
Unrolled	108.343	262.304	370.647	29,3 °C

5. ASIC DESIGN FLOW

In ASIC flow, RTL is converted to standard cells that come with the design kits of process technologies. Intellectual Properties (IPs) created by foundries or third-party companies can also be added to improve functionality. After Place and Route (P & R) of these submodules, signoff checks are carried out to ensure that the design complies with the rules imposed by foundries. Designs that do not comply with these rules may not function as intended or may be rejected by the foundry. At the end of the ASIC flow, design is converted to Graphic Design System (GDSII) format and sent to foundries for fabrication. In this work, no third-party IP was used. The layouts were generated for TSMC 65 nm process, but they were not fabricated. TSMC 65 nm gives two essential choices, Low Power (LP) and General Purpose (GP). LP, which is slower than GP, was chosen to lower power consumption. For ASIC flow, proprietary tools from Cadence, namely Genus Synthesis Solution (v19.11), Innovus Implementation System (v19.11), Quantus Extraction Solution (v19.1.3), Tempus Timing Signoff Solution (v19.11), Voltus IC Power Integrity Solution (v19.11), and Xcelium Logic Simulation (v19.03) were used.

5.1. Synthesis

Synthesis is the first stage after RTL code is written and simulated behaviorally. Cadence Genus was used for this step of flow. The RTL is converted to standard cells provided by the foundry. Addition of constraints determines the boundaries for the design. The tool optimizes the design and tries to comply with the constraints given by the designer. The tool gives a synthesized netlist to be passed to the next stage of the flow, which is Place and Route (P & R). Inputs to the synthesis tool are: RTL files (.vhd, .v, .sv), constraint files (.sdc), standard cell timing libraries (.lib), standard cell physical libraries (.lef), and RC extraction libraries (QRC tech file). The results obtained at the synthesis stage can be found in Table 5.1.

Table 5.1. Synthesis Results.

	Rolled	**Obfus.**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
Period (ns)	8	8	8	8	40
Frequency (MHz)	125	125	125	125	25
WNS (ns)	0	0	0	0	0
Sequential Instances	2,147	2,519	2,633	3,155	1,757
Total Instances	23,791	25,172	37,931	51,830	80,450
Total Area (mm^2)	0.150	0.162	0.232	0.312	0.468
Leakage Power (mW)	0.005	0.005	0.020	0.023	0.089
Total Power (mW)	8.473	9.793	17.517	23.097	25.156
SVT (%)	1.0	0.6	12.4	9.2	32.1
HVT (%)	99.0	99.4	87.6	90.8	67.9

WNS is similar to its FPGA counterpart described in Section 4.2. Hold analysis results were not included here as the hold optimizations were not done at this stage. Instances in the table correspond to the number of standard cells. Total instances include both combinational and sequential instances. The last line in the table gives the percentage of standard threshold and high threshold standard cells. The process contains three types of standard cells: low V_T (LVT), standard V_T (SVT), and high V_T (HVT). Low threshold cells have low V_T transistors that can be turned on and off faster, but leakage currents are higher since there is an exponential relationship between the subthreshold current and the threshold voltage, leakage power increases considerably as the threshold voltage is decreased [25]. So, these cells are faster but consume more power. On the other hand, HVT cells consume less power but they are slower. The tool tries to trade-off between timing and power consumption. It puts HVT cells where timing is not critical and avoids them for timing critical paths. SVT is between LVT and HVT. During the synthesis stage, LVT cells were not allowed, since LVT cells need to be used during the P & R stage where issues due to the physical layout become more visible. Therefore, the synthesis tool used SVT cells for timing critical paths.

As can be seen in Table 5.1, the largest design is the unrolled version. Since its rounds are expanded, it is built from many standard cells, which results in high leakage & total power. The sequential instances of the unrolled version are fewer compared to the other versions since registers between the rounds are eliminated by combinationally connecting the rounds. Pipelined versions are intermediate designs between rolled and unrolled versions in terms of number of instances, area, and power consumption. They have a higher number of logical cells compared to the rolled version but a lower number compared to the unrolled version. Generally, HVT cells have an overwhelming majority which means most of the nets in the designs have no problem for satisfying the timing constraints. The unrolled version has the lowest frequency since its combinationally connected rounds need a large time period to fit in.

5.2. Place & Route

Place & Route stage is where the final locations of the standard cells are determined and the actual wires between them are laid out. The stage receives synthesized netlist and constraints from Genus as its inputs. For the design corners, Multi-Mode Multi-Corner (MMMC) analysis was carried out in Innovus tool. The designs were implemented so that they work at different corners. The standard cell libraries in the TSMC 65 nm LP process are characterized according to the corners in Table 5.2.

The process column in Table 5.2 stems from the intra-wafer variations during the fabrication step. Because of the slight fabrication parameter differences between the location of the different dies within the wafer, some chips work faster and some chips work slower than nominal. These different chips are modeled as FF (Fast-Fast), TT (Typical-Typical), SS (Slow-Slow). The letters correspond to the NMOS and PMOS corners, respectively [26]. FF means both NMOS & PMOS are faster while FS would mean fast NMOS and slow PMOS. Values in the voltage column are set up according to the typical core voltage, which is 1.2V and this is the voltage used in our design. Higher voltage means faster circuits but higher power consumption.

Table 5.2. Process, Voltage, Temperature (PVT) Corners.

Corners	Process	Voltage (Volts)	Temp. (°C)
Best Case (BC)	FF	1.32	0
Low Temperature (LT)	FF	1.32	-40
Maximum Leakage (ML)	FF	1.32	125
Typical Case (TC)	TT	1.2	25
Worst Case (WC)	SS	1.08	125
Worst Case Low Temp. (WCL)	SS	1.08	-40
Worst Case Zero Temp. (WCZ)	SS	1.08	0

Apart from PVT corners, there are also RC extraction corners. For older technologies, cell delay dominates the total delay, and the effect of the interconnects remains small. However, with the advanced technology nodes, the interconnects become more and more important and their contribution to the total delay increases [25]. Therefore, more detailed modeling of the interconnect delay, roughly R*C, becomes necessary. For the 65 nm LP node, there are five different RC extraction corners: Cmin, Cmax, RCmin, RCmax, and typical. The combination of the PVT corners and RC extraction corners is used during analysis. In the flow, the tool tries to maintain timing in different cases. The remaining flow can be summarized as in Figure 5.1.

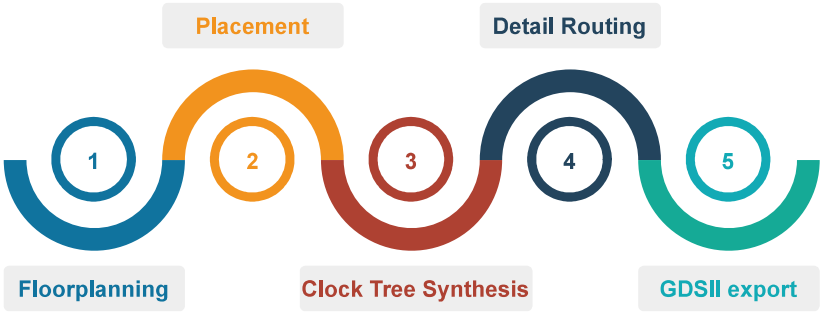


Figure 5.1. P & R Flow.

When the libraries, synthesized netlist, and constraints are given, the floorplanning stage starts. The area of the layout was determined according to 60% standard cell density so that the tool can route easily, but the area is not unacceptably high. So, a minimum area layout with a clean Design Rule Check (DRC) and clean timing was targeted. 60% standard cell density means the total area of the individual standard cells are multiplied by $\frac{10}{6}$ to give the total floorplan area. By keeping the floorplan size flexible value enables observing area differences between different versions.

The standard cells in the netlist are placed into the layout at the placement stage. The appearance of the rolled obfuscated version at this stage without nets is in Figure 5.2. The gray cloud contains many gray boxes representing the standard cells.

