Area and Energy Optimizations in ASIC Implementations of AES and PRESENT Block Ciphers

by

Jenny Yu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Applied Science
in
Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

When small, modern-day devices surface with neoteric features and promise benefits like streamlined business processes, cashierless stores, and autonomous driving, they are all too often accompanied by security risks due to a weak or absent security component. In particular, the lack of data privacy protection is a common concern that can be remedied by implementing encryption. This ensures that data remains undisclosed to unauthorized parties. While having a cryptographic module is often a goal, it is sometimes forfeited because a device's resources do not allow for the conventional cryptographic solutions. Thus, smaller, lower-energy security modules are in demand. Implementing a cipher in hardware as an application-specific integrated circuit (ASIC) will usually achieve better efficiency than alternatives like FPGAs or software, and can help towards goals such as extended battery life and smaller area footprint.

The Advanced Encryption Standard (AES) is a block cipher established by the National Institute of Standards and Technology (NIST) in 2001. It has since become the most widely adopted block cipher and is applied in a variety of applications ranging from smartphones to passive RFID tags to high performance microprocessors. PRESENT, published in 2007, is a smaller lightweight block cipher designed for low-power applications.

In this study, low-area and low-energy optimizations in ASICs are addressed for AES and PRESENT. In the low-area work, three existing AES encryption cores are implemented, analyzed, and benchmarked using a common fabrication technology (STM 65 nm). The analysis includes an examination of various implementations of internal AES operations and their suitability for different architectural choices. Using our taxonomy of design choices, we designed Quark-AES, a novel 8-bit AES architecture. At 1960 GE, it features a 13% improvement in area and 9% improvement in throughput/area^2 over the prior smallest design. To illustrate the extent of the variations due to the use of different ASIC libraries, Quark-AES and the three analyzed designs are also synthesized using three additional technologies. Even for the same transistor size, different ASIC libraries produce significantly different area results. To accommodate a variety of applications that seek different levels of tradeoffs in area and throughput, we extend all four designs to 16-bit and 32-bit datawidths.

In the low-energy work, round unrolling and glitch filtering are applied together to achieve energy savings. Round unrolling, which applies multiple block cipher rounds in a combinational path, reduces the energy due to registers but increases the glitching energy. Glitch filtering complements round unrolling by reducing the amount of glitches and their

associated energy consumption. For unrolled designs of PRESENT and AES, two glitch filtering schemes are assessed. One method uses AND-gates in between combinational rounds while the other used latches. Both methods work by allowing the propagation of signals only after they have stabilized. The experiments assess how energy consumption changes with respect to the degree of unrolling, the glitch filtering scheme, the degree of pipelining, the spacing between glitch filters, and the location of glitch filters when only a limited number of them can be applied due to area constraints. While in PRESENT, the optimal configuration depends on all the variables, in a larger cipher such as AES, the latch-based method consistently offers the most energy savings.

Acknowledgements

I would like to thank my supervisor Dr. Mark Aagaard for his guidance and continued support throughout my undergraduate and graduate careers. His attentive and caring nature made my time at the University of Waterloo enjoyable and gratifying. I would also like to thank Nusa Zidaric for her lessons on finite fields and for her help on constructing the transformation matrices to perform a change of basis. It was transformative.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES Advanced Encryption Standard 2

ASIC application-specific integrated circuit 4

BPU Byte Permutation Unit 41

CBC Cipher Block Chaining 16

CFB Cipher Feedback 16

CTR Counter 16

DES Data Encryption Standard 2

DUT device under test 28

ECB Electronic Codebook 16

EDA electronic design automation 34

FPGA field-programmable gate array 4

GE gate equivalent 6

GF Galois field 9

HDL hardware description language 4

IoT Internet of Things 3

IV initialization vector 16

LFSR linear feedback shift register 39

mux multiplexer 39

NIST National Institute of Standards and Technology 2

OFB Output Feedback 16

P-box permutation box 14

RFID radio-frequency identification 2

RTL register-transfer level 4

S-box substitution box 14

SP-network substitution-permutation network 14

UUT unit under test 28

VCD value change dump 29

VHDL Very High Speed Integrated Circuit Hardware Description Language 28

Chapter 1

Introduction

1.1 Cryptography

Cryptography saw its beginnings thousands of years ago. Coming from two Greek words 'krypto', meaning 'secret', and 'graphene', meaning 'writing' [1], it refers to the use of secret codes to conceal meaning of information. One of the earliest known cryptographic techniques is the Caesar cipher, which substitutes each letter by another letter some fixed number of positions down the alphabet. A more sophisticated encryption device is the Enigma machine, used by the Germans in World War II to protect military communication. In modern days, cryptography is a branch of information security that concerns itself with the protection of digital information in computer systems. Encryption is the process of converting readable information, called plaintext, into unintelligible form, called ciphertext, while decryption is the reverse process. A pair of encryption and decryption algorithms constitutes a cipher. The transformation of plaintext into ciphertext (and vice versa) usually depends on the algorithm as well as a cryptographic key. Because it is often difficult to keep the details of an algorithm secret, the security of an encryption system relies on the secrecy of the key.

Information security has three primary objectives: confidentiality, integrity, and availability. Cryptography is a tool used primarily to attain confidentiality; it ensures that the meaning of a message remains undisclosed to unauthorized parties. Cryptography can be decomposed into two subcategories: symmetric-key and asymmetric key cryptography. Whereas asymmetric-key ciphers use different keys for the encryption and decryption

processes, symmetric-key algorithms use the same key for both operations. Furthermore, symmetric key algorithms can be implemented as stream ciphers or block ciphers. Stream ciphers generate a key stream which gets combined with the plaintext, while block ciphers operate on a fixed size block using a fixed size key. Two block ciphers, AES and PRESENT, are the focus of this work. The process of breaking and finding weaknesses in a cipher is called cryptanalysis. Both cryptography and cryptanalysis fall under the broader umbrella term, cryptology. The focus of this thesis is on cryptography and not cryptanalysis.

The Advanced Encryption Standard (AES) is a widely accepted encryption specification established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It became the successor to the Data Encryption Standard (DES), which was becoming vulnerable to attacks due to weaknesses in its algorithm and its small 56-bit key size. The selection of the AES algorithm was a three-year process lasting from 1997 to 2000 that began as a call, made by NIST, for a new encryption algorithm to succeed DES. NIST opened its doors to designs from the cryptographic community and welcomed submissions from around the globe. Among fifteen submissions from several different countries, the Rijndael cipher, submitted by two Belgian cryptographers (Vincent Rijmen and Joan Daemem), was selected as the winner. AES supports a block size of 128-bits and three key sizes: 128, 192, and 256 bits.

Although AES has become the most popular block cipher, finding use in consumer phones and laptops, it is not always suitable for extremely constrained environments such as radio-frequency identification (RFID) tags and sensor networks. To address this issue, an ultra-lightweight block cipher called PRESENT was developed. Its design was published in 2007 by the Orange Labs, Ruhr University Bochum, and the Technical University of Denmark. Designed for low-power consumption, PRESENT supports a 64-bit block size and key sizes of 80 and 128 bits.

1.2 Motivation

As technology advances, it brings creative solutions to existing problems in society, but it can also inadvertently introduce new security issues that are vulnerable to exploitation. The consequences of a security breach can range from theft of personal information to invasion of privacy to something as fatal as a car crash. Regrettably, there is no shortage of news reports recounting some of the sinister and unwanted effects.

In the 2015 Black Hat conference, security researchers Charlie Miller and Chris Valasek revealed they were able to take control of a Jeep Cherokee by remotely hacking the entertainment system through a cellular network [2], [3]. They gained control over the car's brakes, accelerator, radio, horn, and windshield wipers. According to a senior security researcher at Kaspersky Lab, the hack succeeded in part due to weak authentication algorithms and a possible incorrect implementation of a cryptographic component [3]. Another security breach that made headlines occurred in baby monitors, which have developed into sophisticated WiFi-enabled, smart devices. TRENDnet's baby monitoring webcams contained faulty software that enabled a person to look and listen through it if he or she obtained the camera's IP address [4]. In early 2017, it was revealed that St. Jude Medical's cardiac devices could allow a hacker to access the device and deplete the battery or administer incorrect pacing or shocks [4]. It was the transmitter, which reads the device's data and remotely shares it with physicians, that presented a vulnerability.

These examples illustrate the rise of the Internet of Things (IoT), a network of devices connected to the Internet. More importantly, however, these examples show how difficult it is to build safe and secure smart devices. Being connected to the Internet introduces a new set of challenges, whose solutions are still in their infancy. The use of radio frequency identification (RFID) tags has emerged with great popularity and has a myriad of applications including inventory tracking, wireless sensor networks, and IoT. In 2016, research showed that 73% of retailers had implemented RFID to minimize out-of-stock situations and provide real-time merchandise location data [5]. The increase in these kinds of devices implies a need for smaller, lower-energy security modules.

The motivation for implementing security in new devices is now clear. Yet, there are several reasons why security is not implemented in a system. There could be a lack of awareness from higher management. Perhaps getting the product to market as quickly as possible is a criterion that trumps security. Other reasons include a lack of means or resources; having a security component could be too costly, result in too much energy consumption, or have too large an area footprint. The work in this thesis alleviates some of the latter challenges related to energy and area.

Designing security components for small devices entails a different set of difficulties compared to designing for resource-abundant devices such as laptops and mobile phones. Although encryption standards do exist, the implementation details of a cipher, whether in software or in hardware, depend on the application and are left to the designer's discretion. Software solutions may be suitable for general purpose processors, which are relatively abundant in resources. Resource-constrained devices have higher efficiency demands and

often implement hardware solutions. Generally, dedicated hardware circuits can offer better performance and lower power consumption.

As a product improves and matures, specifications and requirements can change, which may require modifications to an implementation. To keep up with continual change and the ever-decreasing size of devices, security components need to be continually enhanced as well, to provide higher throughput, lower energy and power consumption, lower area, lower cost *etc*. With fast moving technology, it is beneficial to have a configurable design to reduce development time as well as achieve desired tradeoffs.

1.3 Thesis Overview

Efficient digital hardware implementations of AES and PRESENT have been the topic of many previous works. These works include both field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) implementations, with objectives ranging from high performance to low power and energy to low area. An ASIC is an integrated circuit designed for a specific function rather than for general-purpose use. These contrast with FPGAs, which are programmable logic blocks that can be reconfigured to be used in many different applications. In this thesis, the cipher designs are implemented at the register-transfer level (RTL) using a hardware description language (HDL) to describe the behaviour of circuits.

In the years closely following the inception of the AES standard, research work relating to it focused primarily on throughput optimizations, while, recently, low-area designs have received more interest. Low-area designs can take on many different kinds of architectures and employ different implementations of internal AES functions. While a straight-forward implementation of the algorithm processes one block (128 bits) every clock cycle, a low-area design would usually take on a serial approach, adopting a smaller datawidth. For instance, a design might operate on only 16-bits per clock cycle, requiring 8 clock cycles to process one round of encryption. In general, a smaller datawidth will result in lower area, at the expense of increased latency.

One of the first low-area implementation of AES appeared in the work of Feldhofer *et al*. in 2005 [6], which set a benchmark for future AES low-area designs. The authors developed an 8-bit design with an area of 3400 GE and encryption latency of over 1000 clock cycles. Hamalainen *et al*. [7] also developed an 8-bit encryption core, which has an area of

3100 GE and a latency of 176 clock cycles. In 2011, Moradi *et al.* [8] achieved 2400 GE for an encryption module by employing a row-major design, instead of the standard column-major architecture. The first sub-2000 GE encryption design came from Mathew *et al.*[9], who reported an area of 1947 GE with a latency of 336 clock cycles. Atomic-AES, developed by Banik *et al.* [10], is a dual functionality core (*i.e.* it supports encryption and decryption) built upon the work of Moradi and its area is 2060 GE.

The low-area work in this thesis has two parts. The first part is an analysis and benchmark of existing architectures. Comparing existing works based on their reported area values may be unreliable because of the different tools and ASIC cell libraries used for each design. In this work, three existing low-area encryption architectures are implemented on the same frabrication technology and synthesized with the same tools and settings so that the comparison is more objective. The second part of this work describes a novel 8-bit architecture, called Quark-AES, which is 13% smaller than the lowest area design indicated by our benchmark in part one. Quark-AES combines the best features from Hamalainen's [7], Moradi's [8], Jarvinen's [11], and Ahmed's [12] works into a low-area and low-latency architecture.

The other element of this thesis pertains to energy optimizations in both AES and PRESENT. In recent years, low energy has also become an important goal. To this end, previous works have applied techniques such as round unrolling (a form of loop unrolling) and operand gating to achieve energy savings. Its application to encryption modules is called round unrolling, which performs unrolling at the granulairty of block cipher round. Since this reduces the number of required registers, it should subsequently reduce the overall energy consumption. However, this is not always the case because round unrolling increases the number of glitches and thus, the glitching energy. Operand gating minimizes energy by selectively blocking the propagation of glitches. Banik *et al.* [13] were the first to explore the energy efficiency of loop unrolling in block ciphers and provided an analysis of energy consumption under various degrees of round unrolling in ASICs. While round unrolling proves to be effective for lower degrees of unrolling (as a result of a decrease in energy consumption of registers), higher degrees of unrolling increases glitching power. With two opposing forces, there exists an optimal number of rounds within a clock cycle that minimizes energy consumption.

Operand gating complements round unrolling by effectively removing some glitches and their associated energy consumption. Banik *et al.* [14] introduced a glitch reduction technique called Round Gating, which applies AND gates between combinational rounds. The AND gates are driven by *enable* signals, which assert only after enough time has passed to allow the signal of a previous round to stabilize. In a similar technique called

Combinational Checkpointing, the authors use latches in place of AND gates [15] and compare their technique against Round Unrolling. In this thesis, both Round Gating (AND-gate glitch filter) and Combinational Checkpointing (latch glitch filter) are compared against a baseline in pipelined and non-pipelined designs of AES and PRESENT. The assessment metrics include energy, area, and throughput.

In a design, high throughput, low area, low power, and low energy are all desirable, but it is impossible to optimize all aspects simultaneously and therefore tradeoffs have to be made. This can be a difficult process because 1) there are multiple parameters to take into consideration and 2) a different version of RTL code is required for each design/-circuit/configuration. Using a parameterized design, provided by this thesis, can reduce some of the work and allow a designer to choose the best design by simply passing different parameters. That is, one version of the RTL code supports all the possible configurations, and different circuits can be generated based on the parameters passed to the tools during synthesis. As a result, it becomes very easy to tune a design to achieve the desired tradeoffs among energy, throughput, and area.

1.4 Contributions

There are many low-area AES implementations in the research community, and simply comparing their reported area results can be unreliable and misguiding because the notion of a gate equivalent (GE) varies for different ASIC libraries. A gate equivalent is a unit of measure that is equivalent to the area of a two-input NAND gate. The same design synthesized using different ASIC libraries can result in significantly different results due to differences in the relative size of a circuit component to the NAND-gate. To remove this bias, we provide an analysis and benchmark of existing 8-bit AES encryption cores on a common technology. The rank of the area results of existing works differs on our benchmark compared to the rank based on the reported values, which indicates that the ASIC library used plays a significant role in the area results. The architectural analysis presented in this thesis is developed from our iterations and variations of existing works, from which the insights and lessons learned can be beneficial to other AES designers.

We also contribute our own encryption architecture, Quark-AES, which offers a 13% improvement in area and 8% improvement in throughput/area^2 compared to the next smallest design. Quark-AES can find use in applications requiring a small area footprint,

such as wearable devices, RFID tags, and sensor networks. Moreover, since power consumption is often directly proportional to area, Quark-AES can also be used for low-power applications.

In the territory of low-energy optimizations, a comparison of two glitch filtering techniques for AES and PRESENT on ASICs is provided. The latch glitch filtering technique has not been demonstrated on the PRESENT and AES-128 ciphers before, so we offer this addition and compare the results to our results for the AND-gate glitch filtering method. While prior work explored the energy consumption as a function of the number of unrolled rounds and as a function of the spacing between the glitch filters, they did not explore the optimal placement of glitch filters when a limited number of them are available for use. The work in this thesis investigates this problem using PRESENT and provides empirical results for the optimum placement of 1, 2, and 3 glitch filters in 8 unrolled rounds. Consequently, we illustrate how to achieve large energy savings with less area overhead than other works have demonstrated. Part of our energy analysis shows the energy distribution of the message signals in different unrolled rounds across time, which is an important step towards building an analytical model for glitch filtering in future work. Finally, the effect of energy on designs that are both unrolled and pipelined is explored, which has not been done before. With the countless number of configurations resulting from the large number of variables at play, this thesis can be used as a reference to select a design to achieve the desired trade-offs in area and throughput while having low energy as the primary goal.

1.5 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 presents the background knowledge required for understanding the work. It includes some mathematical background on finite fields, a description of the AES and PRESENT algorithms, and an overview of related works. Chapter 3 offers a discussion of FPGA vs ASIC, and explains the evaluation metrics of throughput, area, and energy. In Chapter 4, a benchmark and analysis of three existing 8-bit AES encryption cores is presented. Using our taxonomy of design choices, a novel low-area and low-latency 8-bit AES encryption design is developed. The energy optimizations for PRESENT and AES are discussed in Chapter 5, which examines the efficacy of AND-gate and latch glitch filtering techniques when they are used in conjunction with round unrolling. Finally, a summary of the work in this thesis and recommendations for future work are presented in Chapter 6.

Chapter 2

Background and Related Work

Modern cryptography has matured from its early techniques into an advanced subject based on mathematical foundations. To have a proper understanding of cryptographic concepts, some knowledge of finite fields, presented in Section 2.1, is required. Section 2.2 provides background information on block ciphers, including block cipher modes, the AES encryption algorithm, and the PRESENT encryption algorithm. Finally, existing works related to low area and low energy are highlighted in Section 2.3.

2.1 Finite Fields

2.1.1 Definitions and Properties

Before introducing the concept of finite fields, it is necessary to understand what a field is. A *field* is a set of elements on which addition, subtraction, multiplication, and division are defined and which satisfy a number of properties. More formally, it is defined as follows.

Definition 2.1.1 *A **field**, $\mathbb{F}$, is a set of elements on which two operations addition, "+", and multiplication, "·", are defined, and which satisfies the following field axioms:*

1. *Commutativity of addition: For all $a, b \in \mathbb{F}$, $a + b = b + a$*

2. *Associativity of addition: For all* $a, b, c \in \mathbb{F}$, $a + (b + c) = (a + b) + c$
3. *Additive identity: There exists an element* $0 \in \mathbb{F}$ *such that* $a + 0 = 0 + a = a$ *for all* $a \in \mathbb{F}$
4. *Additive inverse (subtraction): For any* $a \in \mathbb{F}$, *there exists* $b \in \mathbb{F}$ *such that* $a + b = b + a = 0$
5. *Commutativity of multiplication: For all* $a, b \in \mathbb{F}$, $a \cdot b = b \cdot a$
6. *Associativity of multiplication: For all* $a, b, c \in \mathbb{F}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
7. *Multiplicative identity: There exists an element* $1 \in \mathbb{F}$ *such that* $a \cdot 1 = 1 \cdot a = a$ *for all* $a \in \mathbb{F}$
8. *Multiplicative inverse (division): For any* $a \in \mathbb{F}$ *such that* $a \neq 0$, *there exists* $a^{-1} \in \mathbb{F}$ *such that* $a \cdot a^{-1} = a^{-1} \cdot a = 1$.
9. *Distributivity: For all* $a, b, c \in \mathbb{F}$, $a \cdot (b + c) = a \cdot b + a \cdot c$
10. *Closure: For all* $a, b \in \mathbb{F}$, *it holds that* $a + b = c \in \mathbb{F}$ *and* $a \cdot b = d \in \mathbb{F}$

Some examples of fields are the set of real numbers, $\mathbb{R}$, the set of complex numbers, $\mathbb{C}$, and the set of rational numbgers, $\mathbb{Q}$, all of which have an infinite number of elements. A field with a finite number of elements is called a *finite field*, denoted $GF(p^n)$ or $\mathbb{F}_{p^n}$, where p is prime and n is a positive integer. A finite field is also called a Galois field (GF), in honour of its discoverer, Evariste Galois. The *order* of a field refers to the number of elements in it and equals p^n. If $n = 1$, then the finite field is called a *prime field*. The elements of the prime field $GF(p)$ can be represented by the integers $0, 1, \ldots, p - 1$. For example, the field $GF(5) = \{0, 1, 2, 3, 4\}$. Operations on this field are performed modulo p (*e.g.* $4 + 2 = 6 \mod 5 = 1$ and $4 \cdot 2 = 8 \mod 5 = 3$).

Of particular interest is the finite field of order 2, $GF(2) = \{0, 1\}$, in which arithmetic is performed modulo 2. The addition and multiplication operations are summarized in the following tables.

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

The field $GF(2)$ is important from a hardware perspective as addition in $GF(2)$ is equivalent to the binary XOR operation, and multiplication in $GF(2)$ is equivalent to the binary AND operation. In what follows, only finite fields with $p = 2$ will be considered.

When $n > 1$, the field $GF(p^n)$ is called an *extension field*. For such fields, the representation of field elements and the arithmetic rules differ from those of prime fields. The elements of an extension field $GF(2^n)$ are represented as polynomials that have coefficients in $GF(2)$ and a maximum degree of $n-1$. For instance, each element $A \in GF(2^4)$ is represented as

$$A(x) = a_3x^3 + a_2x^2 + a_1x + a_0, \qquad a_i \in GF(2)$$

Since there are four coefficients, each of which can take on two values, there are $2^4 = 16$ unique polynomials that make up the finite field $GF(2^4)$. As a concise notation, an element can be represented by an n-bit vector of its coefficients:

$$A = (a_3, a_2, a_1, a_0)$$

For example, $(1, 0, 1, 1)$ represents the polynomial $x^3 + x + 1$.