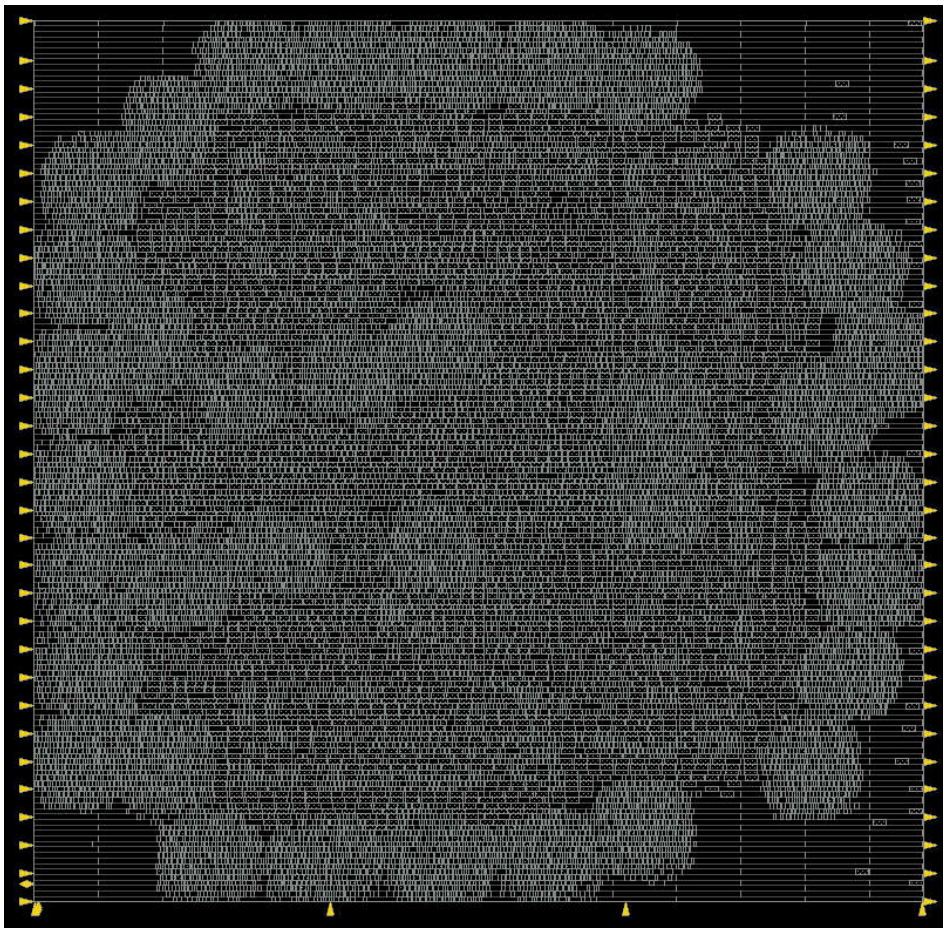


Figure 5.2. Placed Design.

Clock Tree Synthesis (CTS) is another major step in the backend flow. Clock, which is a high fanout net, needs special attention. Special clock cells are used in the clock network in order to distribute the clock efficiently. The clock nets are chosen to be wider than the other nets in the same metal layer in order to lower the resistance of the nets. The clock nets are separated farther away from other nets compared to the separation between regular nets in the same metal layer. This measure protects clock signals from other switching nets, i.e it maintains signal integrity. Both widening the clock nets and increasing the separation were used in this thesis. This way, the clock signals are preserved at the expense of additional routing resources.

At the detail routing stage, the real nets are laid out. The most comprehensive violation fixing happens there. Timing violations (setup, hold, etc.), DRC violations, Design Rule Violations (maximum fanout, maximum transition, etc.) are addressed. This stage is one of most time consuming steps in the IC flow. The design statistics are provided in Table 5.3. All designs are timing clean as can been with positive WNS & WHS. The layout of the rolled version is depicted in Figure 5.3 after the nets are laid out.

Table 5.3. Design Statistics (Innovus Final).

	Rolled	**Obfus.**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
Logical Instances	20,808	22,498	35,986	50,809	80,062
Area (mm^2)	0.134	0.149	0.208	0.278	0.415
Period (ns)	8	8	8	8	40
Frequency (MHz)	125	125	125	125	25
WNS (ns)	0.297	0.255	0.086	0.264	0.124
WHS (ns)	0.094	0.076	0.090	0.080	0.056
LVT (%)	4.3	6.6	11.5	10.0	20.1
SVT (%)	2.2	2.0	10.2	7.9	21.4
HVT (%)	93.4	91.3	78.3	82.1	58.4

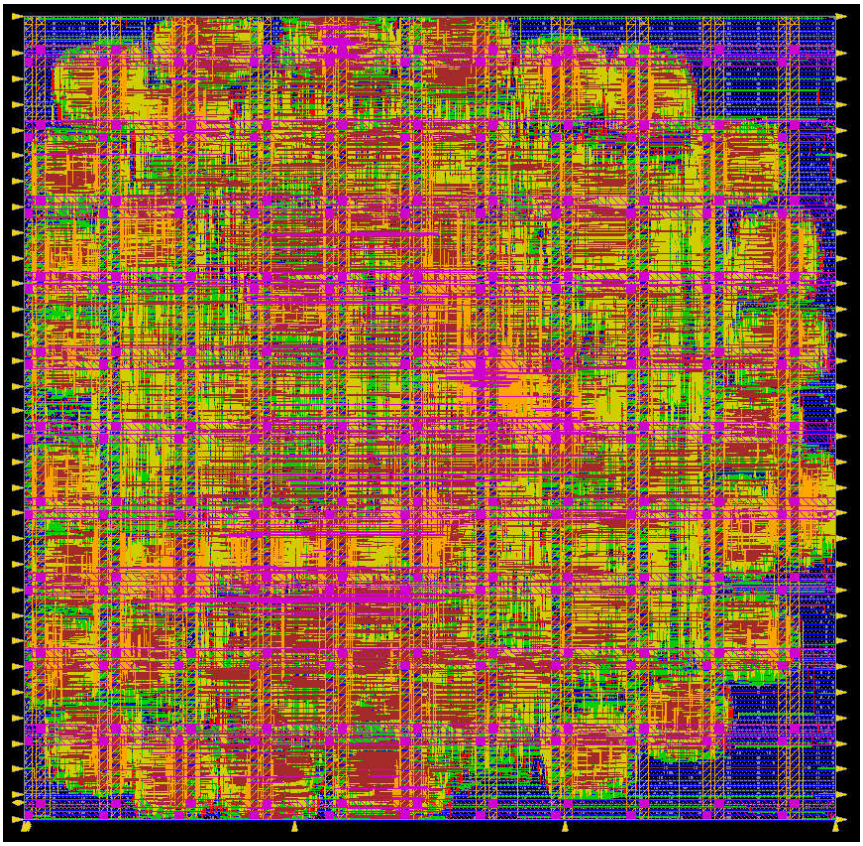


Figure 5.3. Layout (Final).

After the design is completed the layout was exported as Graphic Design System (GDSII) file. It is a binary file that describes the geometric shapes along with other information about the layout [27]. The GDSII file is then sent to the foundry for fabrication. In our case, the GDSII file was generated but the design was not fabricated due to high fabrication costs.

5.3. Signoff RC Extraction

Signoff parasitics were extracted at this stage using Cadence Quantus tool. Standard Parasitic Exchange Format (SPEF) files, which shows parasitic RC values, are generated and parasitic values are used in other signoff tools to accurately model the interconnect delays. Signal integrity is also closely associated with parasitics.

5.4. Signoff Timing Analysis

Static Timing Analysis (STA) was executed with Cadence Tempus for the signoff timing analysis. The tool takes the design netlist and placement information from Innovus, and RC extraction information (.spef files) from Quantus as inputs. It analyzes all paths and checks for possible timing violations. Innovus also makes timing analyzes but Tempus performs the analysis at the signoff quality by using signoff quality RC extraction results. STA results of the design can be examined below in Table 5.4 and in Table 5.5. As can be seen, all slack values except wcl_cworst - unrolled, which fails slightly, are greater than zero, which means that the designs meet the setup & hold timing at every PVT corner mentioned in Table 5.2. Finally, .sdf file was generated to be used at the gate-level simulation. Standard Delay Format (SDF) is a format that contains timing information such as interconnect delays, constraints, and cell path delays [28].

Table 5.4. Timing Violations (Setup).

	Rolled	**Obfus.**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
Period (ns)	8	8	8	8	40
Frequency (MHz)	125	125	125	125	25
Views	**Slack Values (ns)**				
tc_typical	3.664	3.739	3.359	3.374	16.346
wc_cworst	0.352	0.474	0.054	0.164	0.177
wcl_cworst	0.083	0.092	0.031	0.060	-0.165

Table 5.5. Timing Violations (Hold).

	Slack Values (ns)				
	Rolled	**Obfuscated**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
bc_cbest	0.097	0.082	0.098	0.078	0.058
bc_cworst	0.097	0.076	0.100	0.088	0.062
lt_cbest	0.092	0.074	0.091	0.081	0.055
lt_cworst	0.064	0.079	0.096	0.089	0.058
tc_typical	0.215	0.194	0.199	0.204	0.174
wc_cworst	0.355	0.367	0.372	0.357	0.361
wcl_cworst	0.328	0.385	0.334	0.401	0.385

5.5. Gate-Level Simulation

The gate-level netlist, obtained from Innovus, was functionally simulated at this stage using the Xcelium tool from Cadence. The .sdf file generated from Tempus was used to annotate the delays in the design for a more accurate model, compared to the behavioral simulation model. As a test bench at the signoff, the files in Appendix C and D were used. The results were checked so that the design still works correctly after the ASIC flow.

For each of the different inputs that were simulated, switching information was dumped in Value Change Dump (VCD) files, which report the nets that switch and their switching times. These files are used in the signoff power analysis tool to accurately predict power consumption according to the particular scenario. Another file called Toggle Count Format (.tcf) file reports the number of times a particular net switches without specifying when they switch. Both files were used in this work.

5.6. Signoff Power Analysis

Signoff power analysis was completed using Voltus tool from Cadence. Similar to the Tempus tool, netlist and placement information from Innovus, RC extraction information (.spef files) from Quantus are taken as inputs. Several different power analyses were completed in this step. Static power analysis assumes a default switching activity for nets, i.e. every net switches once every five clock cycles. It gives a rough estimation of the design. In the dynamic power analysis with vectors, the activity information received from .tcf and .vcd files were used. Results from the mentioned power analyzes at typical case are summarized in Table 5.6.

Table 5.6. Power Consumption (Total).

Power (mW)	**Rolled**	**Obfuscated**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
Static	8.313	9.503	12.9	17.92	4.26
Dyn. (.tcf)	5.049	7.091	16.01	32.15	21.49
Dyn. (.vcd)	5.051	7.097	16.85	32.16	21.47

Since the test bench contains many different inputs, power analysis using .vcd files was executed many times. By doing this, current traces for different inputs were obtained. The format of the trace file can be seen in Figure 5.4. It contains three columns: point number, time of the point (second), and the current value (ampere). Traces were obtained in 50 ps time steps. These traces were transferred to the ChipWhisperer tool for side-channel analysis. It should be noted that the mentioned current traces will be called power traces in the rest of the thesis. Since the voltage can be assumed as nearly constant, a current trace can be interpreted as a power trace.

```
 0    0.0000e+00   0.00114845
 1    5.0000e-11   0.00392821
 2    1.0000e-10   0.00516825
 3    1.5000e-10   0.0125037
 4    2.0000e-10   0.0141497
 5    2.5000e-10   0.0054537
 6    3.0000e-10   0.00347611
 7    3.5000e-10   0.0109725
 8    4.0000e-10   0.0428358
 9    4.5000e-10   0.0607904
10    5.0000e-10   0.058673
```

Figure 5.4. Power Trace File.

Figure 5.5 presents the current trace graph, for the rolled version. The peaks correspond to the clock edges. The circuit exhibits significantly lower power consumption between the clock edges, which is expected in digital Complementary Metal Oxide Semiconductor (CMOS) circuits. Initial and final points in the graph stem from receiving the plaintext and giving the ciphertext. They result in higher peaks as the interface is connected to the high load capacitances. The middle points correspond to the power consumption during the actual encryption operation.

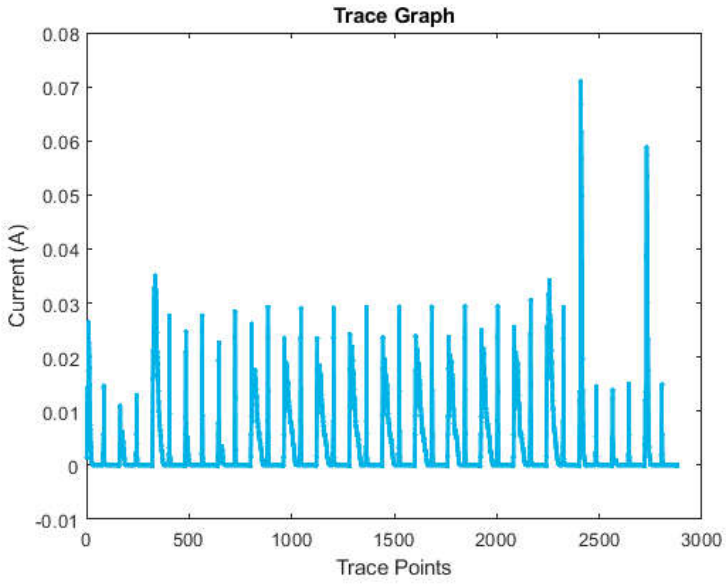


Figure 5.5. Current Trace Graph.

6. EXPERIMENTS AND RESULTS

After the protected and unprotected implementations are completed, the designs were subjected to the side-channel attack analysis. The effectiveness of the protection schema was compared to the literature using key metrics.

6.1. Side-Channel Attack Resistance Evaluation

Side-channel attack resistance was evaluated using ChipWhisperer, Windows environment version 5.5. ChipWhisperer is an open-source toolchain that has a complete setup, namely target hardware, capture hardware, firmware, software. For example, ChipWhisperer-Lite board has Atmel XMEGA as a target. The algorithms are run on the target and the capture part is used with analog circuitry to capture the power consumption data. The board is designed so that special circuitry provides clear power traces. In this way, power and glitch side-channel attacks can be run easily with its specialized setup [29]. Though analyzed data come from the ChipWhisperer developed boards, it is also possible to analyze traces obtained from other places. The second approach was used during the thesis. Since it is very expensive to tape out our custom AES implementation and it is harder to extract relevant power traces when implemented in FPGA, the analyzer module of the ChipWhisperer (cwa) was mainly used.

There are different leakage models available to be used during the attack. They focus on different stages of the AES to extract the secret key. They come to different conclusions with different correlation values. The results were obtained after 100,000 random inputs, which were generated on MATLAB, were encrypted using the same key. Traces have data points separated by 50 ps to give the total period per trace. Total period changes according to the version analyzed. Target cipherkey is in Table 6.1. The most significant bytes in the tables are the bytes with the number 15.

Table 6.1. Target Cipherkey.

Bytes of the 128-bit Target Cipherkey															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
2b	7e	15	16	28	ae	d2	a6	ab	f7	15	88	09	cf	4f	3c

Subkey predictions depending on the leakage model that is more successful in revealing the key for particular version, are given in Tables 6.2 to 6.16. Different models were tried with 10,000 traces and selected ones were used for 50,000 and 100,000 traces. Chosen leakage model for rolled version is after_key_mix, which uses the Hamming Weight of the output of the first AddRoundKey operation as an attack point. mix_columns_output, which attacks the first round MixColumns using Hamming Weight model, was chosen for rolled obfuscated and pipelined versions. sbox_output model was chosen for unrolled version. It attacks the first round SubBytes operation using Hamming Weight model. The software compares the actual traces with the predictions obtained using all possible key combinations and the chosen model. The key prediction that is more compatible with the actual traces has a higher correlation value. The order of the results follows the version order in Section 3.1. The 100,000-input attack took approximately one hour and 29 minutes for the rolled version, one hour and 34 minutes for the obfuscated version, one hour and 32 minutes for 32-bit pipelined version, one hour and 20 minutes for 64-bit pipelined version in a computer that has AMD Ryzen 5 3600 6-Core Processor and 16.0 GB RAM. They are approximately around one hour and thirty minutes as background applications also affect the result. In the tables below, there are five subkey predictions for every byte position. The uppermost ones are the ones with the highest correlation with the traces. The bytes that matched the target have their cells highlighted. Three different trace numbers are selected: 10,000, 50,000, and 100,000. Analyzing different trace numbers gives insight into how the results change as the attack continues. In addition, the reliability of the comparison is increased as the comparison is repeated with different trace numbers.