2.1.2 Arithmetic

Addition and subtraction in extension fields are performed like standard polynomial addition and subtraction. In other words, the coefficients of same powers of x are simply added or subtracted modulo 2. Formally, the operations are given by Definition 2.1.2.

Definition 2.1.2 *[16] Let $A(x), B(x) \in GF(2^n)$. The **sum** of the two elements is then computed as*

$$C(x) = A(x) + B(x) = \sum_{i=1}^{n-1} c_i x^i, \quad c_i \equiv a_i + b_i \mod 2$$

*The **difference** is computed as*

$$C(x) = A(x) - B(x) = \sum_{i=1}^{n-1} c_i x^i, \quad c_i \equiv a_i - b_i \equiv a_i + b_i \mod 2$$

Multiplication in finite fields is not as straightforward as addition. Recall from the definition of a field that the closure property must be satisfied. When two polynomials are multiplied together, the resulting polynomial may have a degree that is higher than $n-1$ and thus, it must be reduced so that it remains an element in the field. The reduction is performed by dividing the result of the multiplication by an *irreducible polynomial*, $P(x)$, and taking only the remainder (*i.e.* modulo operation). The definition of an irreducible polynomial is given in Definition 2.1.3. It is similar to the concept of a prime number, whose factors are only 1 and itself.

Definition 2.1.3 *A polynomial, $P(x)$, is **irreducible** over a field F if it cannot be factored into the product of two or more smaller-degree polynomials whose coefficients are in F.*

To reduce a polynomial to degree less than or equal to $n-1$, the irreducible polynomial must have degree n. There may exist many possible irreducible polynomials for a given degree. In general, finding an irreducible polynomial is a non-trivial task [17]. For degrees less than 20, irreducible polynomials can be found in reference tables [18]. Note that the irreducibility of a polynomial depends on the field over which it is defined. For instance, $x^4 + x^3 + 1$ is irreducible over $GF(2)$. However, over $GF(2^4)$, it is reducible.

The multiplication of two elements in an extension field is defined in Definition 2.1.4.

Definition 2.1.4 *[16] Let $A(x), B(x) \in GF(2^n)$ and let $P(x) = \sum_{i=0}^{n} p_i x^i, p_i \in GF(2)$ be an irreducible polynomial. **Multiplication** of the two elements $A(x)$, $B(x)$ is computed as*

$$C(x) = A(x) \cdot B(x) \mod P(x)$$

An example of the multiplication of two finite field elements is given in Example 2.1.1.

Example 2.1.1 *Given $A(x) = x^2 + x$ and $B(x) = x^2 + 1$ such that $A(x), B(x) \in GF(2^3)$ and irreducible polynomial $P(x) = x^3 + x + 1$, find $C(x) = A(x) \cdot B(x) \mod P(x)$.*

$$\begin{aligned} C(x) &= A(x) \cdot B(x) \mod P(x) \\ &= (x^2 + x) \cdot (x^2 + 1) \mod P(x) \\ &= x^4 + x^3 + x^2 + x \mod P(x) \end{aligned} \tag{1}$$

$$\begin{array}{r|l} & x + 1 \\ \hline x^3 + x + 1 & x^4 + x^3 + x^2 + x \\ & -x^4 \quad\quad -x^2 - x \\ \hline & x^3 \\ & -x^3 \quad\quad -x - 1 \\ \hline & -x - 1 \end{array}$$

From Definition 2.1.2, $-x - 1 = x + 1$. Therefore, $C(x) = x + 1$.

The inverse operation in finite field arithmetic is defined in Definition 2.1.5. It is a core operation in many cipher algorithms. The inverse of an element can be found by a brute-force approach (multiply by every element until the product equals one), by using the extended Euclidean algorithm [16, Chapter 6.2.3], by exploiting Fermat's Little Theorem [16, Chapter 6.3.4], or by using lookup tables containing precomputed inverses.

Definition 2.1.5 *[16] For a given element $A(x) \in GF(2^n)$ and irreducible polynomial $P(x)$, the **inverse**, $A^{-1}(x)$, of $A(x)$ is defined as:*

$$A^{-1}(x) \cdot A(x) = 1 \mod P(x)$$

2.1.3 Construction of $GF(2^n)$ Field

For a given order p^n, there is only one finite field. However, there may exist many isomorphic mappings of that field; that is, different element representations. Loosely speaking, an isomorphism is a map that preserves the properties, operations, and relations of the set.

An element is a *primitive element* or *generator* if its powers generate all the nonzero elements of a field. More formally, an element α is primitive if $\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^n-2}\}$ is the set of all elements of the field $GF(p^n)$. Every finite field has a primitive element. A naive method for finding a primitive element is to pick an element and compute its powers (up to power p^n) while adding them to a set [19]. If the cardinality of the set equals $p^n - 1$, then a primitive element is found. If not, pick another element and repeat the process. Finding a primitive element efficiently is, in itself, an area of research [20].

A *primitive polynomial* is an irreducible polynomial whose roots are primitive elements. When the irreducible polynomial of a finite field is primitive, then the field can be constructed as follows.

Let γ be a root of the primitive polynomial (*i.e.* $P(\gamma) = 0$). Then, all the non-zero elements of the field can be generated recursively by the relation $\gamma^j = \gamma^{j-1} \cdot \gamma, j \in \mathbb{Z} > 0$, where the result is reduced when necessary. The construction of $GF(2^3)$ using two different primitive polynomials is shown in Table 2.1. The field constructed using $P_1(x) = x^3 + x + 1$ is isomorphic to the field constructed using $P_2(x) = x^3 + x^2 + 1$. Let α be a root of P_1 and β be a root of P_2. Since $P_1(\alpha) = \alpha^3 + \alpha + 1 = 0$, then $\alpha^3 = \alpha + 1$. In the same manner, $P_2(\beta) = \beta^3 + \beta^2 + 1 = 0$, so $\beta^3 = \beta^2 + 1$.

Table 2.1: Construction of $GF(2^3)$ using irreducible polynomials $P_1(x) = x^3 + x + 1$ and $P_2(x) = x^3 + x^2 + 1$

$P_1(x) = x^3 + x + 1$, $P_1(\alpha) = 0$			$P_2(x) = x^3 + x^2 + 1$, $P_2(\beta) = 0$		
Powers of α	**Polynomials in α**	**Vector repres. $(\alpha^2, \alpha, 1)$**	**Powers of β**	**Polynomials in β**	**Vector repres. $(\beta^2, \beta, 1)$**
0	0	(0,0,0)	0	0	(0,0,0)
1	1	(0,0,1)	1	1	(0,0,1)
α	α	(0,1,0)	β	β	(0,1,0)
α^2	α^2	(1,0,0)	β^2	β^2	(1,0,0)
α^3	$\alpha + 1$	(0,1,1)	β^3	$\beta^2 + 1$	(1,0,1)
α^4	$\alpha^2 + \alpha$	(1,1,0)	β^4	$\beta^2 + \beta + 1$	(1,1,1)
α^5	$\alpha^2 + \alpha + 1$	(1,1,1)	β^5	$\beta + 1$	(0,1,1)
α^6	$\alpha^2 + 1$	(1,0,1)	β^6	$\beta^2 + \beta$	(1,1,0)

For more information on finite fields, *Understanding Cryptography: A Textbook for Students and Practitioners* [16], *Handbook of Finite Fields* [21], and *Finite Fields* [22] are excellent resources.

2.2 Block Ciphers

Symmetric-key cryptography can be achieved by either stream ciphers or block ciphers. This work concerns only with the latter, whose architecture is described in Section 2.2.1. Block cipher modes of operation are explained in Section 2.2.2. Section 2.2.3 explains the AES algorithm and Section 2.2.4 the PRESENT algorithm.

2.2.1 Overview

A *block cipher* applies a deterministic algorithm to a fixed-size block of bits at a time. The encryption algorithm, *enc*, is a function of the message, M, and key, K, and produces a ciphertext, C, as follows:

$$C = enc(M, K)$$

Decryption, *dec*, takes the ciphertext, C, and key, K, as inputs, and outputs the original plaintext message, M:

$$M = dec(C, K)$$

The internal structure of a block cipher encryption or decryption algorithm generally consists of several iterations of a *round function*. Each round uses a *round key*, derived from K using a key generation algorithm, to transform the message. The security of the cipher is determined by the length of the key.

There are several design models for a block cipher, including substitution-permutation networks (SP-networks), Feistel ciphers, and Lai-Massey ciphers [23, Chapter 1]. The two ciphers presented in this thesis, AES and PRESENT, are SP-networks. In substitution-permutation networks, the round function consists of a substitution stage followed by a permutation stage, shown in Figure 2.1. This arrangement satisfies the *confusion* and *diffusion* properties which Claude Shannon, known as "the father of information theory", described in his 1949 paper titled *Communication Theory of Secrecy Systems* [24]. Confusion refers to the fact that each bit of the ciphertext should depend on several parts of the key. To satisfy diffusion, if a single bit of the plaintext is changed, then half of the bits in the ciphertext should change. The following is an excerpt from Shannon's paper:

> In the method of diffusion the statistical structure of M[the message] which leads to its redundancy is "dissipated" into long range statistics—*i.e.*, into statistical structure involving long combinations of letters in the cryptogram. ... The method of confusion is to make the relation between the simple statistics of E [the encrypted message] and the simple description of K[the key] a very complex and involved one. [24]

The substitution stage is implemented using substitution boxes (S-boxes). Each S-box substitutes a small block of bits (eg. 8-bits) with another block of bits following certain rules. It is a non-linear one-to-one mapping which achieves Shannon's confusion property; a small change in the input effects a significant change in the output. The permutation stage is realized by a number of permutation boxes (P-boxes). This component takes a block of bits and outputs a rearrangement (permutation) of the given bits. The operation is linear and it ensures Shannon's diffusion property, dissipating redundancies. In addition to the substitution and permutation layers, at each round, the message is combined with the round key, usually by an XOR operation.

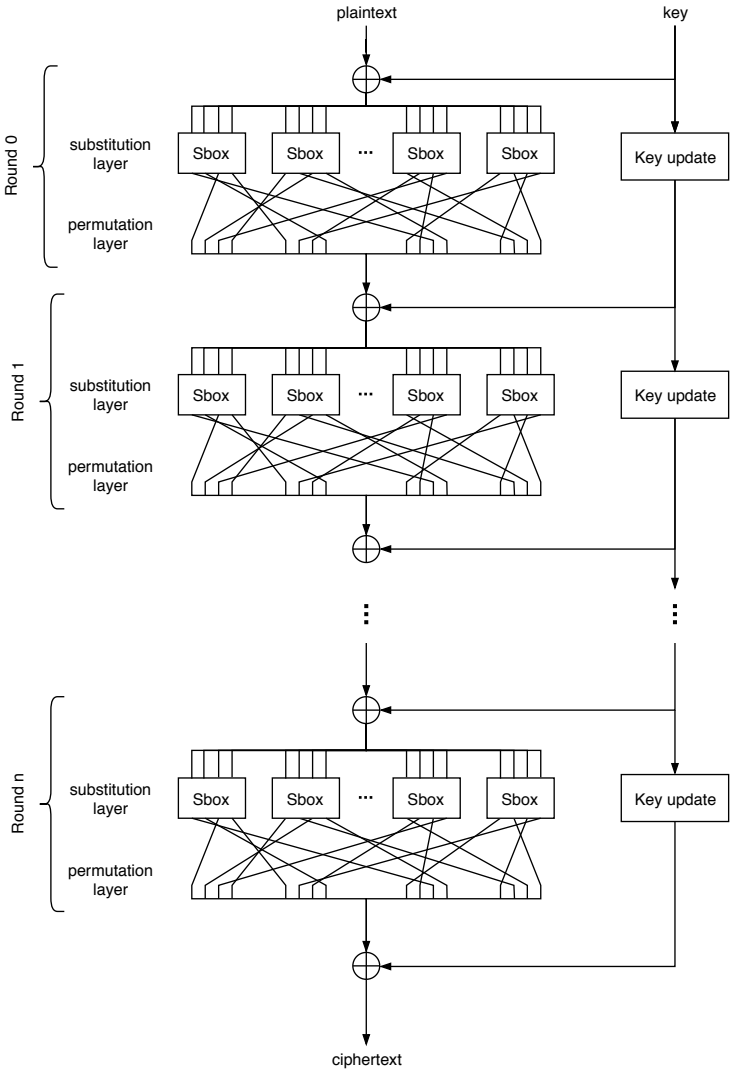


Figure 2.1: Substitution-permutation network structure

2.2.2 Modes of Operation

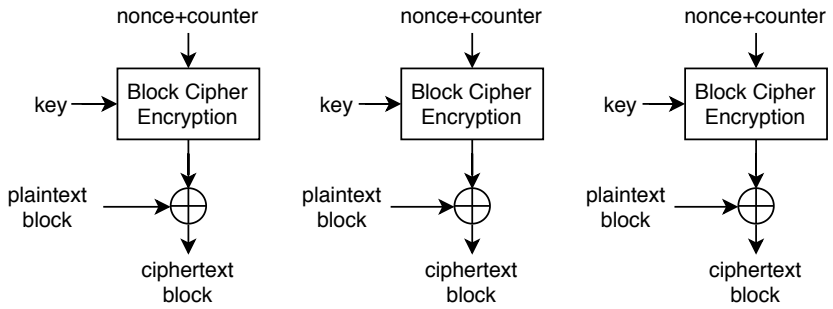
A block cipher specifies the algorithm for encrypting or decrypting a single block of data. However, a message is usually much longer than the length of a block so it is necessary to employ a *mode of operation*, which specifies how the block cipher should be applied to encrypt or decrypt a stream of blocks. Most modes of operation require, in addition to

the key and plaintext, an *initialization vector (IV)* or a *nonce*. The IV ensures that when the same plaintext and key are used together multiple times, the ciphertext is different. A nonce is an arbitrary number that can be used just once and it serves the same purpose as an IV.

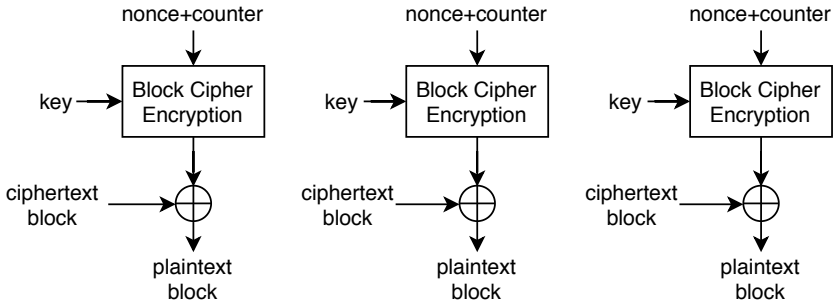
In 2001, NIST published a report defining five modes of operation: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) [25]. The simplest mode of operation is ECB, which encrypts and decrypts each block independently using the block cipher's encryption module, decryption module, and the same key. It is easy to implement but is weak in terms of security because the same plaintext always produces the same ciphertext. As patterns can be easily revealed, ECB becomes vulnerable to replay attacks. CBC overcomes the pitfalls of ECB mode by having a chained dependency, where the ciphertext of one block depends on the ciphertext of a previous block. Although this technique blurs the encrypted output, it presents two disadvantages: blocks cannot be encrypted in parallel and errors are propagated. If one bit of a plaintext is erroneous, all subsequent ciphertext blocks will be incorrect and thus, cannot be decrypted.

The purpose of this section is not to provide a comprehensive overview of block cipher modes, but rather to emphasize a point. In this work, only the encryption function of the block cipher is implemented, which, initially, may not seem practical. However, modes of operation like cipher feedback, output feedback, and counter require only a block cipher's encryption module. Among the three, counter mode is particularly popular. It effectively makes the block cipher act as a stream cipher. Counter mode requires a nonce and a counter. The counter is incremented for every block and combined with the nonce (by an XOR operation or concatenation). The nonce and counter combination is given as the input to the block cipher. The output of the block cipher, serving as the key stream, is XORed with the plaintext to obtain the ciphertext. Encryption and decryption in counter mode require only the block cipher's encryption module, as illustrated in Figure 2.2.

Counter mode surmounts the pitfalls of both ECB and CBC modes: patterns are concealed, parallelization is easily implemented, and errors are not propagated. Counter mode can be easily parallelized because each thread needs only to do independent work; there are no dependencies among blocks. Because of this independence, errors are also not propagated. If one bit is faulty, it only affects that block it belongs in, whereas, in CBC, every block thereafter would be affected. Correction algorithms can be used to restore the original message.



(a) Encryption



(b) Decryption

Figure 2.2: Counter mode

2.2.3 The Advanced Encryption Standard (AES)

AES supports a block size of 128 bits and key sizes of 128, 192, or 256 bits [26]. The key size determines the number of rounds in the algorithm: 10, 12, or 14, respectively. Hereafter, all mentions of AES refer to AES-128, which uses a 128-bit key. Except for the final round, every round of the encryption algorithm consists of the following operations, in the specified order:

1. SubBytes - substitution layer (S-boxes)
2. ShiftRows - permutation of bytes
3. MixColumns - mixing operation
4. AddRoundKey- state is combined with round key using XOR operation

The AES rounds are prefaced by an initial step consisting of only *AddRoundKey*. The

final round involves all the above operations except for *MixColumns*. The AES structure is shown in Figure 2.3.

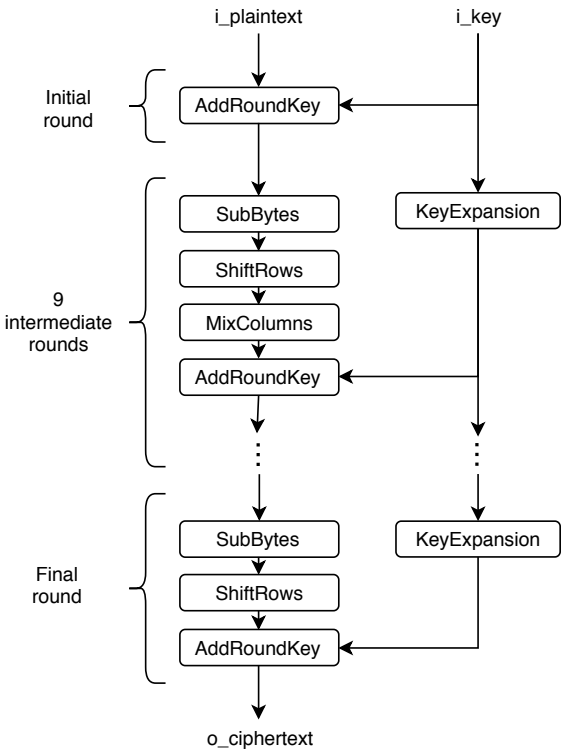


Figure 2.3: AES structure

The 128-bit data block, called the *state*, is often represented as a 4×4 column-major matrix of bytes:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

The following sections describe each of the operations (AddRoundKey, SubBytes, ShiftRows, MixColumns) in more detail.

AddRoundKey

The initial step of the AES encryption algorithm is AddRoundKey, in which the key and message are combined using a simple bitwise XOR operator.

SubBytes

The SubByte operation is the substitution layer of AES. Each S-box operates on a byte. The S-box is the only non-linear operation in the cipher and is a bijective function so that it can be inverted (required for decryption). In Table 2.2, the S-box output for a given input byte can be found by finding the row that corresponds to the most significant nibble, and then finding the column that corresponds to the least significant nibble. For example, the S-box output for input `0x4B` is `0xB3`.

Table 2.2: S-box lookup table in hexadecimal notation

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
A0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
B0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
C0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
D0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
E0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
F0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

The core of the SubBytes operation is actually a finite field inversion (with irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$) followed by an affine transformation. The affine

function is composed of a linear function and a translation:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \mod 2$$

The SubBytes operation can be implemented as lookup table or as a circuit which performs the actual inverse and affine mapping computation.

ShiftRows

The ShiftRows layer performs a left circular shift for each row of the state matrix, except the first. The second row shifts by one byte, the third row by two bytes, and the last row by three bytes.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix} \longrightarrow \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_5 & b_9 & b_{13} & b_1 \\ b_{10} & b_{14} & b_2 & b_6 \\ b_{15} & b_3 & b_7 & b_{11} \end{bmatrix}$$

MixColumns

The ShiftRows and MixColumns operations, together, form the permutation layer of the AES SP-network. In MixColumns, every column of the matrix undergoes a linear transformation. If $a_0a_1a_2a_3$ are four bytes of a column, then the result of a MixColumns operation, $b_0b_1b_2b_3$ is obtained as follows:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The matrix multiplication is a dot product operation, with multiplication and addition being finite field arithmetic operators.

Key Expansion

Each round key is derived from the round key of the previous round (or from the input cipher key, if it is the first round). Suppose the columns from left to right are indexed 0 to 3. First, column 3, $[k_{12}, k_{13}, k_{14}, k_{15}]$, is left rotated by one byte (RotWord): $[k_{13}, k_{14}, k_{15}, k_{12}]$. Then, each byte of the result undergoes the S-box function (SubWord) and the first byte gets XORed with the round constant, $rcon$. The value of $rcon_i$ for a round i is generated by

$$rcon_i = \begin{cases} 1 & \text{if } i = 1 \\ 2 \cdot rcon_{i-1} & \text{if } i > 1 \text{ and } rcon_{i-1} < \text{0x80} \\ (2 \cdot rcon_{i-1}) \oplus \text{0x1B} & \text{if } i > 1 \text{ and } rcon_{i-1} \geq \text{0x80} \end{cases}$$

Table 2.3 lists the $rcon_i$ values for $0 < i < 11$.