Table 6.2. Byte Prediction Results (Rolled, 10,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	00	81	ff	00	ff	ff	ff	ff	c4	00	ff	00	ff	20	00	ff
2	ff	7e	00	ff	00	00	00	00	3b	ff	00	ff	00	df	ff	00
3	08	00	ea	e9	d7	ae	bf	59	00	10	f7	88	e6	30	4f	e9
4	f7	ff	15	16	28	51	40	a6	ff	ef	08	77	19	cf	b0	16
5	15	c1	fe	a9	f7	fe	01	40	d4	80	fb	04	f2	00	04	c3

Table 6.3. Byte Prediction Results (Rolled, 50,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	00	81	ff	00	ff	ff	ff	ff	00	00	ff	00	ff	20	ff	ff
2	ff	7e	00	ff	00	00	00	00	ff	ff	00	ff	00	df	00	00
3	15	00	ea	e9	d7	ae	fd	a6	c4	01	fb	88	e6	30	4f	16
4	ea	ff	15	16	28	51	02	59	3b	fe	04	77	19	cf	b0	e9
5	35	c1	fe	a9	57	59	fb	fb	d4	02	02	80	0d	00	fd	3c

Table 6.4. Byte Prediction Results (Rolled, 100,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	ff	7e	ff	ff	ff	ff	ff	ff	ff	ff	00	ff	ff	20	ff	ff
2	00	81	00	00	00	00	00	00	00	00	ff	00	00	df	00	00
3	15	00	ea	e9	d7	ae	fd	59	54	08	80	88	19	cf	4f	3c
4	ea	ff	15	16	28	51	02	a6	ab	f7	7f	77	e6	30	b0	c3
5	35	76	ef	a9	57	ef	bf	bf	d4	01	08	02	0d	ff	02	16

Table 6.5. Byte Prediction Results (Rolled Obfuscated, 10,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	40	ff	ff	ff	09	ff	ff	ff
2	4a	fe	fe	fe	1e	fe	fe	fe	a7	fe	fe	fe	3b	fe	fe	fe
3	e7	fd	fd	fd	08	fd	fd	fd	ab	fd	fd	fd	ad	fd	fd	fd
4	8f	fc	fc	fc	f3	fc	fc	fc	a6	fc	fc	fc	f9	fc	fc	fc
5	f6	fb	fb	fb	4f	fb	fb	fb	5d	fb	fb	fb	bb	fb	fb	fb

Table 6.6. Byte Prediction Results (Rolled Obfuscated, 50,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	e7	fe	fe	fe	56	fe	fe	fe	ef	fe	fe	fe	0d	fe	fe	fe
3	44	fd	fd	fd	99	fd	fd	fd	a6	fd	fd	fd	23	fd	fd	fd
4	97	fc	fc	fc	40	fc	fc	fc	d5	fc	fc	fc	3f	fc	fc	fc
5	25	fb	fb	fb	13	fb	fb	fb	ad	fb	fb	fb	45	fb	fb	fb

Table 6.7. Byte Prediction Results (Rolled Obfuscated, 100,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	25	fe	fe	fe	56	fe	fe	fe	ef	fe	fe	fe	0d	fe	fe	fe
3	eb	fd	fd	fd	40	fd	fd	fd	a6	fd	fd	fd	45	fd	fd	fd
4	37	fc	fc	fc	2e	fc	fc	fc	a8	fc	fc	fc	08	fc	fc	fc
5	67	fb	fb	fb	34	fb	fb	fb	ac	fb	fb	fb	9d	fb	fb	fb

Table 6.8. Byte Prediction Results (32-bit Pipeline, 10,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	a4	fe	fe	fe	e4	fe	fe	fe	d5	fe	fe	fe	b4	fe	fe	fe
3	07	fd	fd	fd	67	fd	fd	fd	a9	fd	fd	fd	d8	fd	fd	fd
4	3c	fc	fc	fc	29	fc	fc	fc	73	fc	fc	fc	00	fc	fc	fc
5	76	fb	fb	fb	c5	fb	fb	fb	03	fb	fb	fb	77	fb	fb	fb

Table 6.9. Byte Prediction Results (32-bit Pipeline, 50,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	de	fe	fe	fe	25	fe	fe	fe	a6	fe	fe	fe	85	fe	fe	fe
3	79	fd	fd	fd	5c	fd	fd	fd	1b	fd	fd	fd	65	fd	fd	fd
4	2f	fc	fc	fc	f0	fc	fc	fc	55	fc	fc	fc	2c	fc	fc	fc
5	50	fb	fb	fb	2c	fb	fb	fb	19	fb	fb	fb	96	fb	fb	fb

Table 6.10. Byte Prediction Results (32-bit Pipeline, 100,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	2f	fe	fe	fe	25	fe	fe	fe	3f	fe	fe	fe	04	fe	fe	fe
3	45	fd	fd	fd	2c	fd	fd	fd	8d	fd	fd	fd	20	fd	fd	fd
4	c7	fc	fc	fc	e8	fc	fc	fc	9b	fc	fc	fc	b4	fc	fc	fc
5	39	fb	fb	fb	8b	fb	fb	fb	bd	fb	fb	fb	28	fb	fb	fb

Table 6.11. Byte Prediction Results (64-bit Pipeline, 10,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	b3	fe	fe	fe	67	fe	fe	fe	72	fe	fe	fe	c1	fe	fe	fe
3	46	fd	fd	fd	e1	fd	fd	fd	3c	fd	fd	fd	e4	fd	fd	fd
4	f5	fc	fc	fc	9d	fc	fc	fc	be	fc	fc	fc	b3	fc	fc	fc
5	bf	fb	fb	fb	74	fb	fb	fb	e0	fb	fb	fb	31	fb	fb	fb

Table 6.12. Byte Prediction Results (64-bit Pipeline, 50,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	3a	fe	fe	fe	25	fe	fe	fe	13	fe	fe	fe	b5	fe	fe	fe
3	87	fd	fd	fd	2a	fd	fd	fd	ad	fd	fd	fd	ec	fd	fd	fd
4	40	fc	fc	fc	45	fc	fc	fc	89	fc	fc	fc	11	fc	fc	fc
5	2f	fb	fb	fb	db	fb	fb	fb	19	fb	fb	fb	a7	fb	fb	fb

Table 6.13. Byte Prediction Results (64-bit Pipeline, 100,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	2b	ff	ff	ff	28	ff	ff	ff	ab	ff	ff	ff	09	ff	ff	ff
2	2f	fe	fe	fe	86	fe	fe	fe	ca	fe	fe	fe	ee	fe	fe	fe
3	20	fd	fd	fd	50	fd	fd	fd	13	fd	fd	fd	0b	fd	fd	fd
4	da	fc	fc	fc	7f	fc	fc	fc	78	fc	fc	fc	10	fc	fc	fc
5	48	fb	fb	fb	22	fb	fb	fb	f7	fb	fb	fb	c9	fb	fb	fb

Table 6.14. Byte Prediction Results (Unrolled, 10,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	d7	4f	4f	f0	4f	09	f4	b0	b0	b4	b0	6f	09	0b	f4	4f
2	2a	b0	09	cf	2f	4f	b2	4f	0f	b0	f4	0b	f4	f4	b2	b2
3	5b	f4	f4	09	92	2d	2b	c1	0a	4f	4f	f0	2b	09	4f	d2
4	2c	b4	4b	b6	0b	f4	09	fe	f2	2d	0f	f4	0f	d2	09	b4
5	fb	29	2f	ab	f4	f0	2f	d0	4f	6f	92	f6	f0	2d	2f	f4

Table 6.15. Byte Prediction Results (Unrolled, 50,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	22	f4	4f	b0	f4	09	f4	09	f4	b0	b0	f4	f4	f4	f4	4f
2	db	0f	b0	f4	0b	4f	09	f4	ab	0b	4f	09	09	0b	09	f4
3	9f	4f	09	09	b0	f4	2f	4f	b0	09	f4	b0	b0	b0	4f	09
4	f3	b0	f4	0f	d6	b4	f2	0f	0f	f4	09	4f	4f	09	b0	b0
5	d7	0b	b4	f0	4f	a2	b0	d0	4b	4f	0b	0f	0f	4f	f0	0f

Table 6.16. Byte Prediction Results (Unrolled, 100,000 Traces).

	f	**e**	**d**	**c**	**b**	**a**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	97	f4	4f	f4	f4	4f	f4	09	f4	f4	b0	09	f4	f4	f4	f4
2	9f	0f	09	b0	b0	f4	09	f4	b0	b0	f4	f4	09	0b	09	4f
3	14	b0	b0	09	4f	09	4f	4f	ab	0b	4f	b0	4f	09	b0	09
4	22	2f	f4	0f	0b	b0	f0	b0	f0	09	0b	4f	b0	4f	4f	b0
5	bb	4f	f0	f0	09	a2	b0	b2	09	4f	09	f0	0f	b0	f0	0f

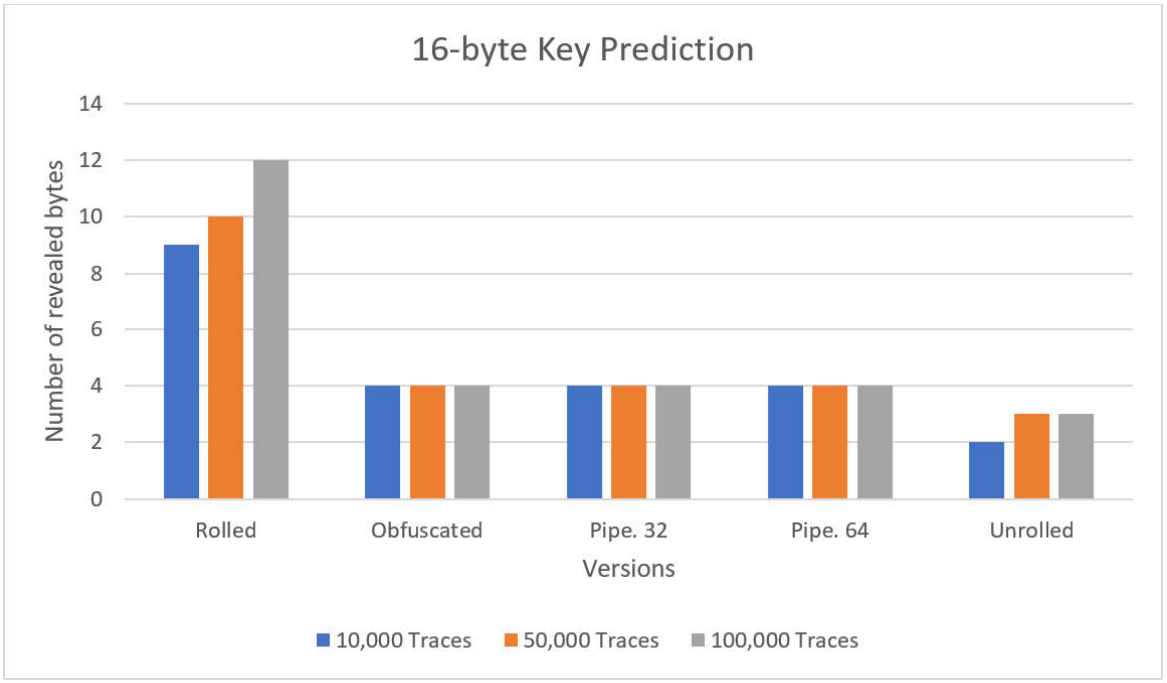


Figure 6.1. Side-Channel Attack Results.

The results from the tables can be seen at a glance in Figure 6.1 and the inferences can be summarized as follows:

- The most revealing one is the first version, i.e. rolled one.
- Effect of obfuscating is clearly visible when Table 6.3 and Table 6.6 are compared. The obfuscated version has a better revealing rate as 10 bytes are visible for the rolled version but four bytes are visible for the obfuscated version.
- Though both rolled and 32-bit pipelined versions take the input from the 32-bit interface, the pipelined version reveals much less information with 100,000 traces as can be seen in Table 6.10 and Table 6.4. This is consistent with the expectations as only one input is processed at a time in the rolled version, and hence there is no interference from other inputs.
- 64-bit pipelined version has similar revealing performance compared to the 32-bit pipelined version when Table 6.9 and Table 6.12 are compared. This is contrary to the expectations because more inputs are processed at the same time in the

64-bit pipelined version which is expected to increase side-channel resistance.

- Pipelined versions reveal four bytes with 10,000, 50,000, and 100,000 traces. As far as the attack point is concerned, which is the MixColumns step, the location of the revealed bytes implies that the attack is effective in the particular 32-bit block. This attack point reveals four bytes in all versions except the unrolled version, which implies that MixColumns is less protected compared to the other attack points. The reason can be the compute-intensive nature of MixColumns, which contains multiple vector multiplications in the Galois field that can give larger power fluctuations.
- Pipelined versions have better performance compared to the rolled version as visible in Table 6.4, Table 6.10, and Table 6.13.
- The unrolled version performs better compared to all other versions. This can be attributed to the combinational nature of the unrolled version since transitions between are not visible as clear as the spikes at the clock edges. Therefore, it is hard to distinguish power consumption differences between operations where different operations occur one after another with just a couple of gates delay.
- Collecting more traces reveals in general more subkeys as nine bytes are revealed with the analysis of 10,000 traces for the rolled version. 50,000 traces reveal 10 bytes as in Table 6.3. However, this assumption does not hold for 32-bit pipelined version as shown between Table 6.8 and Table 6.9 as both reveal four bytes. Some bytes are revealed with fewer traces but some bytes may be revealed after collecting much more traces. So, 10,000 and 50,000 may not be a good comparison window. The equal byte disclosure between pipelined and obfuscated versions can also be attributed to the mentioned reason.

Partial Guessing Entropy (PGE) gives the number of wrong byte predictions above the correct byte in the correlation list. For example, if the correct byte is the second most correlated prediction, the PGE becomes one. The results with 100,000 traces are tabulated in Table 6.17. Total PGE is the sum of the PGE values that belong to the 16 different bytes. Having a low PGE means more bytes are close to being revealed and hence the design has lower side-channel attack resistance.

Table 6.17. Partial Guessing Entropy (PGE) (100,000 Traces).

	PGE Values for the 128-bit Cipherkey Prediction				
Bytes	**Rolled**	**Obfus.**	**Pipe. 32**	**Pipe. 64**	**Unrolled**
f	31	0	0	0	178
e	0	129	129	129	5
d	3	234	234	234	56
c	3	233	233	233	40
b	3	0	0	0	29
a	2	81	81	81	82
9	9	45	45	45	15
8	3	89	89	89	15
7	3	0	0	0	2
6	3	8	8	8	23
5	16	234	234	234	29
4	2	119	119	119	47
3	6	0	0	0	1
2	2	48	48	48	17
1	2	176	176	176	3
0	2	195	195	195	61
Total	90	1591	1591	1591	603

6.2. Comparison

In Table 6.18, the proposed designs are compared against previous work. The design [9] was implemented in TSMC 22 nm CMOS process. It simulates the RC annotated post-layout netlist with CustomSim, a tool from Synopsys. Then, CPA is performed on the design. Masking on both linear and non-linear transformations is employed to prevent the side-channel attack. The design in [30] was fabricated on 65 nm process. DPA is performed on the power traces that are directly collected from the

hardware. A charge-recovery logic family, Bridge Boost Logic (BBL) is utilized to make energy dissipation independent of the switching. The design in [31] was fabricated on 130 nm process. Switched capacitor block is employed to equalize the current drawn by the sensitive blocks. The traces are again collected from the hardware. The target AES step is MixColumns in [30] and [31]. [32] was also implemented on 130 nm process. True Random Number Generator (TRNG) is used to provide random key values for the encryption and this TRNG calibrates itself according to the PVT variations. For side-channel attack resistance, Rotating S-box Masking (RSM) is utilized in which two barrel shifters are placed before and after the SubBytes stage and the masked data go through the S-box. Power traces are obtained using HSpice and the correlations are calculated with MATLAB OpenSCA. First byte of the SubBytes in the last round is targeted.