Table 2.3: $rcon$ values in hexadecimal format for $0 < i < 11$

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

Column 0 of the new key, $[k'_0, k'_1, k'_2, k'_3]$, is obtained by XORing each byte of column 0 with each byte of the transformed column 3. Columns $j = 1, 2, 3$ of the new key are then obtained by performing the XOR of column $j-1$ of the new key and column j of the current key. The key schedule is illustrated in Figure 2.4.

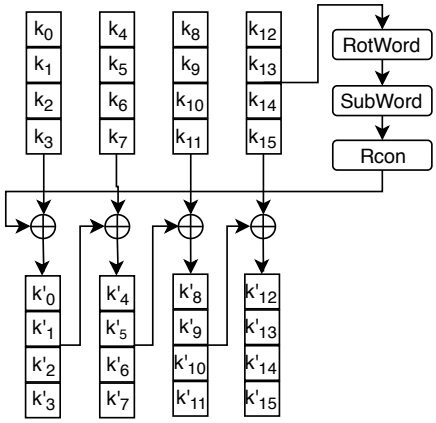


Figure 2.4: AES KeyExpansion architecture

2.2.4 PRESENT

Designed for hardware efficiency, PRESENT has a 64-bit state and supports key sizes of 80 bits and 128 bits [27]. This section describes the PRESENT encryption algorithm for a key size of 80 bits. It has 31 rounds, each consisting of

1. *AddRoundKey*: state is combined with round key
2. *sBoxLayer*: substitution layer
3. *pLayer*: permutation layer

After the 31 rounds, a final AddRoundKey operation is performed. A high-level architecture of PRESENT is depicted in Figure 2.5.

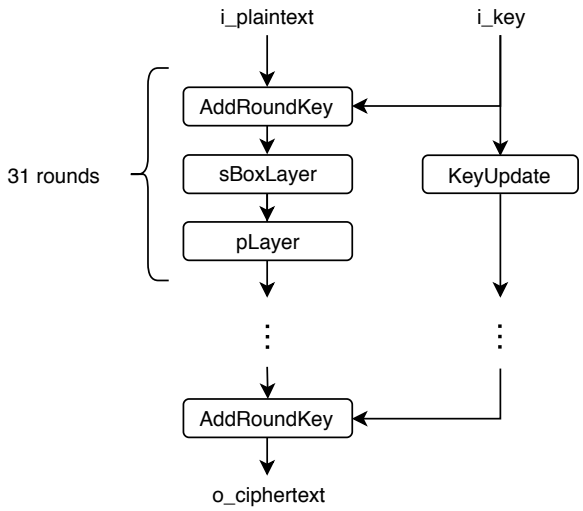


Figure 2.5: PRESENT structure

AddRoundKey

In AddRoundKey, the current state is combined with the 64 most significant bits of the round key using an XOR operation.

sBoxLayer

The S-box in PRESENT operates on a 4-bit input. The mapping from input to output is shown in Table 2.4 [27].

Table 2.4: PRESENT S-box in hexadecimal format

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

pLayer

The pLayer is a permutation layer of the SP-network. The bit i is moved to position $P(i)$, as shown in Table 2.5 [27].

Table 2.5: PRESENT P-layer

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	4	54	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Key Schedule

A round key is generated based on the previous round key and a round counter. The round key k_i is obtained by performing the following steps on the previous round key k_{i-1}:

1. 61-bit rotation to the left
2. Apply PRESENT S-box to 4 left-most bits
3. XOR bits [19..15] with the round constant

The round constant is a binary counter set to 1 at the beginning of the encryption operation and incremented by 1 after every round. The structure of the key schedule is shown in Figure 2.6.

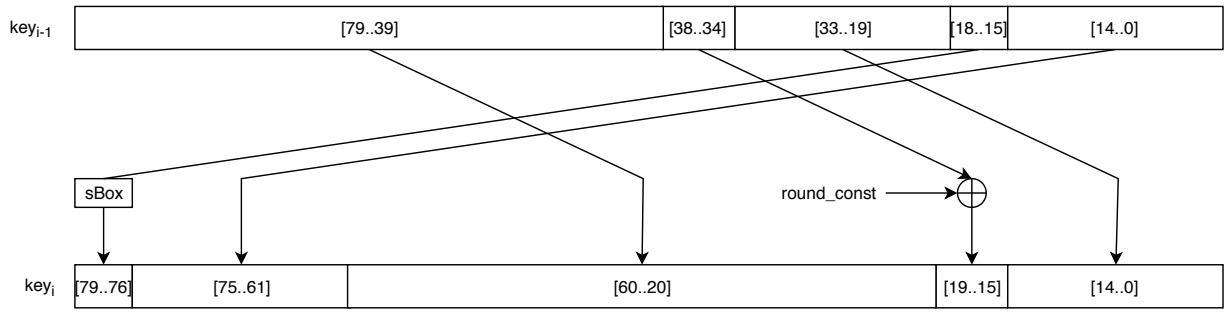


Figure 2.6: PRESENT key schedule

2.3 Related Work

2.3.1 Low-area AES Works

Low-area implementations of AES have appeared since the outset. In the same year that AES became standardized, Satoh *et al.* [28] introduced a compact architecture that combined the encryption and decryption datapaths. They achieved an area of 5400 GE on a 0.11 um process and a latency of 54 clock cycles by using a datawidth of 32 bits. One of the first 8-bit architectures of AES came from Feldhofer *et al.* [6], who implemented both encryption and decryption cores on a 0.35 um process, which occupied 4400 GE altogether. Encryption takes 1032 clock cycles and decryption 1165 clock cycles. The authors used a single S-box for both encryption and key generation. Hamalainen *et al.* [7] also designed an 8-bit encryption architecture, which occupies an area of 3100 GE on a 0.13 um process. Performing on-the-fly key schedule and encryption in parallel, their design achieves a latency of 160 clock cycles. The authors use two instances of the S-box (one for encryption, one for key schedule), which borrows Canright's design [29]. Their ShiftRow operation is performed serially using Jarvinen's Byte Permutation Unit [11], which outputs the bytes of the state in ShiftRows order.

Moradi *et al.* [8] took a holistic approach for minimizing the total area, rather than optimizing each component individually. Their 8-bit encryption module achieves an area of

2400 GE on a 0.18 um process and has a latency of 226 clock cycles. Moradi uses a single S-box that is shared between encryption and key generation and has nearly 100% utilization. Building on the work of Moradi, Banik *et al.* [10] developed Atomic-AES and Atomic-AES v2.0, which perform both encryption and decryption. The former has a 226 clock cycle latency for both encryption and decryption and an area of 2605 GE on STM 90 nm CMOS library. Atomic-AES v2.0 achieves a smaller area (2060 GE) at the cost of greater latency (246/326 for encryption/decryption).

A sub-2000 GE implementation of an AES encryption module appeared in the work of Mathew *et al.* [9]. Occupying 1947 GE on a 22 nm process, the design performs encryption in 336 clock cycles. The 8-bit architecture performs encryption entirely in a new composite field, applying the isomorphic mapping prior to the first round and then the inverse mapping at the end of the final AES round. One S-box is used for both encryption and key generation, which occur in alternation. The authors did an exhaustive test of all possible polynomials for the ground-field and extension-field and chose the ones that provided the smallest area. Their results show that the optimal encrypt and decrypt cores use different polynomials. Sub-atomic AES, designed by Wamser *et al.* [30], is the smallest encryption/decryption dual core to date, reported to be more than a 10% reduction in area compared to Atomic-AES v2.0. However, it comes at a cost of a large increase in latency: 689/1281 clock cycles for encryption/decryption.

2.3.2 Low-energy Works

The works surrounding low-energy implementations of block ciphers are much less abundant than those focused on low area. In 2015, Banik *et al.* [13] studied the energy efficiency of unrolled rounds in a number of lightweight block ciphers and developed a model to predict the optimal number of rounds in a single clock cycle to achieve the lowest energy. Banik *et al.* [14] provided a glitch filtering technique using AND-gates for unrolled rounds, which they termed *Round Gating*. The authors benchmarked the results for 10 lightweight ciphers for number of unrolled rounds ranging from 1 to 4.

Dumpala *et al.* [31] presented a different technique for glitch filtering, which uses latches instead of AND gates. They showed their results for SIMON-128 and AES-256 on FPGA and assessed the tradeoff between the energy reduction and the area increase when using their glitch filter. Dhanuskodi [15] also explores a latch-based glitch filter for unrolled designs, called *Combinational Checkpointing*, but on ASICs. The authors explained that Round Gating causes unnecessary switching activity when the enable signals must be

reset low at the end of each clock cycle. Since Combinational Checkpointing does not require such resetting, it is more energy efficient than Round Gating. They compared both Combinational Checkpointing and Round Gating against a baseline (no glitch filters) using SIMON and AES. The authors also explored the optimal spacing between glitch filters for their technique (*i.e.* how many rounds between each glitch filter).

Chapter 3

Digital Hardware Design and Tradeoffs

As with all engineering designs, there are tradeoffs that need be considered with the AES and PRESENT designs. Section 3.1 discusses the differences between FPGA and ASIC. Section 3.2 introduces the metrics of throughput, area, and energy, each of which is explained more thoroughly in Section 3.2.1, Section 3.2.2, and Section 3.2.3, respectively.

3.1 FPGA vs ASIC

Hardware designs can be implemented in field programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC). ASICs are customized for a specific purpose and perform only one function. FPGAs are made of thousands or even millions of programmable logic blocks and interconnects. Because FPGAs are programmable, they are suitable for prototyping a design and are often used for this purpose. In terms of cost, ASICS are cheaper for large production volumes whereas FPGAs may be more cost effective for smaller production volumes. The non-recurring engineering cost of an ASIC is higher. If there are errors in an FPGA design, it can simply be reprogrammed. If there are errors in an ASIC, it must be refabricated, increasing the cost. Generally, ASICs are faster, more energy efficient, and take up smaller area.

The design flow for achieving an ASIC implementation is shown in Figure 3.1. First, the product specifications are laid out. In this stage, the functionalities, features, architecture,

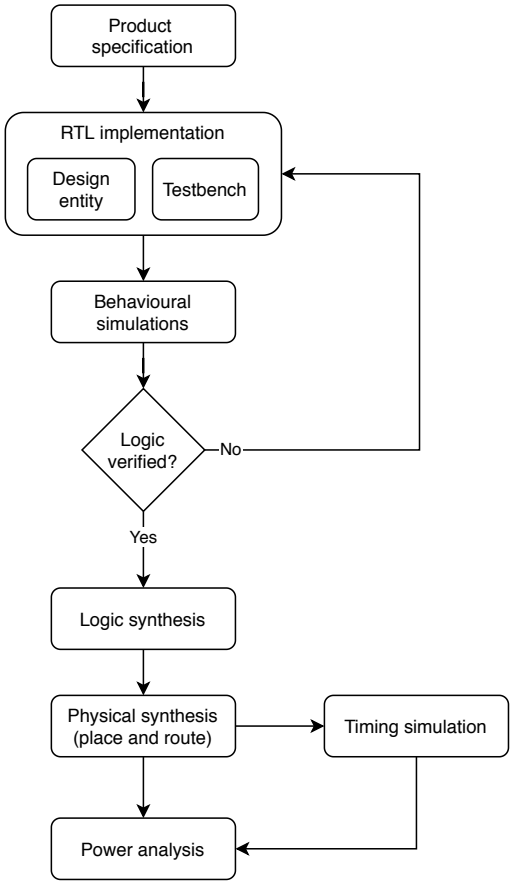


Figure 3.1: ASIC design flow

design goals (*i.e.* area, power, throughput) are defined. For example, for an AES design, the functionalities can include encryption or decryption or both, and the architecture can be 8-bit datawidth if low area is a design goal. The next step is register transfer level (RTL) implementation. The RTL code is constructed using a hardware description language (HDL), such as VHDL or Verilog. These languages describe the behaviour of the circuits and are also used to develop a testbench, which provides stimulus to simulate and verify the logic of the design. The design being tested in the testbench is referred to as the unit under test (UUT) or device under test (DUT). Simulations are performed using simulation tools such as ModelSim, which display waveforms of the signals and aid with debugging. The RTL design and functional verification can often take several iterations, which is common and normal.

Once the logic of the DUT is correct, the next step of the design flow, logic synthesis, can be performed. Using a logic synthesis tool, such as Design Compiler by Synopsys, the RTL code is translated into a gate-level netlist. The translation is a two-step process. The first step is a translation into generic logic gates that do not have a physical implementation. For the design to be physically achievable, the gates need a transistor-level implementation. A collection of such gates, called a technology library or an ASIC cell library, is developed and distributed by a foundry. The second step involves converting the generic logic gates into gates from an ASIC cell library. The library usually contains multiple implementations of the same logic function, differing in area and speed to allow for implementation tradeoffs.

The result of logic synthesis is a list of technology-dependent gates, which do not have a determined location on silicon and so the design still cannot be physically realized yet. The purpose of physical synthesis is to transform the netlist into a layout, in which every gate has a location and wired connection to other gates. The steps in physical synthesis include floor planning, placement, clock tree synthesis, and routing. Floorplan includes defining the width and height of the die, finding placement for pins, while making sure constraints are satisfied. Placement involves finding specific locations for the gates. Clock tree synthesis builds the clock tree and assures that timing requirements are satisfied. Routing determines how the wires are routed so that the gates are connected. During the routing process, the delays of the nets are also calculated.

After physical synthesis generates a layout, timing simulation needs to be performed to verify it. The same testbench used in functional simulation can be used for timing simulation. Unlike in functional simulation, wherein the waveforms are ideal, the waveforms in timing simulation actually reflect the timing delays of a signal through a wire.

Power analysis is optional and is performed using a tool like Cadence Encounter. Encounter evaluates the power consumption of a design using the layout information and a value change dump (VCD) file that was generated during timing simulation. The VCD file contains information about the transitions of signal.

After physical synthesis passes verification, the design is ready to be signed off and sent to a fabrication facility.

3.2 Design Tradeoffs

There are several considerations for tradeoffs, but the three that are of interest in this work are throughput, area, and energy. As a general definition, throughput refers to

how much work can be accomplished in a given amount of time. In the context of a block cipher, throughput can be defined as the number of bits of ciphertext that are generated per clock cycle. A high throughput is usually desirable, as it is a measure of performance. Area (of the silicon) is a metric that is measured in physical units (square microns) or gate equivalents (GE). Because advances in technology continue to create smaller nanometer processes, comparing designs by the physical area is not entirely fair; two identical designs will result in different areas if one is synthesized using a smaller technology process. Thus, the notion of GE was introduced to normalize the size of a circuit across different manufacturing technologies: one GE is equivalent to the silicon area of a 2-input NAND gate. Like area, the energy consumed per unit operation is a metric which should be minimized. Energy is the electrical effort expended during the execution of an operation. Although power, the amount of energy consumed per unit time, correlates to energy, optimizing for the former can be significantly different from optimizing for the latter, as we'll see in the following sections. In an ideal design, all aspects of high throughput, low area, and low energy consumption should be present, but this is very hard to achieve in practice; often, one metric is sacrificed in order to gain in another. In fact, optimizations for different metrics can have opposing effects. For instance, designing for high throughput often inevitably results in higher area, which contradicts a low area goal.

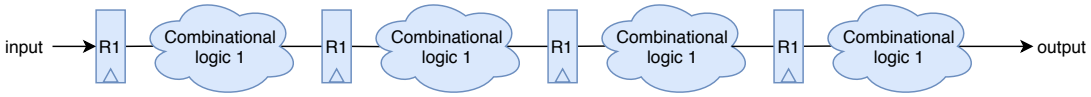
3.2.1 Throughput

The throughput of a block cipher refers to the number of ciphertext bits that are produced per clock cycle. A block cipher design that implements one round per clock cycle, achieves a throughput of $\frac{m}{R}$, where m is the length of the block in bits and R is the number of rounds in the algorithm. There are two ways to increase the throughput: pipelining and loop unrolling.

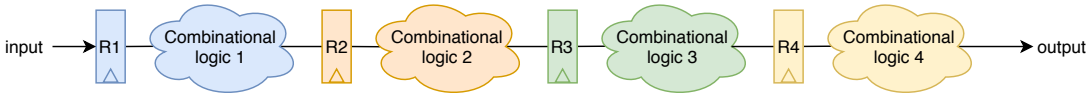
Pipelining is a technique for parallelizing operations, achieved by splitting the entire operation into stages and then overlapping the execution of the stages. It is analogous to an assembly line, in which every person is responsible for one task and the parts move from one workstation to another until it is complete. Pipelining improves throughput without significantly affecting latency, which is the length of time between the start and completion of an operation. In other words, pipelining allows a ciphertext to be produced at every clock cycle, but the latency of encryption will remain at the same number of clock cycles required in the sequential (non-pipelined) design.

A consequence of pipelining is an increase in hardware resources. In a sequential design,

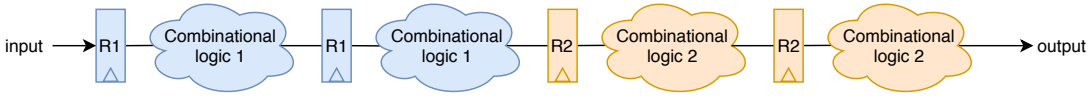
the hardware can be reused in each clock cycle. In a pipelined design, every stage must have its own dedicated resources since it will run simultaneously with all other stages. Therefore, increasing throughput comes at a cost of increased area. When one desires a more balanced design, opting to sacrifice throughput in order to reduce area, a partially pipelined design can be implemented. Whereas a fully pipelined design allows a ciphertext to be generated at every clock cycle, a partially pipelined one generates output at a lower frequency (*e.g.* every 4 clock cycles). The number of clock cycles per stage determines the degree to which a design is pipelined and it is an important parameter that will be referred to as *ccps* (clock cycles per stage) in upcoming sections. Throughput can be calculated as $\frac{m}{ccps}$. The difference between a sequential design, a fully piplined, and a partially pipelined one is illustrated in Figure 3.2.



(a) Sequential (unpipelined): one stage



(b) Fully pipelined: four stages



(c) Partially pipelined: two stages

Blocks of the same colour indicate the same stage. Each stage requires its own set of hardware resources. Within a stage, resources can be reused in different clock cycles.

Figure 3.2: Sequential (unpipelined) vs pipelined

The second method of increasing throughput is loop unrolling. This technique removes registers between some block cipher rounds, resulting in several back-to-back (unrolled) rounds in combinational logic, as Figure 3.3 depicts. In this thesis, the degree of unrolling is denoted by a parameter called *rpcc* (rounds per clock cycle). Loop unrolling increases

throughput and reduces latency. Futhermore, it increases the critical path (within a clock cycle) and subsequently, the minimum clock period, but reduces the total number of clock cycles needed to encrypt a block of plaintext. The elimination of some registers causes the datapath of the whole encryption operation to be shortened, hence reducing latency. The throughput can be calculated as $\frac{m}{\lceil R/rpcc \rceil}$, where m is the length of block in bits, and R is the total number of rounds (of the cipher algorithm). Similar to pipelining, loop unrolling increases area; each combinational round requires its own set of hardware.

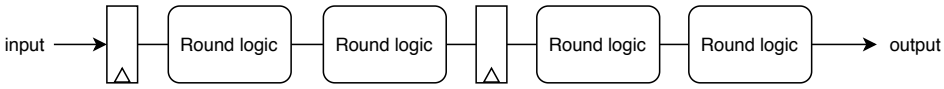
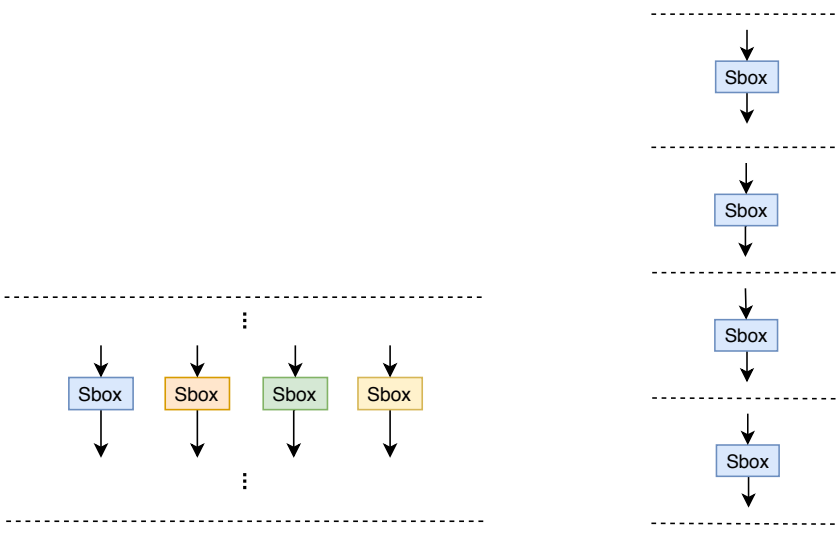


Figure 3.3: Loop unrolling with $rpcc = 2$

3.2.2 Area

In some applications, area has become less of a concern due to the decreasing size of transistors and increasing size of die. However, in other applications, such as portable devices, low area is still a design goal. Methods of optimizing for low area include serializing some operations (*i.e.* performing them sequentially rather than in parallel) and employing different field representations which allow for more efficient hardware. The cost of reducing area is usually an increase in latency. In SP-networks, the component that takes up the most area is typically the substitution layer. Instead of having the S-box function perfomed in parallel (which requires many S-box instances), one might design the cipher to have it executed serially (which requires just one instance of the S-box). In effect, this will increase the total number of clock cycles required to complete the encryption operation, thus increasing the latency of the operation. Both methods are illustrated in Figure 3.4, in which the dotted lines represent a clock cycle boundary. In Figure 3.4(a), four different instances of the S-box (indicated by the different colours) are executing in parallel and the operation finishes in one clock cycle. In Figure 3.4(b), the same instance of the S-box is used in each clock cycle (as indicated by the sole blue colour) and the operation requires four clock cycles.