Table 6.18. Comparison to the Literature.

	Pipe. 32	**Pipe. 64**	**[9]**	**[30]**	**[31]**	**[32]**
Technology (nm)	65	65	22	65	130	130
Area (mm^2)	0.208	0.278	0.0169	0.291	1.37	N.A
Gates	36K	51K	16K	N.A	N.A	183.29K
Power (mW)	16.0	32.2	41.6	98	44.34	N.A
Clock Cycles	4	2	10	10	11	1
Frequency (MHz)	125	125	400	1320	110	100
Throughput (Gbps)	4	8	5.12	16.9	1.28	12.8
MTD	$\geq$ 100K	$\geq$ 100K	N.A	940K	$\geq$ 10M	N.A

The pipelined designs from this thesis have the lowest power consumption values compared to the other ones. [9] has lower area but since it is a more advanced technology, its smaller size is expected as transistors are also smaller. But it also has fewer gates, so it is more area efficient compared to this thesis. Throughput of the 64-bit pipelined design is greater compared to the [9]. Both 32-bit and 64-bit pipelined

designs have greater throughput compared to the [31] with higher operating frequency and lower power consumption. Both pipelined versions are much smaller in terms of gate count in contrast to [32]. Measurements-to-Disclosure (MTD) value is here taken as the number of traces required for the disclosure of all bytes. The correct byte should be distinguished from all the other wrong byte predictions as defined in [33]. The MTD values for this thesis are given with $\geq$ sign because not all bytes were disclosed as a first prediction. Collecting much higher traces would solve the issue but collecting traces from simulation programs needs much more time compared to collecting from hardware. In addition, MTD values are not properly documented for some papers. Therefore, MTD comparison is not clearly conclusive.

7. CONCLUSION

In this thesis, low power and power side-channel attack resistant AES IP is presented. Today, the attacks are becoming more and more diversified and intimidating. Security is threatened and continuously upgraded solutions are necessary. The designers face the challenge of preparing faster and more secure designs with a low power budget. Especially for battery-powered devices, power consumption is an important consideration. In the age of the Internet of Things (IoT), these two targets become critical, hence the topic of this thesis is relevant to the current demands.

Improvement to the side-channel resistance is provided by the obfuscating and pipelining. For obfuscating, a mask value, generated from LFSR, is XOR'ed with the plaintext. XOR output and mask are passed through ShiftRows and MixColumns operations independently. Then the outputs are XOR'ed again and the obtained value goes into AddRoundKey. The seed of the LFSR comes from the cipherkey. For pipelining, both 32-bit and 64-bit versions were constructed. At the same time, different inputs are processed at the different hardware blocks. Power consumption due to different inputs add up and result in complex total power consumption which complicates the side-channel attack by requiring more inputs to disclose the key. Both methods showed better side-channel attack resistance compared to the unprotected base design.

In total, five different versions were used and compared with each other. The rolled version uses single hardware for all rounds. So, at the rising edge of the clock, the output of the round is fed back to the same hardware. The obfuscated version uses the same approach as the rolled version but the round contains obfuscating. The third and fourth versions utilize pipelining by introducing additional hardware to the rolled version. The most area-consuming version unrolls all rounds. So, it is possible to encrypt the plaintext in one clock cycle excluding the delay in the interface.

The versions were designed using both FPGA and TSMC 65 nm ASIC flow. Their speed, area, power consumption, latency, and other statistics were compared. The total area, power consumption, utilization values showed that there is a consistent increase when we go from rolled version, to obfuscated, 32-bit pipelined, 64-bit pipelined, and unrolled version. The clock frequency of the ASIC flow is chosen as equal for all versions, 8 ns, except unrolled version, 40 ns. The latencies between consecutive encryption operations are 17 cycles, 17 cycles, 4 cycles, 2 cycles, and 8 cycles, respectively. The unrolled design has 0.415 mm^2 area, 80K standard cells, 21.5 mW power consumption, 25 MHz operating frequency, and 3.2 Gbps throughput. 64-bit pipelined design has 0.278 mm^2 area, 51K standard cells, 32.2 mW power consumption, 125 MHz operating frequency, and 8 Gbps throughput. 32-bit pipelined design has 0.208 mm^2 area, 36K standard cells, 16.0 mW power consumption, 125 MHz operating frequency, and 4 Gbps throughput. The side-channel attack resistance was evaluated by doing statistical analysis on simulation outputs which enables testing countermeasures before a costly tapeout process. According to the evaluation results unrolled version performed better against power side-channel attack but its high area and lower throughput compared to the pipelined versions make it infeasible in compact applications. Pipelining increases both side-channel attack resistance and throughput.

As a future work, fixing the power consumption can be stated. Though this is a known method, the setup can be established as follows: Cadence Voltus tool outputs are exported as a text file, the same as the file shown in Figure 5.4. Then the values are imported to the Simulation Program with Integrated Circuit Emphasis (SPICE) tool where they will represent current values pulled by a black box block. Then analog circuitry whose job is to stabilize current consumption is connected to the load. In this way, the attacker would see the input of the analog circuitry, instead of the actual power consumption of the cryptographic block. This scheme would harden the side-channel attack by loosening the correlation between executed operations and seen power consumption. The obfuscating method can also be improved with faster mask refreshing and more adaptive seed changing. More complex schemes would improve side-channel attack resistance.

REFERENCES

1. Courtois, N. and J. Pieprzyk, "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations", *Cryptology ePrint Archive*, p. 2, 2002.

2. Simmons, G. J., *Encyclopedia Britannica Cryptology*, 2016, https://www.britannica.com/topic/cryptology, accessed in February 2022.

3. Kehrer, P., *Cryptography.io Asymmetric Algorithms*, 2019, https://cryptography.io/en/latest/hazmat/primitives/asymmetric/index.html, accessed in February 2022.

4. Young, B., *Foundations of Computer Security Lecture 45: Stream and Block Encryption*, University of Texas at Austin Department of Computer Sciences, Austin, 2020.

5. Domnitser, L., N. Abu-Ghazaleh and D. Ponomarev, "A Predictive Model for Cache-Based Side Channels in Multicore and Multithreaded Microprocessors", *Proceedings of the 5th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, p. 70–85, Springer-Verlag, Berlin, Heidelberg, 2010.

6. Benvenuto, C. J., *Galois Field in Cryptography*, University of Washington Department of Mathematics, Seattle, 2012.

7. National Institute of Standards and Technology (NIST), *FIPS 197 Advanced Encryption Standard (AES)*, NIST, 2001.

8. National Institute of Standards and Technology (NIST), *NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation*, NIST, 2001.

9. Chou, Y.-H. and S.-L. L. Lu, "A High Performance, Low Energy, Compact Masked

128-Bit AES in 22nm CMOS Technology", *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–4, 2019.

10. Bogdanov, A., D. Khovratovich and C. Rechberger, "Biclique Cryptanalysis of the Full AES", D. H. Lee and X. Wang (Editors), *Advances in Cryptology – ASIACRYPT*, pp. 344–371, Springer, Berlin, Heidelberg, 2011.

11. National Institute of Standards and Technology Computer Security Resource Center, *Cryptographic Algorithm Validation Program CAVP*, 2022, https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation-search, accessed in February 2022.

12. Bassham, L. E., *The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)*, National Institute of Standards and Technology, 2002.

13. Zhou, Y. and D. Feng, "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing", *Cryptology ePrint Archive*, pp. 2, 22–23, 2005.

14. Kocher, P., J. Jaffe and B. Jun, "Introduction to Differential Power Analysis and Related Attacks", *Cryptography Research*, p. 3, 1998.

15. Kocher, P., J. Jaffe and B. Jun, "Differential Power Analysis", M. Wiener (Editor), *Advances in Cryptology — CRYPTO*, pp. 388–397, Springer, Berlin, Heidelberg, 1999.