For ciphers whose designs are based on finite field representations, area optimizations can be made by using a different representation of the field that offers a more efficient hardware implementation. The AES cipher algorithm is based on the finite field $GF(2^8)$ with irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. Certain composite field representations, such as $GF((2^4)^2)$, can produce the same functionality using a different hardware structure



(a) Parallel Sboxes (b) Serial Sboxes

Figure 3.4: Parallel vs serial substitution layer

that is lower in area. It is important to note that not all composite field representations lead to lower area; the hardware efficiency depends on the reduction polynomials, which need to be selected with care. Composite fields are further discussed in Section 4.1.

3.2.3 Energy and Power

Due to the lively interest and development of IoT devices, low power and low energy have become important optimization goals for ASIC designs. Although both goals sound similar, there are important differences between them that should be distinguished. Energy is the total electric work done by the system. Power is the rate of energy consumption. Low energy consumption is often the objective for battery-powered devices, to ensure that the battery lasts as long as possible. For passively powered devices (*e.g.* passive RFID tags), the main concern is low power. In this case, the passive device relies entirely on, say, an RFID reader for its power source. It uses energy in a signal sent from the RFID reader to power its circuit and to generate a signal back to the RF system, a process known as backscatter. Low power, rather than low energy, is the concern because it is the energy consumed per unit time that matters. Typically, lower area designs lead to lower power consumption, but not necessarily lower energy because the latency increases.

Power dissipation has two components: static power dissipation and dynamic power dissipation. Static power constitutes leakage power. Leakage power is dissipated when there is no switching activity. It is due to the current lost in p-n junctions in CMOS transistors. Dynamic power includes switching power and short-circuit power. Switching power is dissipated when signals transition and it is dependent on the clock frequency and the average switching activity. Short-circuit power dissipation is the power lost due to a temporary direct connection between the supply voltage and ground when transistors switch states. Short-circuit power is usually negligible.

While energy values cannot be directly obtained from electronic design automation (EDA) tools, they can be derived from the power values that EDA tools are able to generate. The energy per bit of a plaintext can be derived as:

$$\frac{energy}{bit} = \frac{power \times clk\_period}{throughput} \tag{3.1}$$

There are several ways to reduce energy. Clock gating is a technique that reduces the dynamic power dissipated by the clock signal by disabling it to certain registers when there is no data. This reduces unnecessary switching activity, which reduces power as well as energy. Clock gating can be added by synthesis tools and do not require changes to the RTL code. It not only reduces power consumption of the clock signal but also removes the need for chip-enabled registers, which occupy more area. The difference between a register with chip-enable and one with clock gating is illustrated in Figure 3.5. Note that Figure 3.5(a) is conceptual and does not represent the actual structure of a chip-enabled register.

(a) Register with chip-enable

(b) Register with clock gating

Figure 3.5: Clock-enabled register vs clock-gated register

Power gating is a low-power technique that reduces leakage power by blocking the current to parts of the circuit that are not in use. Unlike clock gating, which can be

conducted transparently from a design, power gating usually involves modifications to the RTL code.

Loop unrolling, which was discussed in Section 3.2.1 as a method to improve throughput, can also be applied for a low-energy objective. Loop unrolling can reduce energy because it effectively removes registers from the datapath of the cipher, which in turn decreases the energy consumption. However, the energy due to glitches increases as the degree of unrolling increases. A glitch is a superfluous signal transition that does not provide any functionality. Glitch filtering methods can be employed to reduce the amount of glitches and thus, energy. More details about glitch filters appear in Chapter 5.

Chapter 4

Area Optimization in AES

Several kinds of architectures and mathematical structures have arisen in an effort to achieve low area in AES. Recently, these designs and lightweight cryptography, in general, have received a lot of attention because IoT devices demand more resource-efficient modules. Section 4.1 discusses the finite field upon which AES is built and explains how different representations yield different area efficiencies. The methodology is presented in Section 4.2. Section 4.3 presents an analysis and benchmark of existing low-area AES designs, showing how different component implementations lend themselves better to different kinds of architectures. Using this analysis, a novel AES architecture, presented in Section 4.4, was created.

4.1 Composite Field

When an extension field $GF(2^k)$ has k that is not prime (*i.e.* $k = n \cdot m$ for some integers $n, m \neq 1, k$), it can be decomposed into a composite field, $GF((2^n)^m)$, which is a tower of extension fields. $GF(2^n)$ defines the extension field over $GF(2)$ (*i.e.* the elements of $GF(2^n)$ have coefficients in $GF(2)$). Similarly, $GF((2^n)^m)$ defines the extension field over $GF(2^n)$ (*i.e.* the elements of $GF((2^n)^m)$ have coefficients in $GF(2^n)$).

Exploring composite field representations is interesting because they can sometimes achieve more efficient hardware implementations than the standard binary extension field. AES is based on mathematical structures in the field GF(2^8) with irreducible polynomial

$P(x) = x^8 + x^4 + x^3 + x + 1$. The S-box, comprising of the inverse operation plus an affine transformation, is typically the most costly component (in terms of area). Several works have shown that representing the finite field $GF(2^8)$ as a composite field (such as $GF((2^4)^2)$ or $GF(((2^2)^2)^2)$) can yield a more efficient S-box [29], [32], [33], [34]. Such a task entails two problems: 1. selecting the defining polynomials of the composite field to construct the field, and 2. finding an isomorphic mapping between the binary field and composite field, once the defining polynomials are fixed.

Methods for selecting the defining polynomials of the composite field have been explored before [32], [9]. Finding an isomorphic mapping between a binary field and the composite field is well documented by Paar [35]. The result of the process is a transformation matrix, M, which maps an element, E, in the original field $GF(2^8)$ to an element, E', in the composite field through $E' = ME$. For example, the transformation matrix and its inverse used in the following sections of this chapter (Mathew's mapping) are

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

where the least significant bit is in the upper left corner. The algorithm proposed by Paar applies when the defining polynomials are primitive. Zhang proposed an efficient algorithm for finding isomorphic mappings when the defining polynomials, such as the AES polynomial, are not primitive [34]. The procedure for generating the transformation matrix above is listed in Appendix A.

4.2 Design Flow

Logic synthesis was performed with Synopsys Design Compiler version P-2019.03 using the *compile_ultra* command and clock gating. Physical synthesis (place and route) and power analysis were done with Cadence Encounter v14.13 using a density of 95%. We used Mentor Graphics ModelSim SE v10.5c for simulations. Area results are post

place-and-route and power analysis is based on timing simulation. We used the following ASIC cell libraries: ST Microelectronics 65 nm CORE65LPLVT at 1.25V, TSMC 65 nm at tpfn65gpgv2od3_200c and tcbn65gplus_200a at 1.0 V, ST Microelectronics 90 nm CORE90GPLVT and CORX90GPLVT at 1.0 V, and IBM 130 nm CMRF8SF LPVT with SAGE-X v2.0 standard cells. Some past works have used scan-cell flip-flops to reduce area because these cells include a 2:1 multiplexer in the flip-flop, which incurs less area than using a separate multiplexer. We chose not to use scan-cell flip-flops because their use as part of the design would prevent their insertion for fault-detection and hence, prevent the circuit from being tested for manufacturing faults.

4.3 Benchmarking and Analyzing Existing Architectures

The architectural and component options for an 8-bit design are ample. For example, the bytes can be internally processed in a row-major or column-major fashion. AES operations (AddRoundKey, ShiftRows *etc.*) can occur in parallel, requiring duplicate hardware, or serially, which could offer better area at the cost of increased latency. AES operations can also take on a different order as long as dependencies are maintained (*e.g.* the relative order of ShiftRows and SubBytes does not matter).

There are three works focused on low-area AES encryption cores which stand out for their area and time efficiency. Mathew's work reports the smallest encryption-only area[9], Moradi's work achieves nearly the lowest possible latency for a single S-box design [8], and Hamalainen's work implements a serialized method of the ShiftRows operation [7]. To better compare these designs, we implemented and synthesized them on a common technology: STMicro 65 nm. In the analysis of these works in Section 4.3.1, the different architectural and component options are introduced, and in Section 4.3.2, they are examined in greater detail.

4.3.1 Architecture Comparison

It is important to note that the following analysis is on the architectures of Mathew, Moradi, and Hamalainen's works, thus our implementations follow their descriptions and circuits as closely as possible, with the exception of common components which remain

uniform across all designs. For instance, the same S-box (Mathew's) is employed in all three architectures. Since Moradi and Hamalainen's design are done in the original AES field, the isomorphic mappings are placed immediately before and after the S-box in our implementation of their circuits. To allow rapid prototyping of many different architectures, a binary encoding is used for the counters. Once an architecture is finalized, the implementation can be optimized by replacing the binary counter with a linear feedback shift register (LFSR), such as we've done for the optimized version of our design. See Appendix B for a description of LFSRs.

Mathew's Architecture

To our knowledge, the smallest prior reported encryption-only module of AES is from the work of Mathew, who reported an area of 1947 GE on a 22 nm process. Mathew applies their custom composite field to the entire algorithm and performs almost all AES operations in the new basis. The design uses one instance of the S-box and has a total latency of 336 clock cycles. In each round, 16 clock cycles are dedicated to encryption and 16 clock cycles to key expansion. During encryption, AddRoundKey, SubBytes, and MixColumns are performed serially, and the results are stored in 128 *intermediate* registers. The ShiftRows operation is performed in the first clock cycle of key expansion mode when the contents of the intermediate register are transferred to the state register. Because AddRoundKey, SubBytes, and MixColumns are applied in every clock cycle and MixColumns depends on ShiftRows, the latter operation is moved to the beginning of the round. This has two consequences: the input bytes must be loaded in ShiftRows order and the key must be added to the state in a ShiftRows order by introducing a 4:1 multiplexer (mux). The authors also use scan flip-flops in their design.

Analysis of Mathew's Architecture

Our implementation of Mathew's design achieves an area of 2640 GE. The large discrepancy between our and Mathew's area could be due to two reasons. First, we opted not to use scan flip-flops. Since scan flip flops incur less area than the combined area of a mux and a flip-flop, our implementation will be larger. Second, while the composite field selected by Mathew may be optimal on their technology, it might not be on ours. See Appendix C for our calculations of the S-box and MixColumns operations in Mathew's composite field.

The main drawbacks of Mathew's design are the additional intermediate registers, the need to reorder inputs, and the 336 clock cycle latency. Most of the intermediate registers can be eliminated simply by writing the result of MixColumns directly into the data registers instead of into the intermediate registers. This eliminates 12 of the 16 8-bit registers (4 are still needed for temporary storage of the result) and introduces four 8-bit multiplexers. This version achieves 2240 GE in our experiments. Another option for MixColumns is to have a 32-bit implementation that operates on a column at a time and to do it in 4 clock cycles after the ShiftRows operation. The advantage of this is the elimination of all intermediate registers and no longer needing to reorder inputs. The area result for this implementation is 2270 GE.

While Mathew decided it would be better to do the entire algorithm in the composite field, Canright argues that this may be less efficient than having just the S-box in a composite field because the simplicity of the constants in the MixColumns operation in the original basis, 0x03 and 0x02, would be lost in the composite field [29]. As an experiment, we implemented a version that performs the isomorphic mapping immediately before and after the S-box, leaving all other operations in the original basis. The results showed a decrease in area from 2640 GE to 2580 GE.

Moradi's Architecture

Moradi's 8-bit encryption architecture is next smallest at a reported 2400 GE and stood out for its low latency for a single S-box design. Sixteen clock cycles are dedicated to AddRoundkey and SubBytes, one to ShiftRows, and four to the simultaneous execution of MixColumns and SubWord (of the key expansion), for a total round latency of 21 clock cycles. The S-box borrows Canright's design and MixColumns is performed on a column at a time, requiring no additional registers. Moradi's design also uses scan flip-flops and requires both inputs and outputs to be reordered, as bytes are processed in a row-major order.

Analysis of Moradi's Architecture

Our implementation of the design occupies 2370 GE. We also point out that our implementation uses a larger S-box (the same one we used in our implementation of Mathew's design) and a regular binary counter (instead of an LFSR).

A slight improvement on latency can be made by doing the ShiftRows operation in the same clock cycle as the final AddRoundKey byte. This eliminates one clock cycle in each round, resulting in a total cycle count of 216 compared to the original 226. Although this causes the location of the muxes to change, the number of muxes remains the same (every register in the second and third rows require a mux and the left-most register of the bottom row requires a mux). In our experiments, making this change caused area to decrease by 20 GE, which could be attributed to how the tools optimize the design.

Other improvements have been suggested by Banik *et al.*[10], who extended Moradi's work to support both encryption and decryption. The registers in the two middle columns of the key do not need to shift during clock cycles 17-20; thus, they do not need to be scan flip-flops or, in our case, require muxes. In Atomic-AES v2.0, ShiftRows is performed over 3 clock cycles, during which the rows are selectively shifted by one byte at a time. This eliminates all muxes except the ones for the rightmost column. More details appear in Section 4.3.2.

Moradi's paper states that using a row-major design reduces area by 13.5% and that if column-wise ordering is needed, 20 additional 8-bit wide 2-to-1 multiplexers are required. However, in our experiments, the column-major design in fact showed a decrease in area: 2280 GE. According to our analysis, a column-major design should decrease area because although more muxes are required for the ShiftRows operation, a greater number of muxes are saved in the key register. In a row-major design, 18 muxes (9 for ShiftRows and 9 for key register) are required. In a column-major design, 15 muxes are needed (12 for ShiftRows and 3 in key). See Section 4.3.2 for details. Moradi argues that the extra registers and control logic associated with a serial version of MixColumns would exceed the area of a combinational 32-bit MixColumns. Our analysis is presented in Section 4.3.2.

Hamalainen's Architecture

At 3100 GE, Hamalainen's architecture uses two instances of the S-box, one for encryption and the other for the key schedule. The authors did the ShiftRows operation serially by employing the Byte Permutation Unit (BPU) developed by Jarvinen *et al.*[11]. Consisting of 12 8-bit state registers and some muxes, the BPU outputs the bytes in the correct order by reading from one of four registers (depending on the clock cycle) and systematically reordering the bytes as they shift through the state registers. Most of the BPU registers simply maintain their normal shifting operation as a shift register and do not require muxes to preface them. MixColumns is also performed serially, similar to the way Mathew did,

using four additional 8-bit registers to store the accumulating result. The SubBytes operation is performed between ShiftRows and MixColumns operations, which differs slightly from the AES specification, but this rearrangement has no effect on functional correctness.

Analysis of Hamalainen's Architecture

Our implementation of the design achieves 2260 GE. Using the BPU has two benefits. First, the area overhead is small. While in the straightforward implementation, the ShiftRows operation requires twelve 8-bit 2-to-1 muxes, the BPU requires only three 8-bit 2-to-1 muxes and one 8-bit 4-to-1 mux (equivalent to three 2-to-1 muxes). Second, now that every operation is serialized, every clock cycle is uniform, making the control logic very simple and reducing the number of muxes. As a result, the round latency can be kept to 16 clock cycles, which is the minimum latency of an 8-bit design, since there are 16 bytes of state.

Summary

Table 4.1 shows a summary of the discussed designs. Latency is measured from the clock cycle the first byte goes in until the clock cycle the last byte becomes ready (*i.e.* it includes cycles for loading and unloading the key and plaintext). To allow rapic prototyping of many designs, we decided not to implement simultaneous loading and unloading in our designs. While Moradi and Hamalainen report both the latency including loading/unloading and the effective latency when load/unload is performed simultaneously, Mathew reports only one latency value of 336, which appears to be the effective latency. To keep the comparisons consistent, we add 16 clock cycles to Mathew's reported latency to account for loading.

4.3.2 Taxonomy of Design Choices

In this section, we compare different ways of implementing MixColumns, ShiftRows, and Key Expansion. Since SubBytes and AddRoundKey are byte-wise operations, they lend themselves well to serial implementations and do not have many architectural options.

Table 4.1: Benchmarking on ST 65 nm process

Design	**Notes**	**Area** (GE)	**Latency** (clk cycles)	**Power** (mW)	**Energy** (nJ/bit)
Mathew [9]		2640	352	0.109	29.9
Mathew variation 1	most intermediate registers removed	2240	352	0.122	33.6
Mathew variation 2	no external reorder of inputs	2270	352	0.090	24.6
Mathew variation 3	original AES basis	2580	352	0.110	30.1
Moradi [8]		2370	226	0.101	17.9
Moradi variation 1	latency reduced	2350	216	0.100	16.9
Moradi variation 2	column-major	2280	226	0.097	17.1
Hamalainen [7]		2260	176	0.145	20.0

Mix Columns

There are three methods of implementing MixColumns: 1. non-serialized, 2. serial with four 8-bit registers, and 3. lightweight serial with two 8-bit registers. The non-serialized method is simply the straightforward method of implementing MixColumns for a single column. It can be done combinationally without the need for any registers. Hamalainen and Mathew both opted for a serialized version, which requires four 8-bit registers to store the temporary result. A third method appeared in the work of Wamser [30], who implemented a serialized version of MixColumns requiring only two additional registers, based on the work of Ahmed *et al.*[12]. Ahmed shows that, if $[s_0, s_1, s_2, s_3]$ is a column, then the MixColumns result, $[s'_0, s'_1, s'_2, s'_3]$ can be calculated as

$$\begin{aligned} tmp &= s_0 \oplus s_1 \oplus s_2 \oplus s_3 \\ s'_0 &= s_0 \oplus tmp \oplus [2 \times (s_0 \oplus s_1)] \\ s'_1 &= s_1 \oplus tmp \oplus [2 \times (s_1 \oplus s_2)] \\ s'_2 &= s_2 \oplus tmp \oplus [2 \times (s_2 \oplus s_3)] \\ s'_3 &= s_3 \oplus tmp \oplus [2 \times (s_3 \oplus s_0)] \end{aligned}$$

In this method, only two values, tmp and s_0 need to be stored in registers. Each MixColumns byte is calculated and ready in one clock cycle, rather than accumulated over four clock cycles. We also test the effect of implementing the methods in Mathew's

composite field. Table 4.2 lists the required number of registers and area for each method. The lightweight serial method in the original basis achieves the lowest area and the non-serial method in Mathew's composite field gives the highest area. For a given method, the original basis always produces lower area than Mathew's basis. In the original basis, the non-serial method obtains lower area than the serial method, but in the composite field, the non-serial method obtains greater area than the serial method. This indicates that the best method depends on the area trade-off between the combinational circuitry, which is determined by the basis, and the registers. Since Moradi stays in the original basis, the non-serial method is the better option. In Mathew's basis, the combinational circuitry of the non-serial method appears to be larger than the area of the four 8-bit registers, thus the serial method is the better option.

Table 4.2: MixColumns implementations

Method	**Num Reg** (8-bit)	**Area** (GE)	
		Original basis	**Mathew's basis**
Non-serial	0	226	366
Serial	4	267	303
Lightweight serial	2	202	234

ShiftRows

We saw three ways of implementing ShiftRows: 1. straightforward method in one clock cycle, 2. over 3 clock cycles, and 3. serially using Jarvinen's BPU. Since the ShiftRows operation is a row-wise operation, a row-major design can offer efficiencies which a column-major design cannot. In a column-major implementation, the single clock cycle method requires 12 muxes (every register in the last 3 rows of the matrix needs one), shown in Figure 4.1(a). However, in a row-major implementation, the ShiftRows operation for the second row of the matrix (*i.e.* left shift by one byte) happens to be the same operation as the shift register. Thus, the ShiftRows operation shown in Figure 4.1(b) requires one mux for every register in the third and fourth rows and only one mux in the second row, for a total of nine muxes. Even more registers can be saved if ShiftRows is done over three clock cycles. In each clock cycle, the second, third, and fourth rows are selectively shifted and they shift by only one byte at a time. Figure 4.1(d) demonstrates that this method requires only three 8-bit muxes. The drawback is the increase in latency. In a column-major design,

performing ShiftRows over three clock cycles doesn't offer any benefits because the column-major shifting doesn't coincide with the row-wise shifting of ShiftRows shifting, shown in Figure 4.1(c). Thus, it still needs 12 muxes. The final method of ShiftRows, the BPU, is designed for a column-major ordering. It requires three 2:1 muxes and one 4:1 mux. If we count the 4:1 mux as three 2:1 muxes, then the total mux count is six for BPU, depicted in Figure 4.1(e). Table 4.3 lists the different methods and their corresponding mux count.

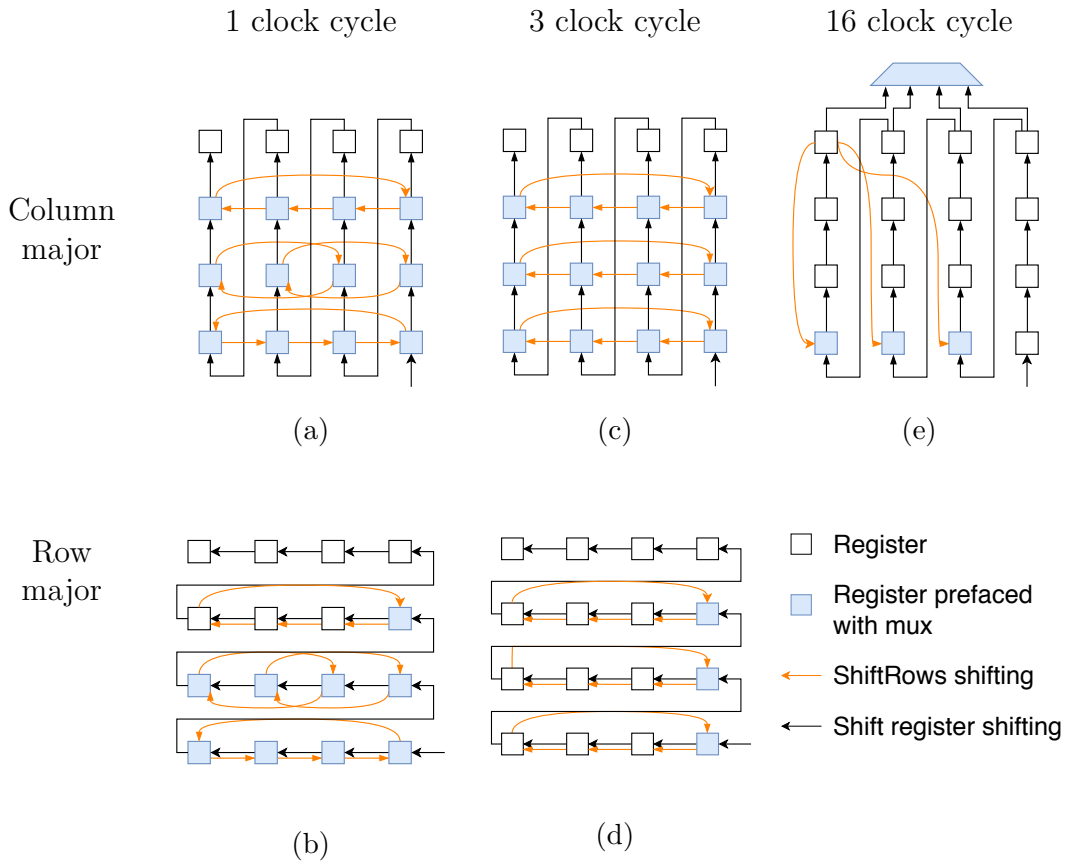


Figure 4.1: ShiftRow implementations

Key Expansion

The movement of the key schedule should be compatible with the movement of the state register. That is, if the state register shifts in a row-major order, then the key register should shift in row-major order as well, so that the AddRoundKey operation can be done

Table 4.3: ShiftRows Implementations

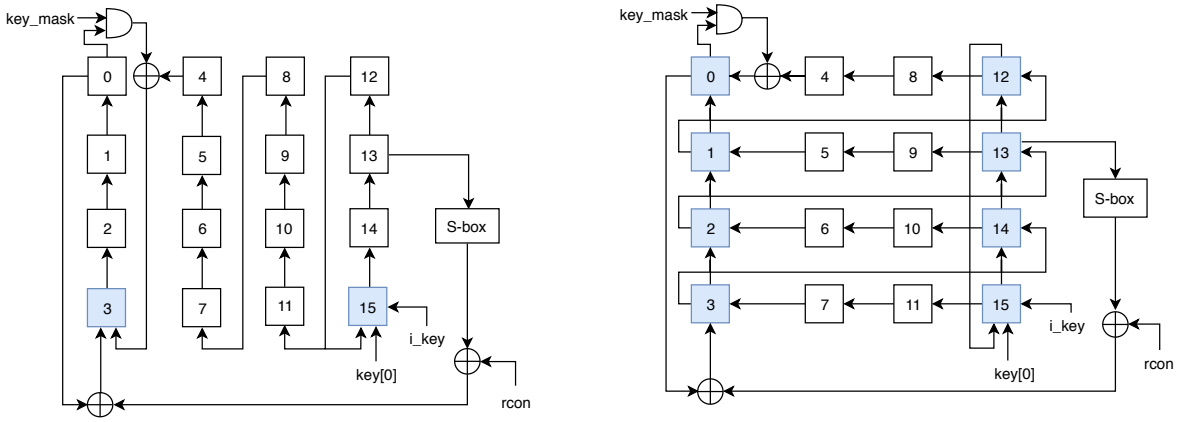
Method	**Number of 2:1 muxes**	
	Column-major	**Row-major**
One clock cycle	12	9
Three clock cycle	12	3
BPU	6	

efficiently. If the key and state registers differ in order, then additional circuitry is needed to make them compatible, which adds area. For instance, in Mathew's circuit, the state register is in ShiftRows order when it is time to do AddRoundKey. Thus, they must use a 4:1 mux to select the correct byte from the key register to add to the state.

While a row-major design can offer area efficiency in the state register (as we saw in the ShiftRows discussion), it complicates the circuitry surrounding the key register. We will show how by using Moradi's key schedule architecture and a corresponding column-major version of it. The key algorithm naturally lends itself better to a column-major design. The left-most column of the next round key is derived from a transformed version of the right-most column. As we populate the left column, we can either rotate the right-most column and keep reading from register 13 or keep the column static and use a 4:1 mux. In a column-major design, keeping the column rotating is a better choice because it follows the normal column-major shifting operation. Muxes need to preface the bottom registers of the rightmost and leftmost columns, shown in blue in Figure 4.2(a). In a row-major design, the column-wise shifting that is required to generate the first column forces a mux to be used on each register in the left-most and right-most columns, shown in Figure 4.2(b).

4.3.3 Summary

Different design goals will motivate different design choices, and the analysis above can facilitate the decision process. For instance, if reordering of inputs/outputs is acceptable, then a row-major design could be the best choice. Given a row-major design, the ShiftRows method that offers the lowest area would be doing it over three clock cycles. On the other hand, if a column-major design is favoured (so that inputs and outputs do not require reordering), then the ShiftRows could be implemented as the BPU, the lowest-area method in a column-major design. Using this analysis, we developed our own design.



(a) Column-major design requires 3 muxes

(b) Row-major design requires 9 muxes

Figure 4.2: Key schedule implemented in column-major and row-major style. Blue registers require muxes. The number of muxes needed for a register equals the number of arrows going into the register minus one.

4.4 Quark-AES

We extract the best features of all the aforementioned designs to realize a novel design. We wanted a single instance of the S-box, the minimum latency for a one-S-box design (20 clock cycles), the lightweight MixColumns implementation, the byte permutation unit, and a column-major design so that inputs and outputs do not have to be reordered externally. The challenge is assembling the desired components into a low-area architecture while adding as few registers and muxes as possible. Although every AES component we chose is serial, we cannot simply stack all of them together in a combinational path; the lightweight MixColumns algorithms requires four consecutive bytes of the ShiftRows output to be available at the beginning of every four clock cycles. Because ShiftRows is a serial operation, the MixColumns unit would require four extra registers to store these bytes. To achieve this without the use of any additional registers, we delayed the MixColumns operation by four clock cycles so we can use four state registers for storage. With careful placement of components, all 16 state registers, which is exactly enough, can be used for the 12 registers in the BPU and the 4 in MixColumns.

4.4.1 Architecture

In our architecture, every encryption function (AddRoundkey, SubBytes, ShiftRows, MixColumns) is performed serially, requiring only 16 clock cycles in a single AES round. The remaining four clock cycles in the round are devoted to the SubWord operation in the key schedule, during which the state register remains idle. Therefore, the S-box has 100% utilization. The architecture is illustrated in Figure 4.3. To offer a fair comparison of our design against existing works, we kept the S-box the same as the one used in our analysis in Section 4.3, which is Mathew's S-box. The area distribution of Quark-AES components is shown in Figure 4.4.

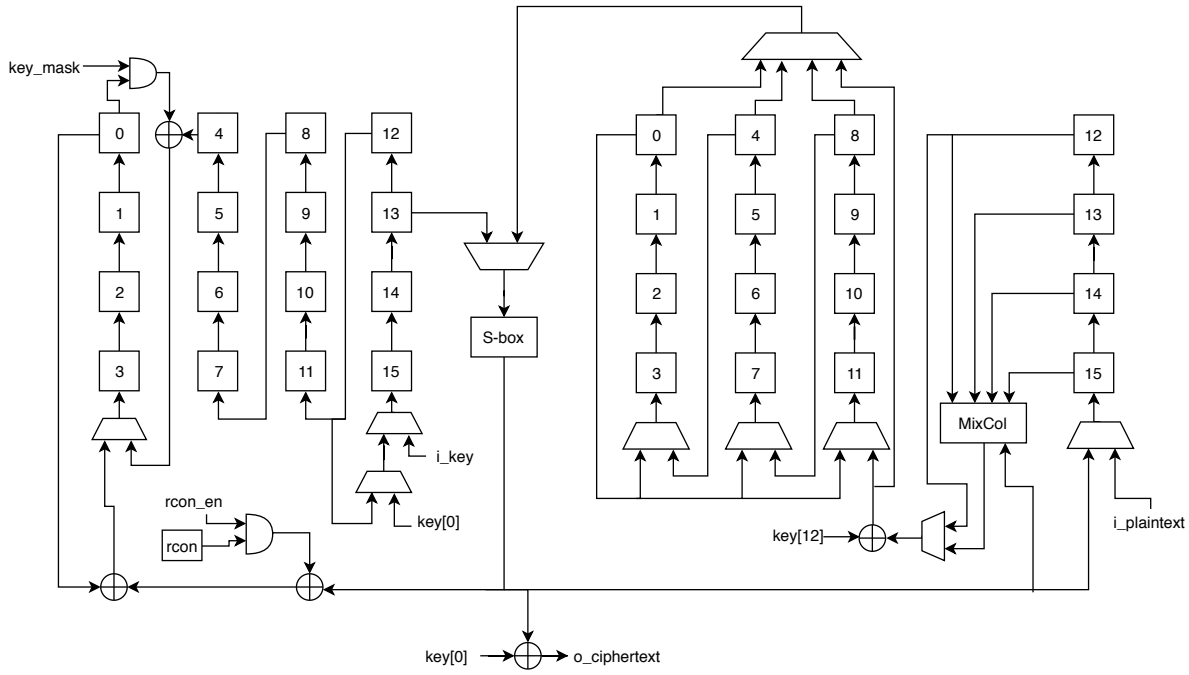


Figure 4.3: Quark-AES architecture

ShiftRows

The ShiftRows operation is implemented as the BPU (Figure 4.1(e)), which consists of state registers [0..11], three 2:1 muxes, and one 4:1 mux. Refer to [11] for implementation specifics.

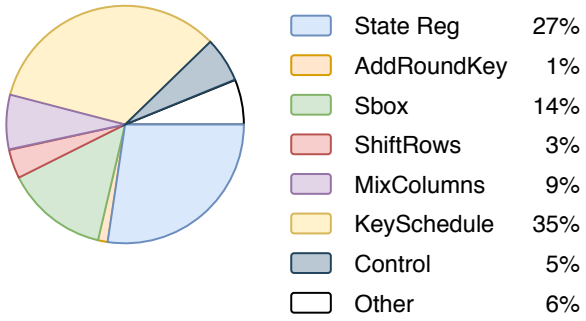
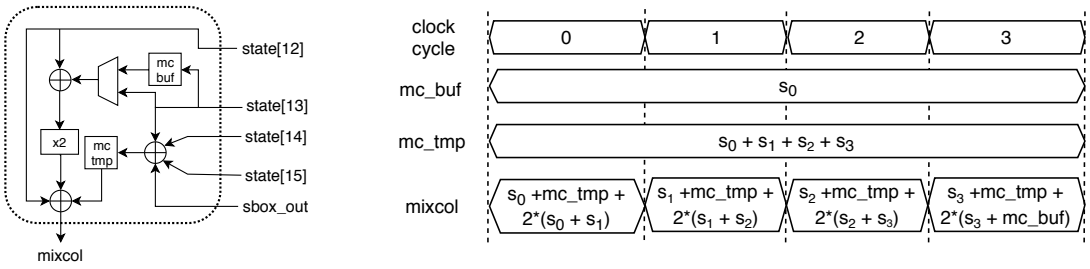


Figure 4.4: Area distribution of Quark-AES components

MixColumns

We employ the lightweight implementation of MixColumns (the third method in Table 4.2), the circuit and behaviour of which is depicted in Figure 4.5. Because the values *tmp* and s_0 are required in clock cycles after they become available, their values must be saved to registers. Every four clock cycles, the sum *tmp* is stored in register *mc_tmp* and value s_0 is stored in register *mc_buf*. The MixColumns operation requires four consecutive bytes to be available after the ShiftRows operation (to calculate *tmp*). Since ShiftRows is a serial operation, it takes four clock cycles for four bytes to be ready. Thus, we delay the MixColumns operation by four clock cycles (until state[12..15] is populated) and place the circuitry after the state[12] register.



(a) Circuit (b) Behaviour

Figure 4.5: MixColumns

Key Expansion

The key generation for the next round begins in clock cycle 16 of the current round. For four clock cycles, the left-most and right-most columns rotate by one byte to generate the left-most column for the next round key. Once the first column is generated, subsequent bytes are generated by doing (key[0] $\oplus$ key[4]), the result of which is written to key[3]. The signal `key_mask` is asserted for 12 clock cycles after the first column of the next round key is generated. By clock cycle 12, all key bytes have been generated, so `key_mask` is set to 0 and the key register needs only to shift the bytes out.

Control

There are two counters in our design: a 5-bit counter (byte_count) for counting the clock cycles within a round and 4-bit counter (round_count) to keep track of the round. Conditions are tested for the beginning and end of each block in Figure 4.6. Control signals are generated for masks and select signals.

4.4.2 Dataflow

Clock cycles 0 to 15

ShiftRows, SubBytes, MixColumns, and AddRoundKey are all performed serially. MixColumns and AddRoundkey are delayed by four clock cycles (relative to the start of the round), starting in clock cycle 4 and extending 4 clock cycles into the next round, shown in Figure 4.6. Shiftrows, requiring only 12 clock cycles, occurs in clock cycles 0 to 11. After clock cycle 11, all the bytes are in the correct order and need only to be shifted out. In the first round, we bypass the MixColumns circuitry and the AddRoundKey takes state[12] instead of MixColumns output.

Clock cycles 16 to 19

The state register remains idle during these clock cycles. The S-box is used to calculate the first four bytes of the next round key, which get stored in the left-most column of the *key* register. During these four clock cycles, the left-most and right-most columns of the key register rotate, while the rest of the key registers remain idle.

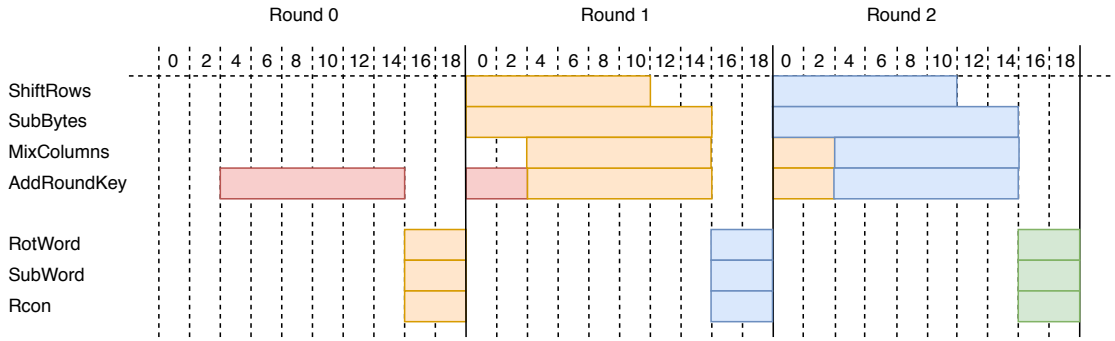


Figure 4.6: Dataflow of Quark-AES showing first 3 rounds. Blocks of the same colour represent operations belonging to the same AES round.

4.5 Results

While an 8-bit datawidth may be an appealing design for some applications, other applications could prioritize throughput over area and desire a higher datawidth. We explored these options and implemented 16-bit and 32-bit datawidth versions of Mathew, Moradi, Hamalainen, and Quark-AES. In the higher datawidth designs of Mathew, the hardware for AddRoundKey and SubBytes is simply duplicated. The MixColumns requires more XORs, but the number of registers remains the same. ShiftRows is performed the same in all datawidths. In Moradi's higher datawidth designs, ShiftRows is performed in one clock cycle in all datawidths. The hardware of AddRoundKey, Sbox, and MixColumns is duplicated. In Hamalainen's higher datawidth architectures, the hardware for AddRoundKey and S-box is duplicated. The higher datawidth versions of MixColumns require more XOR gates and do not require extra registers. The byte permutation units are unique for each datawidth design (based on Jarvinen's circuits [11]) and generally, the number of muxes increases as the datawidth increases.

The results are summarized in Table 4.4. The last two rows include the area results for Atomic-AES and Atomic-AES v2.0, which we synthesized using their publicly available source code. Although they are dual-featured cores supporting both encryption and decryption, it is nice to see where they stand in comparison to encryption-only architectures. The latency values include cycles required for the loading and unloading, and the throughput values do not include simultaneous loading/unloading of the data. The last two columns are optimality metrics of $throughput/area$ and $throughput/area^2$. The first metric is useful if throughput and area are equally important, while the second metric takes

Table 4.4: Summary of designs on STM65nm process

Design	**Data width** (bits)	**Num. S-boxes**	**Area** (GE)	**Latency** (clk cycles)	**Through-put** (bits/clkcycle)	**Power** (mW)	**Energy** (nJ/bit)	**T/A** (10^{-4})	**T/A^2** (10^{-8})
Mathew	8	1	2640	352	0.364	0.109	29.9	1.38	5.22
	16	2	3150	176	0.727	0.174	23.9	2.31	7.33
	32	4	4070	88	1.455	0.238	16.4	3.57	8.78
Moradi	8	1	2370	226	0.566	0.101	17.9	2.39	10.1
	16	2	3000	118	1.085	0.130	11.9	3.62	12.1
	32	4	4110	64	2.000	0.191	9.54	4.87	11.8
Hamalainen	8	2	2260	**176**	**0.727**	0.145	20.0	**3.22**	14.2
	16	4	2980	**88**	**1.455**	0.200	13.7	**4.88**	16.4
	32	8	4290	**44**	**2.909**	0.269	9.24	**6.78**	15.8
Quark-AES	8	1	**1960**	216	0.593	**0.091**	**15.3**	3.02	**15.4**
	16	2	**2420**	108	1.185	**0.129**	**10.9**	4.82	**19.6**
	32	4	**3530**	54	2.370	**0.172**	**7.25**	6.71	**19.0**
Atomic-AES	8	1	2680	226	0.566				
Atomic-AES v2.0	8	1	2480	246	0.520				

power into account by using area as its approximation. Power is dependent on implementation technology, which makes it difficult to compare it against other libraries. Using area to approximate it allows the result to be more dependent on design and less on the library used. For convenience, we reiterate Mathew, Moradi, and Hamalainen's reported area results: 1947, 2400, and 3100 GE, respectively. At 1960 GE, Quark-AES has the smallest area of all the designs. It features a 13% improvement in area, 9% improvement in power, 14% improvement in energy, and 8% improvement in throughput/area^2 over the runner-ups when synthesized on STM 65 nm. In real-world applications, it is often the case that data may not be ready or available every clock cycle. To be practical, our design supports bubbles during loading. The 8-bit Quark-AES outperforms other 8-bit designs in power, energy, and T/A^2. Figure 4.7 shows how well the architectures extend to higher datawidths (16-bit and 32-bit). While Mathew's design has the highest area at an 8-bit datawidth, the 32-bit version is smaller than Moradi's and Hamalainen's.