16. Ors, S., F. Gurkaynak, E. Oswald and B. Preneel, "Power-Analysis Attack on An ASIC AES Implementation", *Proceedings of the International Conference on Information Technology: Coding and Computing*, Vol. 2, pp. 546–552, IEEE, Las Vegas, NV, 2004.

17. Brier, E., C. Clavier and F. Olivier, "Correlation Power Analysis with a Leakage Model", M. Joye and J.-J. Quisquater (Editors), *Cryptographic Hardware and*

Embedded Systems - CHES, pp. 16–29, Springer, Berlin, Heidelberg, 2004.

18. NewAE Technology, *Correlation Power Analysis*, 2018, https://wiki.newae.com/Correlation_Power_Analysis, accessed in February 2022.

19. INVIA, *Clock Randomization against Side Channel Attacks*, INVIA, Paris, 2021.

20. Alfke, P., *Efficient Shift Registers, LFSR Counters, and Long Pseudo- Random Sequence Generators*, 1.1st edition, p. 5, Xilinx, San Jose, 1996.

21. Sedra, A. S. and K. C. Smith, *Microelectronic Circuits*, 6th edition, pp. 362-366, Oxford University Press, New York, 2011.

22. National Institute of Standards and Technology, *Cryptographic Standards and Guidelines AES Development*, 2021, https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development, accessed in February 2022.

23. Xilinx, *Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for Schematic Designs*, 14.7th edition, Xilinx, 2013.

24. Xilinx, *UG906 Vivado Design Suite User Guide: Design Analysis and Closure Techniques*, 2nd edition, Xilinx, 2019.

25. Weste, N. H. and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd edition, pp. 189, 196, Pearson Education, 2005.

26. Iqbal, M. A., N. K. Macha, B. T. Repalle and M. Rahman, "Designing Crosstalk Circuits at 7nm", *IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–4, IEEE, San Mateo, CA, 2019.

27. DiBartolomeo, S., *All About Calma's GDSII Stream Format*, 2011,

https://artwork.com/gdsii/gdsii/index.htm, accessed in February 2022.

28. IEEE Computer Society, *IEEE Std 1497-2001, IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process*, IEEE, 2001.

29. O'Flynn, C., *CW1173: ChipWhisperer-Lite*, NewAE Technology, Dartmouth, 2018.

30. Lu, S., Z. Zhang and M. Papaefthymiou, "1.32GHz High-Throughput Charge-Recovery AES Core with Resistance to DPA Attacks", *Symposium on VLSI Circuits (VLSI Circuits)*, pp. C246–C247, IEEE, Kyoto, Japan, 2015.

31. Tokunaga, C. and D. Blaauw, "Secure AES Engine with A Local Switched-Capacitor Current Equalizer", *IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pp. 64–65,65a, IEEE, San Francisco, CA, 2009.

32. Peng, Y., H. Zhao, X. Sun and C. Sun, "A Side-Channel Attack Resistant AES with 500Mbps, 1.92pJ/Bit PVT Variation Tolerant True Random Number Generator", *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 249–254, IEEE, Bochum, Germany, 2017.

33. Tiri, K., D. Hwang, A. Hodjat, B. cheng Lai, S. Yang, P. Schaumont and I. Verbauwhede, "Prototype IC with WDDL and Differential Routing - DPA Resistance Assessment", *Cryptographic Hardware and Embedded Systems – CHES, 7th International Workshop*, pp. 354–365, Springer, Edinburgh, Scotland, 2005.

APPENDIX A: FPGA FLOW CONSTRAINTS

"..." is put to show the line break. The line which contains "..." and the one below it are normally written in the same line.

```
#set period 50; # for unrolled version.
set period 12; # for 64-bit pipelined version
#set period 11; #for rolled, obfuscated, 32-bit pipelined versions

create_clock -name main_clock -period ${period} -waveform ...
"0 [expr $period/2.0]" [get_ports clk_i]
set_clock_transition 0.2 [get_clocks main_clock]

set_input_delay -clock main_clock -add_delay -max 1.0 ...
[get_ports rst_ni]
set_input_delay -clock main_clock -add_delay -min 0 ...
[get_ports rst_ni]
set_input_delay -clock main_clock -add_delay -max 1.0 ...
[get_ports start_plain_text]
set_input_delay -clock main_clock -add_delay -min 0 ...
[get_ports start_plain_text]
set_input_delay -clock main_clock -add_delay -max 1.0 ...
[get_ports start_key]
set_input_delay -clock main_clock -add_delay -min 0 ...
[get_ports start_key]
set_input_delay -clock main_clock -add_delay -max 1.0 ...
[get_ports plain_text*]
set_input_delay -clock main_clock -add_delay -min 0 ...
[get_ports plain_text*]
set_output_delay -clock main_clock -add_delay -max 1.0 ...
```

```
[get_ports cipher_text*]
set_output_delay -clock main_clock -add_delay -min 0 ...
[get_ports cipher_text*]

set_load 1.4 [get_ports cipher_text*]

set_units -capacitance pF -current mA -voltage V -power mW ...
-resistance Ohm
```

APPENDIX B: KAT TEST BENCH

```
‘timescale 1ns / 1ps

import parameters_pkg::*;

module aes_kat_tb #(
) (

);
  ‘ifdef EXPANDED
    localparam PERIOD = 60;
  ‘else
    localparam PERIOD = 10;
  ‘endif
  localparam HALF_PERIOD = PERIOD/2;
  // Inputs
  reg          rst_ni;
  reg          clk_i;
  reg          start_plain_text;
  reg          start_key;
  reg   [31:0] plain_text;
  // Outputs
  logic [31:0] cipher_text;

  aes #(
  ) uut (
      .rst_ni           (rst_ni),
      .clk_i            (clk_i),
      .start_plain_text (start_plain_text),
```

```
    .start_key        (start_key),
    .plain_text       (plain_text),
    .cipher_text      (cipher_text)
);

integer i;
integer file_id;

reg [31:0] plain_text_memory [0:1135];
reg [31:0] cipher_key_memory [0:1135];

initial begin
    $readmemh("kat_plaintext.mem", plain_text_memory);
    $readmemh("kat_cipherkey.mem", cipher_key_memory);
    file_id = $fopen("output.txt", "w");
end

initial begin
  plain_text = 32'h0;
  clk_i = 1;
  rst_ni = 1;
  start_plain_text = 0;
  start_key = 0;
  i = 0;
  #100;
  rst_ni = 0;
  #100;
  rst_ni = 1;
  #100;
  // GFSbox Know Answer Test Values
  start_key = 1'b1;
```

```
plain_text = cipher_key_memory[0];
#PERIOD;
start_key = 1'b0;
plain_text = cipher_key_memory[1];
#PERIOD;
plain_text = cipher_key_memory[2];
#PERIOD;
plain_text = cipher_key_memory[3];
#(15*PERIOD);
for (i = 0; i < 7; i = i+1) begin
  start_plain_text = 1'b1;
  plain_text = plain_text_memory[0 + 4*i];
  #PERIOD;
  start_plain_text = 1'b0;
  plain_text = plain_text_memory[1 + 4*i];
  #PERIOD;
  plain_text = plain_text_memory[2 + 4*i];
  #PERIOD;
  plain_text = plain_text_memory[3 + 4*i];
  `ifdef EXPANDED
    #(2*PERIOD);
  `else
    #(11*PERIOD);
  `endif
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x\n", cipher_text);
```

```
  #(PERIOD-1);
end
// KeySbox Know Answer Test Values
start_plain_text = 1'b1;
plain_text = plain_text_memory[0 + 4*(i+1)];
#PERIOD;
start_plain_text = 1'b0;
plain_text = plain_text_memory[1 + 4*(i+1)];
#PERIOD;
plain_text = plain_text_memory[2 + 4*(i+1)];
#PERIOD;
plain_text = plain_text_memory[3 + 4*(i+1)];
#(15*PERIOD);
for (i = 7; i < 28; i = i+1) begin
  start_key = 1'b1;
  plain_text = cipher_key_memory[0 + 4*i];
  #PERIOD;
  start_key = 1'b0;
  plain_text = cipher_key_memory[1 + 4*i];
  #PERIOD;
  plain_text = cipher_key_memory[2 + 4*i];
  #PERIOD;
  plain_text = cipher_key_memory[3 + 4*i];
  `ifdef EXPANDED
    #(14*PERIOD);
  `else
    #(23*PERIOD);
  `endif
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
```

```
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x\n", cipher_text);
  #(PERIOD-1);
end
// VarTxt Known Answer Test Values
start_key = 1'b1;
plain_text = cipher_key_memory[0 + 4*(i+1)];
#PERIOD;
start_key = 1'b0;
plain_text = cipher_key_memory[1 + 4*(i+1)];
#PERIOD;
plain_text = cipher_key_memory[2 + 4*(i+1)];
#PERIOD;
plain_text = cipher_key_memory[3 + 4*(i+1)];
#(15*PERIOD);
for (i = 28; i < 156; i = i+1) begin
  start_plain_text = 1'b1;
  plain_text = plain_text_memory[0 + 4*i];
  #PERIOD;
  start_plain_text = 1'b0;
  plain_text = plain_text_memory[1 + 4*i];
  #PERIOD;
  plain_text = plain_text_memory[2 + 4*i];
  #PERIOD;
  plain_text = plain_text_memory[3 + 4*i];
  `ifdef EXPANDED
    #(2*PERIOD);
  `else
    #(11*PERIOD);
```

```
  `endif
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x", cipher_text);
  #(PERIOD-1);
  #1; $fwrite (file_id, "%x\n", cipher_text);
  #(PERIOD-1);
end
// VarKey Known Answer Test Values
start_plain_text = 1'b1;
plain_text = plain_text_memory[0 + 4*(i+1)];
#PERIOD;
start_plain_text = 1'b0;
plain_text = plain_text_memory[1 + 4*(i+1)];
#PERIOD;
plain_text = plain_text_memory[2 + 4*(i+1)];
#PERIOD;
plain_text = plain_text_memory[3 + 4*(i+1)];
#(15*PERIOD);
for (i = 156; i < 284; i = i+1) begin
  start_key = 1'b1;
  plain_text = cipher_key_memory[0 + 4*i];
  #PERIOD;
  start_key = 1'b0;
  plain_text = cipher_key_memory[1 + 4*i];
  #PERIOD;
  plain_text = cipher_key_memory[2 + 4*i];
  #PERIOD;
  plain_text = cipher_key_memory[3 + 4*i];
```

```
      `ifdef EXPANDED
        #(14*PERIOD);
      `else
        #(23*PERIOD);
      `endif
      #1; $fwrite (file_id, "%x", cipher_text);
      #(PERIOD-1);
      #1; $fwrite (file_id, "%x", cipher_text);
      #(PERIOD-1);
      #1; $fwrite (file_id, "%x", cipher_text);
      #(PERIOD-1);
      #1; $fwrite (file_id, "%x\n", cipher_text);
      #(PERIOD-1);
    end
    $fclose (file_id);
  end

  always begin
    #HALF_PERIOD clk_i = ~clk_i;
  end

endmodule
```