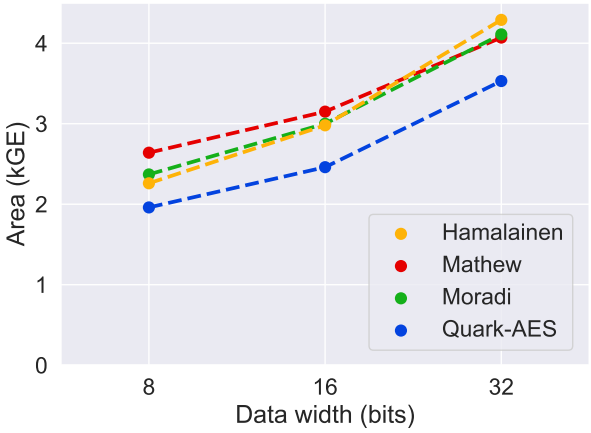


Figure 4.7: Area vs data width

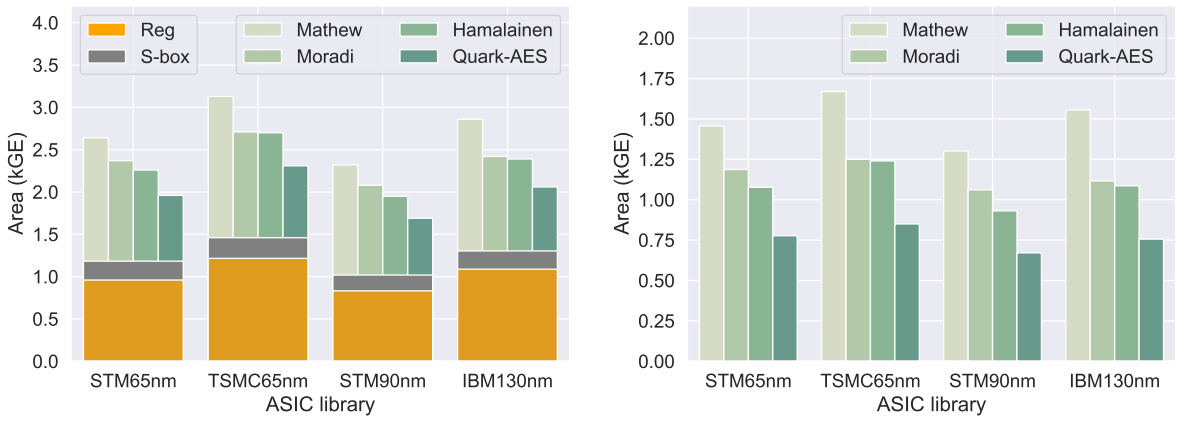
It is difficult to compare works that use different ASIC libraries and toolchains. Although the GE metric attempts to normalize across different technologies, there are still considerable variations among different libraries, illustrated in Table 4.5. As registers make up approximately half the area of an 8-bit design, choosing a library with a small GE/flop will naturally lead to smaller overall area. Figure 4.8 illustrates the area variations of the AES designs on four different ASIC libraries. Figure 4.8(a) shows the area distribution of the state and key registers and the S-box, which are independent of architectural design choices. The area of the state and key registers, shown in orange, also serve as the minimum area threshold of a design. A tighter lower bound would include the area of other

Table 4.5: Area of common cells for different libraries

	NAND		**AND**		**MUX**		**XOR**		**flop**		**flop$_{ce}$** *	
	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE	μm^2	GE
STM 65 nm	2.08	1.00	2.60	1.25	4.16	2.00	4.16	2.00	7.80	3.75	10.40	5.00
TSMC 65 nm	1.44	1.00	2.16	1.50	3.24	2.25	3.60	2.50	6.84	4.75	9.36	6.50
STM 90 nm	4.39	1.00	5.49	1.25	8.78	2.00	7.68	1.75	14.27	3.25	19.76	4.50
IBM 130 nm	5.76	1.00	7.20	1.25	12.96	2.25	11.52	2.00	24.48	4.25	34.56	6.00

*Chip-enabled flip-flop

components. The S-box that we use is roughly 20% larger than the smallest published S-box [36] to date. But this area difference can be used to account for the other AES components. We will consider the lower bound for AddRoundKey to have 8 XOR gates, ShiftRows to have no area, MixColumns to have a multiply-by-2 unit and 8 XOR gates, and KeySchedule to have 8 XOR gates and 8 registers for RCON counter. Altogether they amount to less than 84 GE on STM 65 nm. This area is greater than the area difference between our S-box and an optimal S-box. Thus, we believe the combined area of the state and key registers and our S-box provides a tighter, but still loose, lower bound. While, in theory, the same gate could be reused for multiple components, this would increase the control circuitry and most likely negate the area decrease obtained from reuse. In Fig-



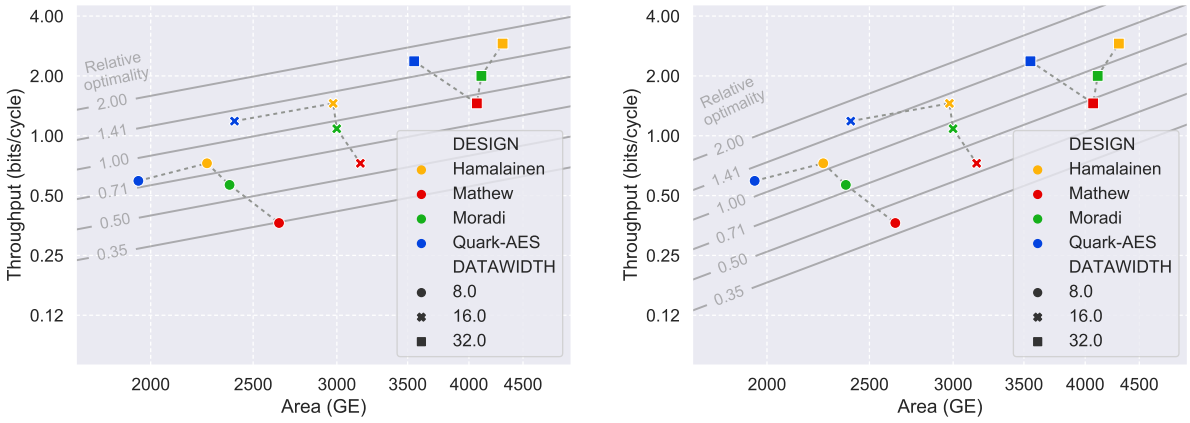
(a) Total area

(b) Area of architectural portion

Figure 4.8: Area on four different ASIC libraries

ure 4.8(a), the area of Quark-AES is the lowest in each ASIC library category and the area of Mathew is the highest in each category. But if we compare Quark-AES's area on TSMC 65 nm to Mathew's area on STM 90 nm, they are almost the same. To more fairly compare architectures of different designs synthesized with different libraries, only the area owing to architectural design decisions (*i.e.* excludes state and key registers and S-box) should be compared. This area is depicted in Figure 4.8(b), which shows smaller area discrepancies across different ASIC libraries, for a given design. Now, Mathew's area on STM 90 nm is much higher than Quark-AES's area on TSMC 65 nm, which better reflects the actuality that Quark-AES's architecture achieves smaller area than Mathew's.

A comparison of the designs in terms of two optimality measures, $\frac{\text{throughput}}{\text{area}}$ and $\frac{\text{throughput}}{\text{area}^2}$, is illustrated in Figure 4.9. Data points of the same colour have the same architecture and the marker style indicates the datawidth in bits. In both graphs, the y axis is *log* scaled. The x axis in Figure 4.9(a) is scaled as $log(x)$ while the x axis in Figure 4.9(b) is scaled as $log(x^2)$. The grey contour lines represent normalized optimality values: $\frac{\text{optimality}}{\text{average optimality}}$. The average optimality of all 12 datapoints is represented by the contour line labeled 1.00. Quark-AES consistently offers the lowest area in each datawidth category. In Figure 4.9(a), its optimality is slightly below Hamalainen's for the 8-bit architecture, but on par for the 16-bit and 32-bit designs. In Figure 4.9(b), Quark-AES outperforms (*i.e.* has higher optimality than) every other architecture in each datawidth category.



(a) Throughput/area

(b) Throughput/area^2

Figure 4.9: Optimality trends

4.5.1 Optimized Quark-AES

In previous sections, many components were kept the same across all designs in order to limit the comparison to just the architecture. Therefore, the final area results were suboptimal. In Table 4.6, an optimized version of Quark-AES is provided. We replace the S-box with Canright's version [29] using the source code from [37]. Since Canright's publication, many new works (in [38],[36]) emerged with smaller S-box implementations. Using any of these state-of-the-art S-boxes with Quark-AES will likely achieve even lower area results than Table 4.6 reports. We also remove the round counter and use the RCON counter in its place. The optimized Quark-AES encryption module is only 1870 GE, which is smaller than all previously reported area results.

Table 4.6: Optimized versions of 8-bit Quark-AES

Sbox	**Counters**	**Area** (GE)	**Energy** (nJ/bit)
Mathew	round_counter byte_counter RCON	1960	15.4
Canright	byte_counter RCON	1870	15.9

4.6 Summary

In this chapter, an analysis of several architectural and component options for 8-bit AES designs was conducted. ShiftRows can be implemented in one, three, or 16 clock cycles; MixColumns can be implemented serially or in parallel; and the KeySchedule can have a row-major or column-major design. Three existing encryption cores from the works of Hamalainen, Moradi, and Mathew were implemented and synthesized on a common technology to provide a benchmark. In this benchmark, Hamalainen's architecture came out on top. Finally, a novel architecture, Quark-AES, was presented. It offers the benefits of low latency, single S-box instance, and no requirement to reorder inputs or outputs externally. Quark-AES has a 13% improvement in area and 8% improvement in throughput/area^2 over Hamalainen's design on STM 65 nm process.

Chapter 5

Energy Optimization in AES and PRESENT

In this part of the thesis, the goal is to reduce the energy consumed per bit, which is the primary concern in battery-powered devices. The less energy the operation takes, the more work can be done on one battery charge. Some of the energy optimizations that have been explored in the research community include loop unrolling and operand gating, which were previously mentioned in Sections 2.3 and 3.2.3. Section 5.1 describes a parameterized implementation, applied to both PRESENT and AES, which allows for easy configuration during design time. The datawidth in each implementation is the cipher's block size. Section 5.2 explains the low-energy techniques and their implementation. Section 5.3 describes the toolchain and libraries used. The results of the techniques, applied to both ciphers, are presented in Section 5.4, which discusses the energy efficiency of non-pipelined and pipelined designs, and the optimal glitch filter spacing and location. Finally, Section 5.5 summarizes and concludes the chapter.

5.1 Parameterized Implementation

In this work, a parameterized implementation of both ciphers was developed. A parameterized design is a design that can be easily re-used in different situations, through the configuration of various developer-defined parameters. By choosing the parameters before

synthesis, a variety of designs can be obtained from a single RTL core. To achieve this, the hardware descriptive language must support such a feature. In VHDL, this is done through the use of 'generics', shown in Figure 5.1.

```
entity PRESENT is
    generic(
        rpcc : integer := 2;
        ccps : integer := 1
    )
    port(
        ...
    )
end entity PRESENT;
```

Figure 5.1: VHDL syntax for 'generic'

The parameters in the parameterized implementation of AES and PRESENT are *rpcc* and *ccps*. The parameter *rpcc* represents the number of encryption rounds per clock cycle: that is, the degree of loop unrolling. The parameter *ccps* refers to the number of clock cycles per stage: in other words, the degree of pipelining (*i.e.* $ccps = 1$ means fully pipelined while $ccps > 1$ means partially pipelined). With these two parameters, different combinations of loop unrolled and pipelined designs can be obtained and, depending on the application, the parameters can be carefully chosen to get the desired tradeoffs in throughput, area, and energy. For PRESENT, the *rpcc* and *ccps* values that are explored are 1, 2, 4, 8, 16, and 32, and for AES, they are 1, 2, 3, 4, 6, and 11. These values were selected so that as few hardware components as possible would have a utilization less than 1, where utilization, U_i, is defined below:

$$U_i = \frac{\text{Number of clock cycles that component } i \text{ is in use}}{\text{Number of clock cycles required to complete encryption operation}} \tag{5.1}$$

A utilization less than 1 means components are unused some of the time. The product of *rpcc* and *ccps* should be less than or equal to the number of rounds. This is best reasoned using the formula for number of stages, S:

$$S = \left\lceil \frac{N}{rpcc \times ccps} \right\rceil \tag{5.2}$$

where N represents the number of rounds in the algorithm. Suppose $rpcc \times ccps > 32$ for PRESENT. Then the fraction inside the ceiling function would be less than 1, which

means the utilization is less than 1 and this is undesirable. Since the number of rounds in AES is a prime, it cannot be helped that some components will have utilization less than 1 when the degree of loop unrolling is greater than 1. For AES, the condition is restricted to $rpcc \times ccps < 12$.

A number of other helpful parameters can be derived from the *rpcc* and *ccps* values. Let B_b represent the number of bits per block and B_k the number of bits per key.

To obtain *ccps*, given S:

$$ccps = \left\lceil \frac{N}{rpcc \times S} \right\rceil \tag{5.3}$$

To obtain *rpcc*, given S:

$$rpcc = \left\lceil \frac{N}{ccps \times S} \right\rceil \tag{5.4}$$

The number of clock cycles, C, required to complete the encryption operation is:

$$C = \left\lceil \frac{N}{rpcc} \right\rceil \tag{5.5}$$

The throughput, T, in bits/cycle, is calculated as:

$$T = \frac{B_b}{ccps} \tag{5.6}$$

The number of state registers, R_p, is:

$$R_p = S \times B_b \tag{5.7}$$

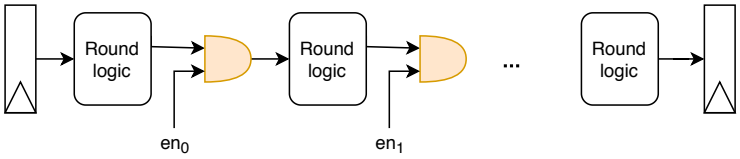
The number of key registers, R_k, is:

$$R_k = S \times B_k \tag{5.8}$$

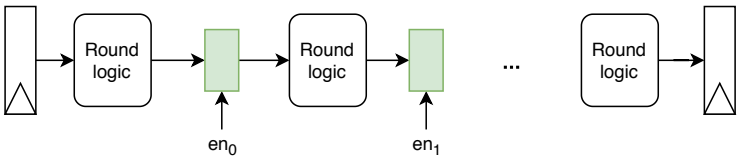
5.2 Glitch Filters

In a non-pipelined design, loop unrolling can reduce latency, as it reduces the number of clock cycles required and subsequently the number of registers in the datapath. Furthermore, it should eliminate the energy otherwise spent in those registers and, in turn, reduce the overall energy per bit. However, in fact, an increase in energy is observed in loop unrolled designs because it introduces more glitching, which arises from the longer combinational paths. The effectiveness of loop unrolling on energy depends on the degree of unrolling, and there exists some optimal number of rounds per clock cycle that offers the greatest energy savings.

Fortunately, the glitching energy from loop unrolling can be reduced by applying operand gating, a low-power technique that blocks the propagation of switching activity through the circuit. One method to achieve this is by gating each combinational round with AND gates, illustrated in Figure 5.2(a). As a result, the output of a round is only propagated when the control signal, en_i, is asserted. These *enable* signals effectively turn a round 'on' or 'off'. The idea is to turn a round off until the signals from the previous round have stabilized. Another implementation of operand gating, illustrated in Figure 5.2(b),



(a) AND-gate gating. AND-gates are shown in orange.



(b) Latch gating. Latches are shown in green.

Figure 5.2: Operand gating implementations

requires the use of latches and works similarly to the AND-gate method. The positive level-sensitive latches go transparent only when en_i is asserted. Otherwise, they latch onto the old value, preventing switching activity (glitching). There is some area overhead which accompanies operand gating due to the AND gates or latches themselves as well as the circuitry required to generate the enable signals.

5.2.1 Generating *enable* Signals

The enable signals need to asssert in successive intervals. In previous works, this was accomplished using a delay chain [14], [15]. Instead of using delay units, we've opted to use a secondary clock generated externally and passed into an input port of the cipher module. On embedded systems, the system clock is usually generated from a crystal oscillator using a circuit divider. It would not be difficult to generate the secondary clock in a similar fashion; simply a different constant in the circuit divider is required. Thus, the cipher module receives two clock signals as inputs: *clk* and clk_{fast}, shown in Figure 5.3. Input ports are listed on the left in Figure 5.3 while output ports are shown on the right.

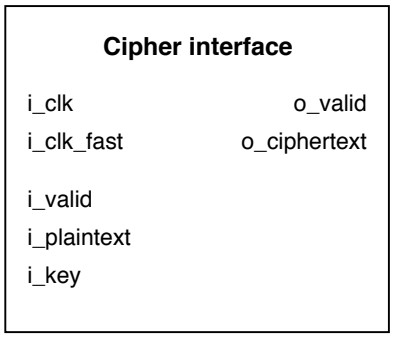
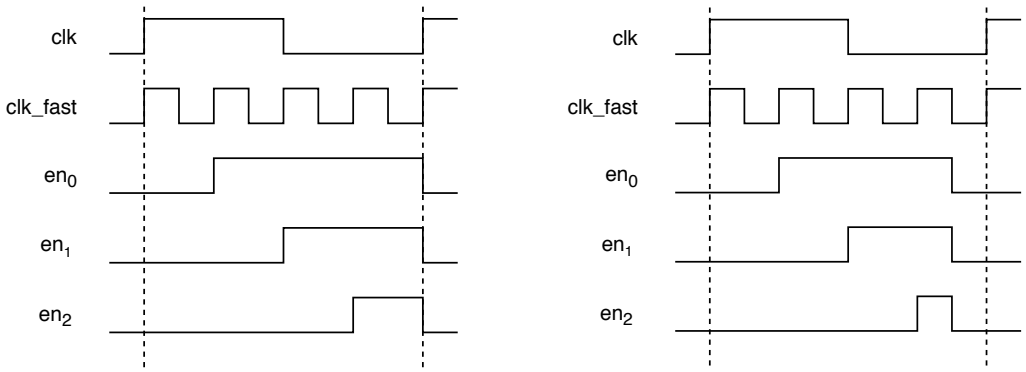


Figure 5.3: Interface of cipher module

The enable signals are generated using a secondary clock, clk_{fast} whose frequency is *rpcc* times that of the primary clock, *clk*. For *rpcc* rounds, $rpcc - 1$ enable signals are needed. Within a clock cycle of *clk*, the enable signals are asserted successively at each rising edge of clk_{fast}. Figure 5.4 depicts the waveforms of the enable signals for AND-gate and latch glitch filtering methods. The waveforms of the enable signals for the two methods are slightly different for a reason. In the AND-gate method, the enable signals must remain high until the round outputs are stored in a register. With latches, the enable signals can go low at any time after the stabilized value has been latched. Theoretically, the enable signals in Figure 5.4(a) could be used for the latch method as well. However, having the falling edge of the enable signals close to the rising edge of *clk* can cause hold and setup violations. This is further explained in Section 5.2.2.

The enable signals for the AND-gate glitch filter and latch-based glitch filter are generated by the circuit depicted in Figure 5.5. The circuit elements in white are required to generate the enable signals for the AND-gate method. In addition to this, the green



(a) AND-gate (b) Latch

Figure 5.4: Enable signals waveforms

circuity is required to generate the enable signals for the latch-based method. A delayed clock signal, clk_d, follows the clk signal with a delay of one clk_{fast} clock cycle. Pulse z is a pulse with width equal to one clock cycle of clk_{fast} occuring right after the rising edge of clk. The first enable signal, en_0^{AND}, is simply the signal z inverted. Subsequent enable signals are generated using delayed versions of it ($en\_d_i^{AND}$) that are masked by en_0^{AND}. To

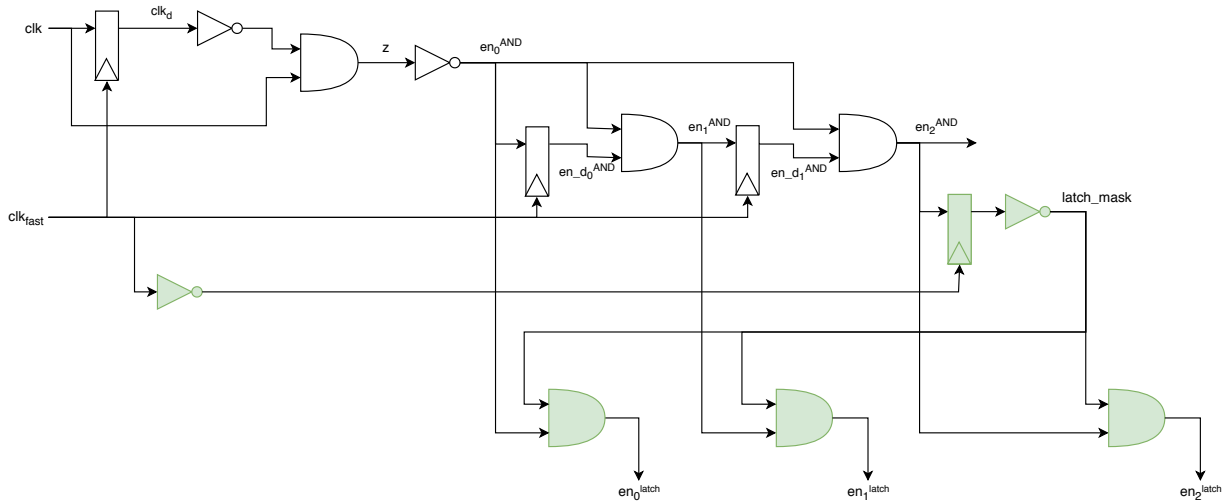


Figure 5.5: Circuit for *enable* signal generation for rpcc=4

achieve the enable signals for the latch-based scheme, the enable signals of the AND-gate method are masked by a signal ($latch\_mask$) that causes the bits to go to zero half a clk_{fast}

clock cycle before the next rising edge of clk. The waveforms of the intermediate signals in Figure 5.5 are shown in Figure 5.6.