APPENDIX C: TEST BENCH (NO PIPELINE)

```
‘timescale 1ns / 1ps

import parameters_pkg::*;

module aes_tb #(
  parameter NumInputs = 100000
) (

);

  ‘ifdef EXPANDED
    localparam PERIOD = 40;
  ‘else
    localparam PERIOD = 8;
  ‘endif
  localparam HALF_PERIOD = PERIOD/2;
  // Inputs
  reg rst_ni;
  reg clk_i;
  reg start_plain_text;
  reg start_key;
  reg [31:0] plain_text;
  // Outputs
  wire [31:0] cipher_text;

  aes #(
  ) uut (
      .rst_ni(rst_ni),
```

```
    .clk_i(clk_i),
    .start_plain_text (start_plain_text),
    .start_key (start_key),
    .plain_text(plain_text),
    .cipher_text(cipher_text)
);

integer i;


reg [31:0] test_memory [0:(4*NumInputs -1)];
initial begin
    $readmemh("../DATA/SIM/input_hex_100000_matlab.mem", test_memory);
end

initial begin
  plain_text = 32'h0;
  clk_i = 1;
  rst_ni = 1;
  start_plain_text = 0;
  start_key = 0;
  i = 0;
  #(10*PERIOD);
  rst_ni = 0;
  #(10*PERIOD);
  rst_ni = 1;
  #(10*PERIOD);
  start_key = 1'b1;
  plain_text = 32'h2b7e1516;
  #PERIOD;
  start_key = 1'b0;
```

```
    plain_text = 32'h28aed2a6;
    #PERIOD;
    plain_text = 32'habf71588;
    #PERIOD;
    plain_text = 32'h09cf4f3c;
    #(10*PERIOD);

    for (i = 0; i < NumInputs; i = i+1) begin
      start_plain_text = 1'b1;
      plain_text = test_memory[0 + 4*i];
      #PERIOD;
      start_plain_text = 1'b0;
      plain_text = test_memory[1 + 4*i];
      #PERIOD;
      plain_text = test_memory[2 + 4*i];
      #PERIOD;
      plain_text = test_memory[3 + 4*i];
      `ifdef EXPANDED
        #(6*PERIOD);
      `else
        #(15*PERIOD);
      `endif
    end
  end

  always begin
    #HALF_PERIOD clk_i = ~clk_i;
  end

endmodule
```

APPENDIX D: TEST BENCH (PIPELINE)

```
`timescale 1ns / 1ps

import parameters_pkg::*;

module aes_tb #(
  `ifdef BIT_64
    parameter BusWidth = 64,
  `else
    parameter BusWidth = 32,
  `endif
  parameter NumInputs = 100000
) (

);

  `ifdef EXPANDED
    localparam PERIOD = 40;
  `else
    localparam PERIOD = 8;
  `endif
  localparam HALF_PERIOD = PERIOD/2;
  // Inputs
  reg rst_ni;
  reg clk_i;
  reg start_plain_text;
  reg start_key;
  reg [BusWidth-1:0] plain_text;
  // Outputs
```

```
logic [BusWidth-1:0] cipher_text;

aes #(

) uut (
  .clk_i            (clk_i),
  .rst_ni           (rst_ni),
  .start_plain_text (start_plain_text),
  .start_key        (start_key),
  .plain_text       (plain_text),
  .cipher_text      (cipher_text)
);

integer i;


reg [31:0] test_memory [0:NumInputs*4-1];
initial begin
    $readmemh("../DATA/SIM/input_hex_100000_matlab.mem", test_memory);
end

initial begin
  plain_text = '0;
  clk_i = 1;
  rst_ni = 1;
  start_plain_text = 0;
  start_key = 0;
  #(10*PERIOD);
  rst_ni = 0;
  #(10*PERIOD);
  rst_ni = 1;
```

```
#(10*PERIOD);

`ifdef BIT_64
  start_key = 1'b1;
  plain_text = 64'h2b7e151628aed2a6;
  #PERIOD;
  start_key = 1'b0;
  plain_text = 64'habf7158809cf4f3c;
  #(12*PERIOD);
  ////////////////////////
  for (i = 0; i < NumInputs; i = i+1) begin
    start_plain_text = 1'b1;
    plain_text = {test_memory[0 + 4*i], test_memory[1 + 4*i]};
    #PERIOD;
    start_plain_text = 1'b0;
    plain_text = {test_memory[2 + 4*i], test_memory[3 + 4*i]};
    #PERIOD;
  end
`else
  start_key = 1'b1;
  plain_text = 32'h2b7e1516;
  #PERIOD;
  start_key = 1'b0;
  plain_text = 32'h28aed2a6;
  #PERIOD;
  plain_text = 32'habf71588;
  #PERIOD;
  plain_text = 32'h09cf4f3c;
  #(11*PERIOD);
  ////////////////////////
  for (i = 0; i < NumInputs; i = i+1) begin
```

```
        start_plain_text = 1'b1;
        plain_text = test_memory[0 + 4*i];
        #PERIOD;
        start_plain_text = 1'b0;
        plain_text = test_memory[1 + 4*i];
        #PERIOD;
        plain_text = test_memory[2 + 4*i];
        #PERIOD;
        plain_text = test_memory[3 + 4*i];
        #PERIOD;
      end
    `endif
  end

  always begin
    #HALF_PERIOD clk_i = ~clk_i;
  end

endmodule
```