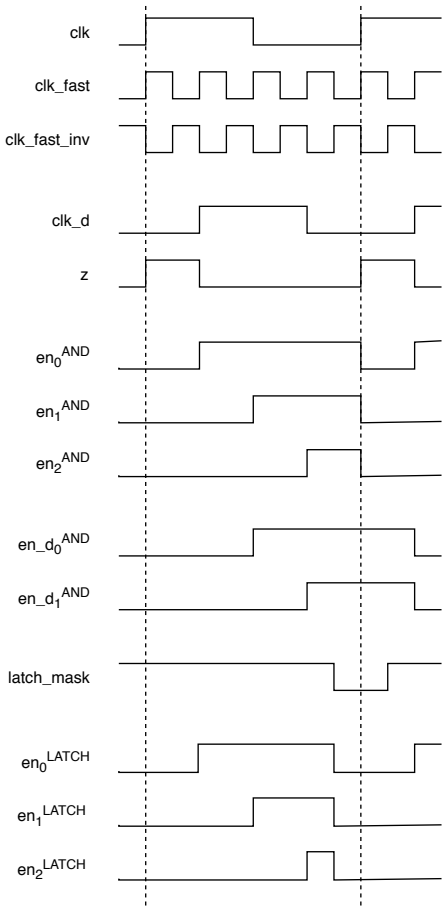


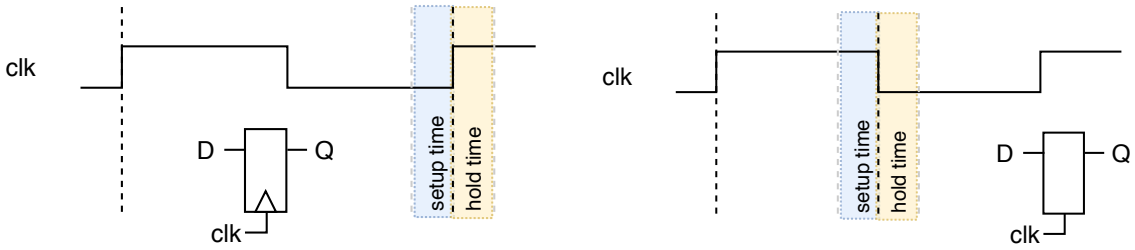
Figure 5.6: Waveform for signals in Figure 5.5

In order for the AND-gate glitch filter to function correctly and to properly eliminate glitches, the time between the assertions of successive enable signals and, in turn, the period of clk_{fast}, $t_{clk_{fast}}$, must be longer than the propagation delay, t_p, through one combinational round. That is, we need $t_{fast\_clk} > t_p$. As for the latch-based filter, the duration of the transparency of a latch must be longer than the propagation delay through one round. The latch that has the shortest transparency duration is the one that gates the last combinational round; it is transparent for half a clk_{fast} clock cycle. Thus, the constraint imposed on the clock period of clk_{fast} is $t_{fast\_clk} > 2t_p$. The number of unrolled rounds

that can be implemented depends on the periods of clk and clk_{fast}; $rpcc_{max} = \frac{t_{clk}}{t_{clk_{fast}}}$.

5.2.2 Setup and Hold Timing

The waveforms shown in Figure 5.4 are ideal signals. In practice, setup and hold timing need to be taken into account. For a rising-edge-triggered register, setup time refers to the amount of time the data signal must be stable before the rising edge of the clock. Similarly, the hold time refers to the amount of time the data signal must be stable after the rising edge of the clock. For a positive level-sensitive latch, the setup time denotes the time the data signal must be stable before the falling edge of the clock. The hold time denotes the time the data signal must be stable after the falling edge of the clock. Hold and setup time requirements for a flip-flop and a latch are illustrated in Figure 5.7.



(a) Rising edge-triggered flip-flop (b) Positive level-sensitive latch

Figure 5.7: Setup and hold timing for a flip-flop and latch

In the AND-gate scheme, the enable signals must satisfy the hold time and thus, should go low after the next rising edge of the primary clock (so that the output of the last unrolled round remains stable). In other words, they need to be held stable up until the end of the yellow region in Figure 5.7(a).

For the latch design, in each primary clock cycle, the enable signals must go low before data in the next clock cycle changes. If the enable signals go low too close to the rising edge of clk, which is when data changes, then a setup or hold violation may occur. Even if a timing violation does not occur, we want to prevent the erroneous case of latching onto the data values of the next clock cycle. For this reason, the latch enable signals go low half a clk_{fast} clock cycle before rising edge of clk.

5.3 Methodology

The AND-gate and latch-based glitch filters were implemented for AES and PRESENT block ciphers. The propagation delay of one encryption round is determined by synthesizing the design with one round per clock cycle and obtaining the value reported by the tools. In PRESENT, the propagation delay is 1.06 ns and in AES 3.04 ns.

The parameterized implementation presented in Section 5.1 is enhanced to accommodate the baseline, latch, and AND-gate implementations. The glitch filter method can be selected via the generic parameter *method*, which can take values 0, 1 or 2, representing the baseline, latch-based glitch filtering, and AND-gate glitch filtering designs, respectively. We will also explore the optimal spacing between glitch filters, which can be selected using the parameter gfs (glitch filter spacing), as well as the optimal placement of a fixed number of filters, which can be selected using the parameter gfl (glitch filter location).

The low-energy optimizations are synthesized using ST Micro 65 nm CORE65LPLVT at 1.25 V. Compilations were performed using the *compile_ultra* command, with clock gating and preservation of component hierarchies. Mentor Graphics ModelSim SE v10.5c was used for simulations. Synopsys Design Compiler version P-2019.03 was used for logic synthesis and Cadence Encounter v14.13 for physical synthesis, with density set to 90%. Power analysis is also conducted by Cadence Encounter. Energy consumption (nJ/bit) is derived using Equation 3.1. Timing simulations were performed using a clock period of 512 ns ($\approx$ 2 MHz frequency), which is sufficient time to accommodate fully unrolled designs of PRESENT and AES.

There are many variables at play in this analysis (*rpcc*, *ccps*, *method*, gfs, gfl) and they all affect energy, area, and throughput differently depending on the combination of variables. There are too many combinations to examine all of them. Thus, only one variable will be changed at a time. First, the energy, area, and throughput for each method are assessed in a non-pipelined implementation (*i.e.* single stage). In other words, only *rpcc* is varied (*ccps* is derived from *rpcc* and S = 1). Next, we investigate the effect of changing the glitch filter spacing and the location of glitch filters, still only for non-pipelined implementation. Finally, we explore the effect of glitch filters on energy in pipelined implementations.

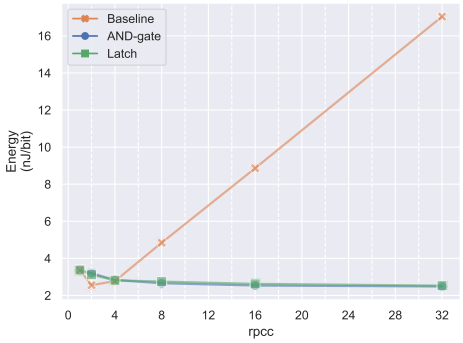
5.4 Results and Analysis

5.4.1 Glitch Filters in Unrolled Rounds

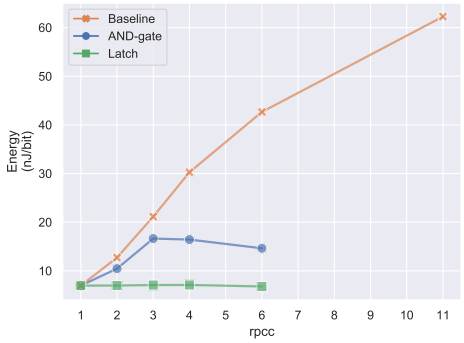
In this section, we consider a non-pipelined design with glitch filters applied after every round. Figure 5.8 depicts the energy, area, and throughput trends for PRESENT and AES. In PRESENT, the optimal glitch filter method depends on the *rpcc* value. For $rpcc \leq 4$, the baseline method is the most energy efficient, as indicated in Figure 5.8(a). For $rpcc > 4$, the AND-gate scheme provides the lowest energy, although it is better than the latch method by a negligible amount. The non-monotonicity in the PRESENT baseline trend demonstrates that simply increasing the degree of loop unrolling does not increasingly save energy. At its global minimum ($rpcc = 2$), the energy saved from the reduction in registers is greater than the energy introduced by glitches, thus the overall energy consumption is lower than the design with no loop unrolling (*i.e.* $rpcc = 1$). Still considering only the baseline trend, for $rpcc > 2$, the glitching energy surpasses the energy saved from the reduction in registers, thus the overall energy increases. When a glitch filtering technique is applied, the glitching energy is reduced and the energy overhead instead comes from the glitch filter circuitry, whose energy is much smaller than the glitching energy. Instead of having a local minimum such as in the case of the baseline, the AND-gate and latch glitch filtering trends appear to be monotonically decreasing.

For AES, the latch glitch filtering technique provides the most energy savings for all values of *rpcc*. The design that offers the lowest energy has $rpcc = 6$, illustrated in Figure 5.8(b). Unfortunately, synthesis results could not be obtained for AES design with AND-gate and latch glitch filters for $rpcc = 11$ because the gate instance count exceeds the maximum that our tool license supports. As the block size of AES (128) is twice that of PRESENT (64) and the AES operations are more complex than that of PRESENT, the baseline energy consumption of AES in Figure 5.8(b) is much greater than that of PRESENT. Moreover, AES has a deeper datapath for a single round. As a result, applying round unrolling without glitch filters is not beneficial; the glitching energy of unrolled rounds always exceeds the energy saved by the reduction in registers. Like in the case of PRESENT, using glitch filters significantly reduces the glitching energy without adding much energy overhead.

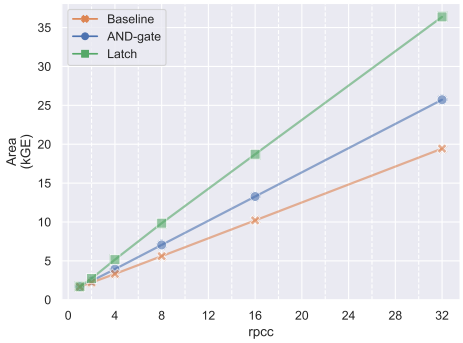
In Figure 5.8(c) and Figure 5.8(d), the area overhead of each method is presented. Since there is only one stage, the area is affected only by *rpcc* and the relationship is linear. This makes sense because *rpcc* directly indicates how many duplicate combinational hardware



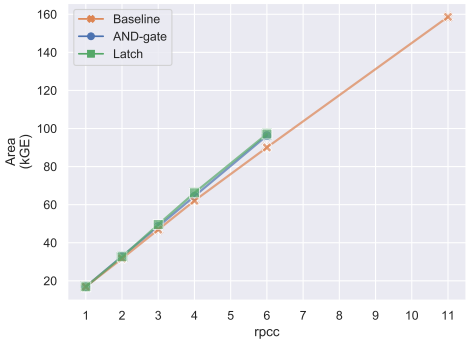
(a) PRESENT Energy vs rpcc



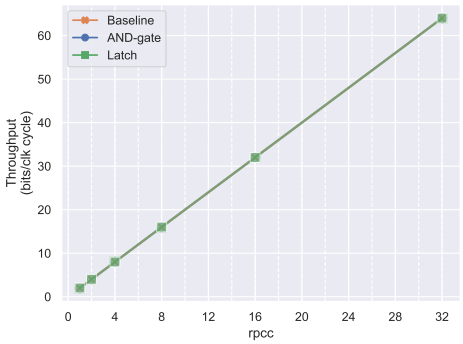
(b) AES Energy vs rpcc



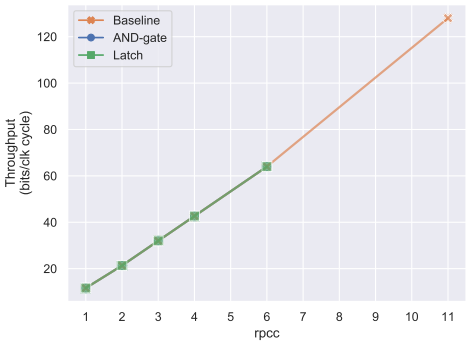
(c) PRESENT Area vs rpcc



(d) AES Area vs rpcc



(e) PRESENT Throughput vs rpcc

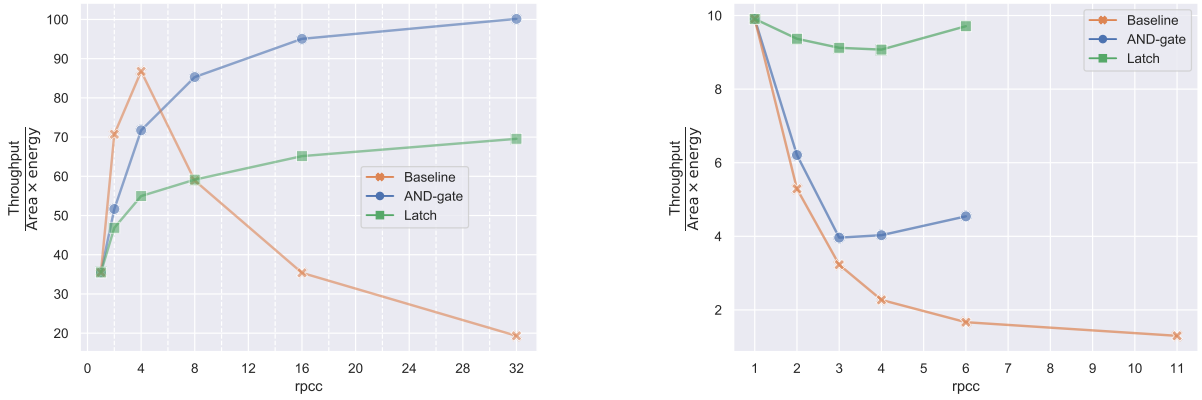


(f) AES Throughput vs rpcc

Figure 5.8: Energy, area, and throughput trends for single stage design

components are required. Because AES is a much larger cipher than PRESENT, the area overhead of applying glitch filtering in AES is, as a percentage, much smaller than applying it in PRESENT. In absolute terms, having glitch filtering in AES results in a higher area overhead due to its block size being twice that of PRESENT. The throughput remains independent of the glitch filtering method, illustrated in Figure 5.8(e) and Figure 5.8(f).

When there are multiple metrics by which to evaluate a design and an overall assessment is desired, it can be helpful to use an optimality measure which combines the important metrics. For instance, if both throughput and area are important in a design, we can use the optimality measure $O = \frac{throughput}{area}$, where higher values of O indicate a better design. The metrics whose higher values indicate a better design should be in the numerator while metrics whose lower values indicate a better design (*i.e.* costs) should be in the denominator. In this discussion, energy, area, and throughput are all important. Thus, we can use the optimality measure $\frac{throughput}{energy \times area}$. Figure 5.9 plots this normalized optimality measure against *rpcc* for a non-pipelined implementation. For PRESENT, the design that gives the highest optimality is the AND-gate design with $rpcc = 32$, shown in Figure 5.9(a). For AES, the optimal designs are the latch designs with $rpcc = 1$, shown in Figure 5.9(b).



(a) PRESENT (b) AES

Figure 5.9: Optimality trends

Table 5.1 summarizes the designs that achieved the lowest energy for each *rpcc*. The design with the global minimum energy is in bold for each cipher.

Table 5.1: Summary of designs having the lowest energy for each *rpcc* value

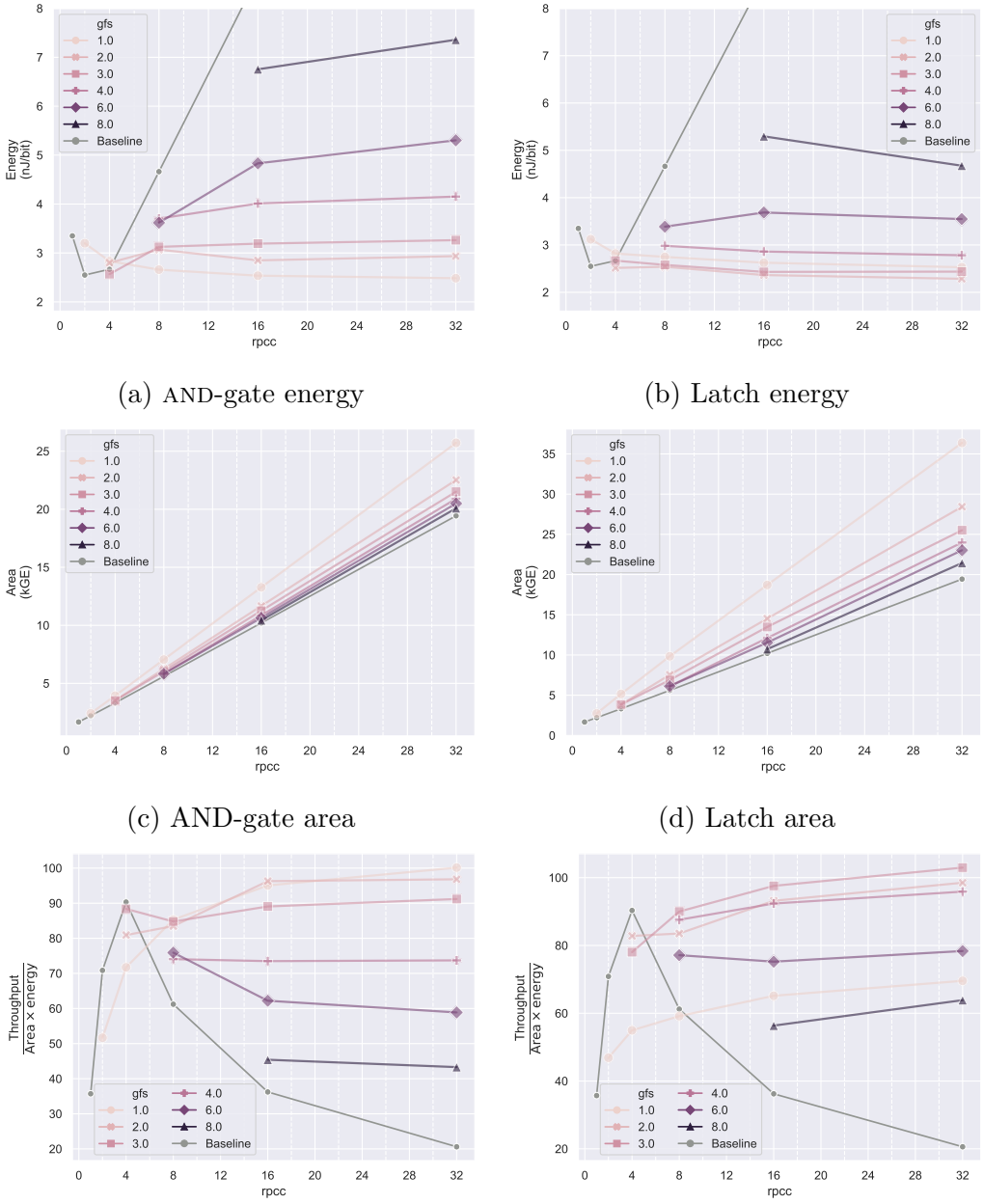
Cipher	**rpcc**	**Lowest energy method**	**Energy** (nJ/bit)	**Area** (GE)	**Optimality**
PRESENT	2	Baseline	2.55	2210	71.0
	4	Baseline	2.78	3320	86.7
	8	Latch	2.75	9840	59.1
	16	Latch	2.63	18700	65.1
	32	Latch	**2.53**	36400	69.5
AES	2	Latch	6.98	32600	9.38
	3	Latch	7.08	49600	9.11
	4	Latch	7.10	66200	9.08
	6	Latch	**6.79**	97100	9.71

5.4.2 Glitch Filter Spacing in Unrolled Rounds

In the previous section, the glitch filters were placed between every combinational round within a clock cycle. Since the glitch filter circuitry consumes energy, some of the glitching energy saved by the glitch filter will be offset by the energy cost of the glitch filter itself. The goal of this section is to find the optimal spacing of glitch filters at which the total energy consumption is minimized. The gfs parameter is used to indicate the glitch filter spacing, where glitch filters are applied every gfs round.

Figure 5.10 shows how the glitch filter spacing in PRESENT affects the energy consumption for different values of *rpcc*. For the AND-gate scheme in Figure 5.10(a), the lowest energy consumption occurs at $rpcc = 32$ and $gfs = 1$. When $rpcc > 4$, applying AND-gate glitch filters between every round provides the biggest energy savings. For the latch-based scheme in Figure 5.10(b), the lowest energy consumption occurs at $rpcc = 32$ and $gfs = 2$. The energy savings are maximized when the latch-based glitch filter is applied every second round, for all *rpcc* values. An interesting observation is made when looking at combinations $rpcc = 4$, $gfs = 2$ and $rpcc = 4$, $gfs = 3$ for the AND-gate method. Both combinations have the same number of glitch filters per round (*i.e.* 1), but the energy results differ. The observation is also present when looking at $rpcc = 8$: the energy for $gfs = 6$ is lower than the energy for $gfs = 4$. We will see why this is the case in the Section 5.4.3.

The effect of the glitch filter spacing on AES is depicted in Figure 5.11. For the AND-gate glitch filtering scheme, the baseline without unrolled rounds achieved the lowest



(a) AND-gate energy

(b) Latch energy

(c) AND-gate area

(d) Latch area

(e) AND-gate optimality

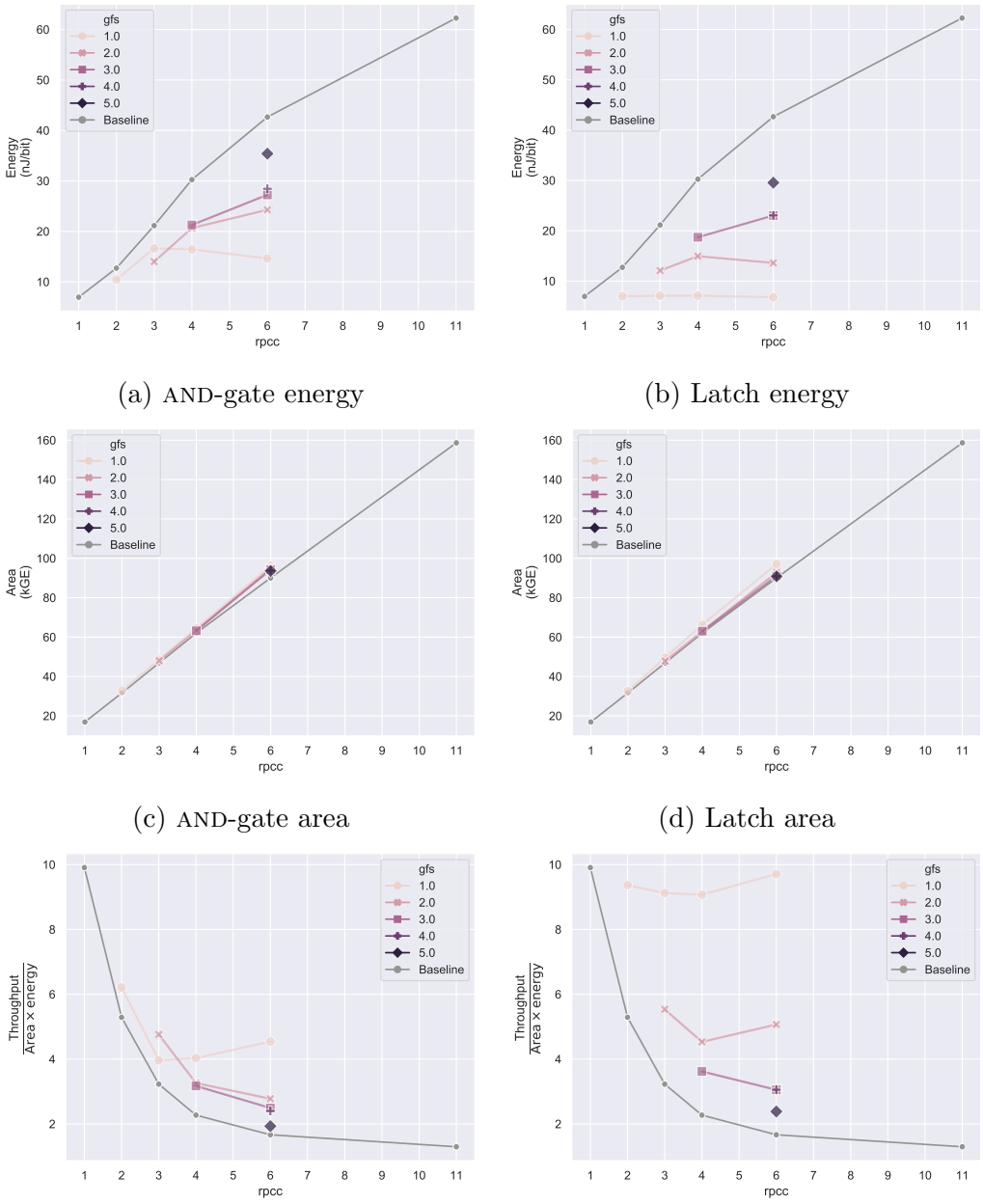
(f) Latch optimality

Figure 5.10: Varying glitch filter spacing in PRESENT

energy. For $rpcc > 1$, having a glitch filter spacing of 1 achieves the lowest energy for all degrees of unrolling except $rpcc = 3$. Generally, the energy decreases as gfs decreases, indicating that the energy cost of the glitch filter is small compared to the energy that it saves. The latch-based method, in Figure 5.11(b), shows similar trends to the AND-gate scheme. A glitch filter spacing of 1 achieves the best results and compared to the AND-gate scheme, it is more energy efficient. While the AND-gate method achieves the lowest energy at $rpcc = 2$ and $gfs = 1$, the latch is more efficient at $rpcc = 6$ and $gfs = 1$. Because AES is a much larger cipher than PRESENT, the area overhead of either glitch filtering scheme is small, illustrated in Figures 5.11(c) and 5.11(d). The optimality trends, in Figure 5.11(e) and Figure 5.11(f), correspond with the energy trends in each glitch filtering scheme; the configurations that prevailed in the energy metric also prevailed in the optimality metric. Table 5.2 summarizes the gfs value that achieved the lowest energy for each $rpcc$ value. Optimal energy consumption values are bolded for each cipher.

Table 5.2: Summary of best glitch filter spacing for each $rpcc$ value

Cipher	**Method**	**rpcc**	**gfs**	**Energy** (nJ/bit)	**Area** (GE)	**Optimality**
PRESENT	AND-gate	2	-	2.55	2210	71.0
		4	3	2.57	3530	88.2
		8	1	2.66	7050	85.3
		16	1	2.54	13300	94.7
		32	1	2.48	25700	100.4
	Latch	2	-	2.55	2210	71.0
		4	2	2.52	3840	12.4
		8	2	2.54	7540	24.5
		16	2	2.36	14500	52.8
		32	2	**2.29**	28400	108.7
AES	AND-gate	2	1	10.5	32900	6.18
		3	2	14.0	48000	4.76
		4	1	16.4	64400	4.04
		6	1	14.6	96300	4.55
	Latch	2	1	6.98	32600	9.38
		3	1	7.08	49600	9.11
		4	1	7.10	66200	9.08
		6	1	**6.79**	97100	9.71



(a) AND-gate energy

(b) Latch energy

(c) AND-gate area

(d) Latch area

(e) AND-gate optimality

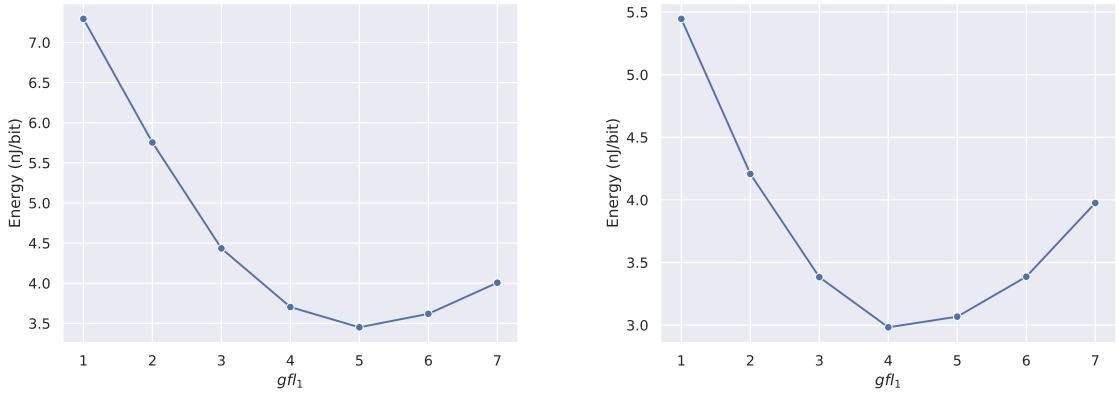
(f) Latch optimality

Figure 5.11: Varying glitch filter spacing in AES

5.4.3 Glitch Filter Locations in Unrolled Rounds

In Section 5.4.2, the spacing between glitch filters was varied, which affected the number of glitch filters as well as their location. However, the optimal location of the glitch filters was not assessed. In situations wherein hardware area is a premium and only a limited number of glitch filters can be employed, it is important to know where the glitch filters should be placed such that energy is minimized. This section explores the optimal location of having one, two, and three glitch filters in PRESENT. Let N_{gf} represent the number of glitch filters. The parameter gfl_x (glitch filter location) will be used to denote the location of the x^{th} glitch filter, where $1 \leq x \leq N_{gf}$. For example, $gfl_2 = 3$ indicates that the second glitch filter is placed after the third combinational round. We limit the exploration to $rpcc = 8$.

We will first explore the optimal location of a single glitch filter. Figure 5.12 illustrates the energy per bit as a function of the location of the single glitch filter. The global minimum at $gfl_1 = 5$ in Figure 5.12(a) indicates that the energy is minimized when the AND-gate glitch filter is placed after the fifth combinational round. For the latch glitch filtering scheme, however, optimal energy savings are achieved when the glitch filter is placed after the fourth combinational round, shown in Figure 5.12(b). To understand why



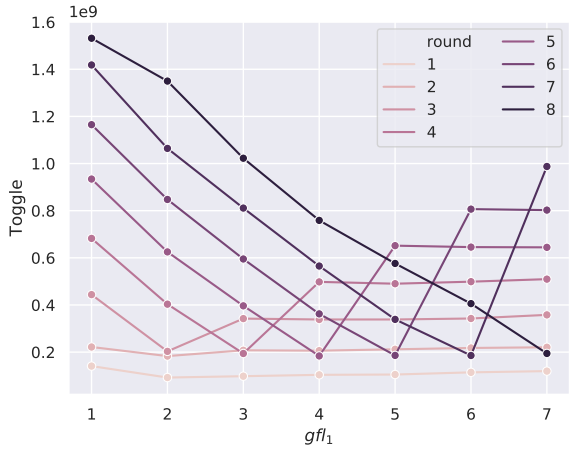
(a) AND-gate glitch filter

(b) Latch-based glitch filter

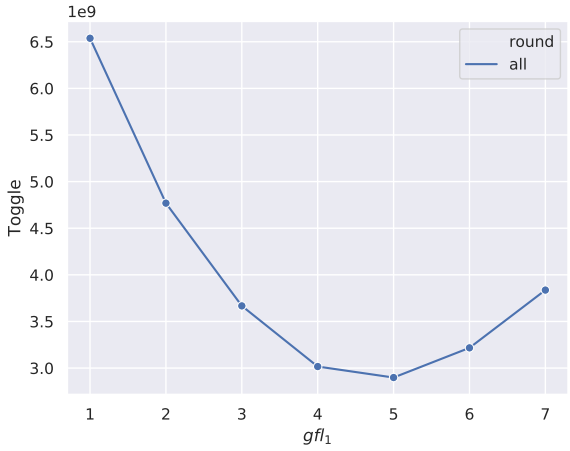
Figure 5.12: Energy of designs with one glitch filter

this is the case, we examined the toggle frequency of the msg signal at the end of each combinational round, shown in Figure 5.13. The toggle trends observed in the msg signal

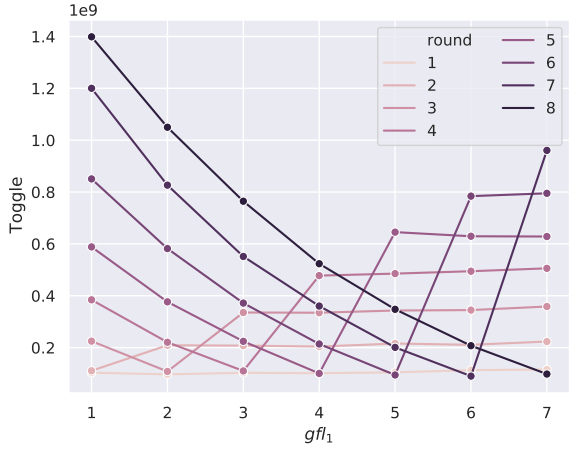
is a good estimation of the toggle trends for the whole circuit. Since energy is proportional to the switching activity, the toggle trend could also provide a good estimation of the energy trend of the entire circuit. Figure 5.13(a) illustrates the toggle activity of *msg* for every round and Figure 5.13(b) depicts the total toggles of that signal (*i.e.* the sum of the points in Figure 5.13(a)). The trend in Figure 5.13(b) closely resembles the trend in Figure 5.12(a), while the trend in Figure 5.13(c) closely resembles that in Figure 5.12(b).



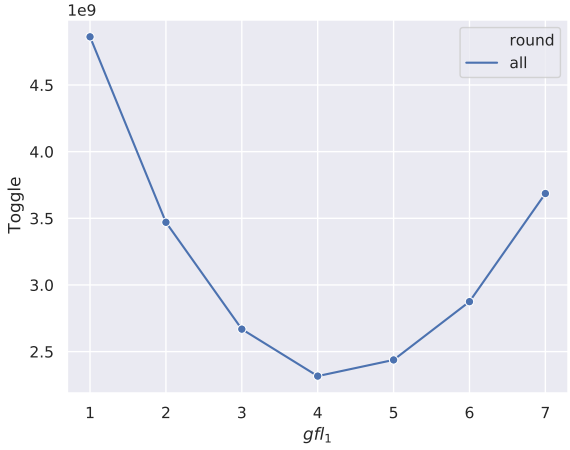
(a) Toggle per round with AND-gate glitch filter



(b) Toggle of all rounds with AND-gate glitch filter



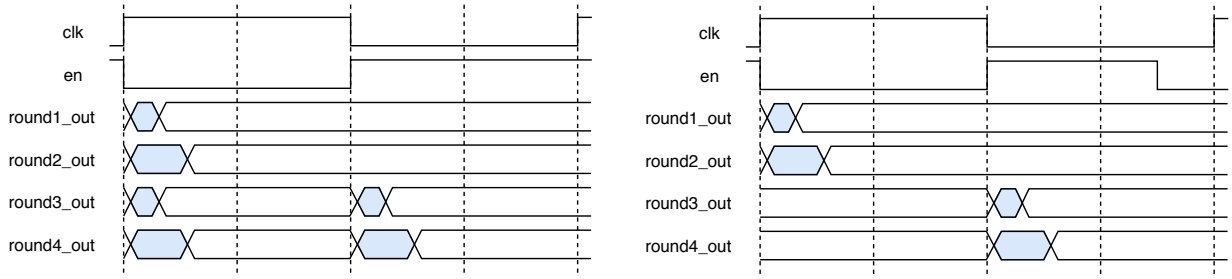
(c) Toggle per round with latch glitch filter



(d) Toggle of all rounds with latch glitch filter

Figure 5.13: Toggle activity of *msg* signal for designs with a single glitch filter

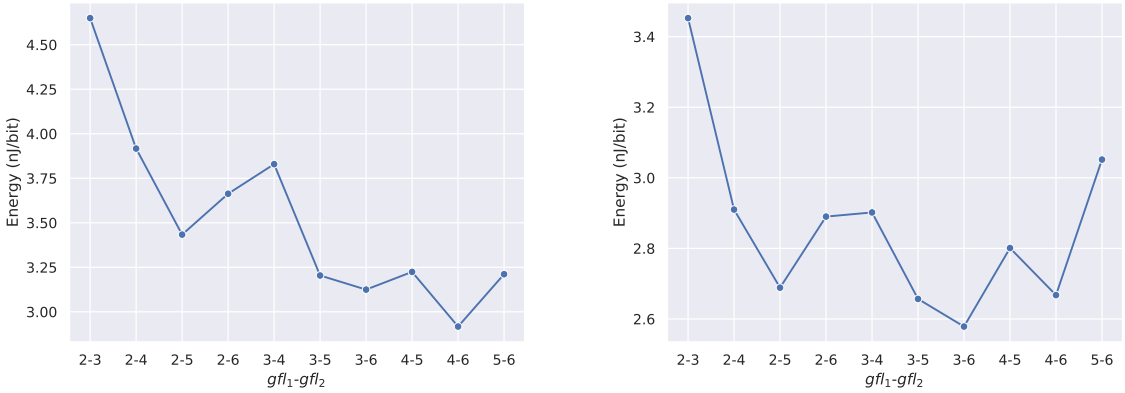
Due to the nature of the AND-gate glitch filtering scheme, there is an imbalance in the amount of switching activity in the rounds preceeding the glitch filter compared to the rounds proceeding the glitch filter. In the rounds before the glitch filter, toggles occur soon after the rising edge of the clock, with the switching activity increasing in the later rounds. The rounds proceeding the glitch filter will see glitches not only when the *enable* signal goes high but also when it goes low at the beginning of the clock cycle. Thus, rounds after the glitch filter will experience glitching on two separate occasions. Figure 5.13(a) demonstrates that the rounds after the glitch filter have higher toggle frequency. Figure 5.14(a) illustrates an example of the glitching activity in one clock cycle for $rpcc = 4$ and $gfl_1 = 2$. Because of this imbalance in switching activity, the optimal location of the single glitch filter lies not at the halfway point of the combinational rounds, but rather at some point beyond that. Unlike the AND-gate glitch filtering method, the rounds after the latch glitch filter do not toggle at two different points in time; they toggle only when *en* goes high and not when *en* goes low. Figure 5.14(b) shows an example of the toggle activity in a latch-based glitch filtering scheme for $rpcc = 4$ and $gfl_1 = 2$.



(a) AND-gate (b) Latch

Figure 5.14: Glitches (in blue) in a clock cycle for $rpcc = 4$ and $gfl_1 = 2$

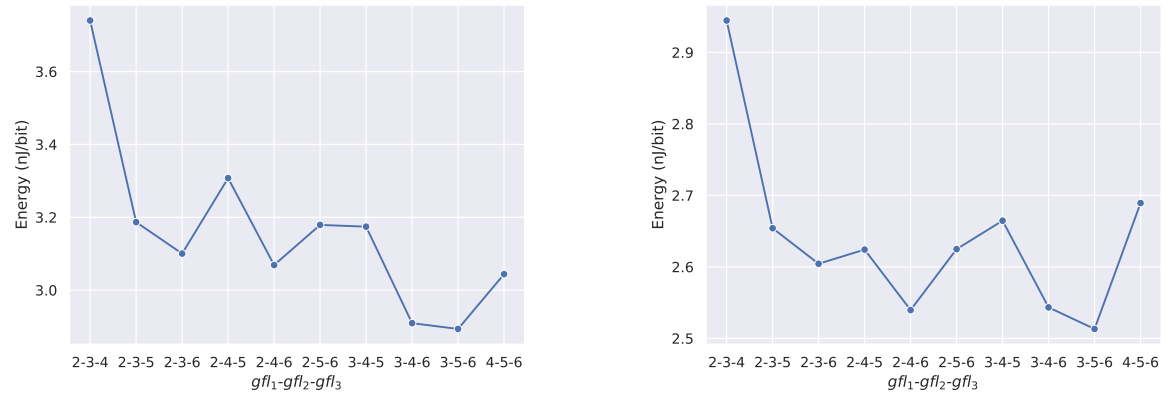
Next we will look at finding the optimal locations of two AND-gate glitch filters. With eight combination rounds, there are 7 possible places to insert glitch filters. Two locations need to be chosen so that gives $\binom{7}{2} = 21$ possible designs. To save time, a subset of all combinations is explored. Glitch filters will be inserted at only the 5 middle locations, which gives $\binom{5}{2} = 10$ possible designs. Figure 5.15(a) illustrates the energy per bit for each of the 10 designs. The most energy savings come from the design with $gfl_1 = 4$ and $gfl_2 = 6$, indicated by the global minimum. The energy for two latch-based glitch filters is shown in Figure 5.15(b). The most energy-efficient design has $gfl_1 = 3$ and $gfl_2 = 6$.



(a) AND-gate glitch filters (b) Latch glitch filters

Figure 5.15: Energy of designs with two glitch filter

With three glitch filters, there are also $\binom{5}{3} = 10$ possible combinations if we limit the locations of the glitch filters to the 5 inner rounds, similar to the investigation of two glitch filters. The energy of designs with three AND-based glitch filters appears in Figure 5.16(a). The energy for three latch-based glitch filters is shown in Figure 5.16(b). The most energy-efficient design for both the AND-gate and latch glitch filtering schemes occurs with $gfl_1 = 3$, $gfl_2 = 5$, and $gfl_3 = 6$.



(a) AND-gate glitch filters (b) Latch glitch filter

Figure 5.16: Energy of designs with three glitch filter

Table 5.3 summarizes the optimal configurations for designs having one, two, and three glitch filters. The 'glitch filter spacing' exploration demonstrated that for *rpcc*=8 and the AND-gate filtering scheme, the most energy-efficient configuration is to have glitch filters in between every round, which means having 7 glitch filters. The resulting energy is 2.66 nJ/bit. Comparable energy savings can be achieved by using only 3 optimally placed glitch filters for *rpcc*=8, which has a much lower area overhead. The 'glitch filter spacing' exploration also showed that for $rpcc = 8$ and the latch-based glitch filtering scheme, the most energy-efficient design has $gfs = 2$, which has three glitch filters. The resulting energy is 2.54 nJ/bit. By assessing different combinations of the location of glitch filters, the 'glitch filter location' exploration arrived at a better design that the 'glitch filter spacing' exploration missed; the final row in Table 5.3 lists a design using three latch-based glitch filters with an energy result of 2.51 nJ/bit.

Table 5.3: Summary of best glitch filter locations in PRESENT

Filter method	**Num. filters**	**Lowest energy design**	**Energy (nJ/bit)**
AND	1	$gfl_1 = 5$	3.45
	2	$gfl_1 = 4$, $gfl_2 = 6$	2.92
	3	$gfl_1 = 3$, $gfl_2 = 5$, $gfl_3 = 6$	2.89
LATCH	1	$gfl_1 = 4$	2.98
	2	$gfl_1 = 3$, $gfl_2 = 6$	2.58
	3	$gfl_1 = 3$, $gfl_2 = 5$, $gfl_3 = 6$	2.51

5.4.4 Glitch Filters in a Pipelined Design

In this section, we investigate how energy, area, and throughput vary under different degrees of pipelining in unrolled designs. To limit the number of variables at play, we will only consider designs in which glitch filters are applied every round. The degree of pipelining can be controlled by modifying either *ccps*, the clock cycles per stage, or S, the number of stages. The *ccps* parameter is useful because in pipelined designs, it dictates the throughput. For instance, $ccps = 2$ means a ciphertext is ready every two clock cycles. The number of stages directly relates to how much duplicate hardware is required. Having three stages, for example, means three copies of hardware are required. The parameters *rpcc*, *ccps*, and S are all interdependent and modifying one variable will affect at least one of the other two. See Section 5.1 for details of the relationship among these variables. In previous sections,

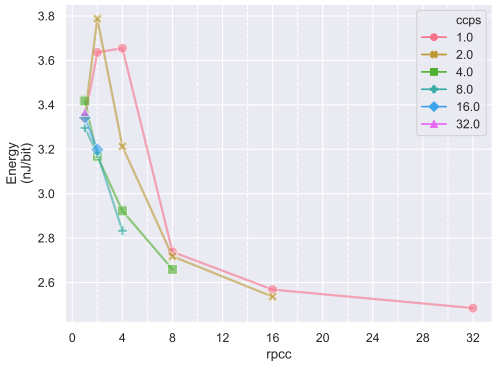
we considered round unrolling in non-pipelined designs. Although throughput can be increased by unrolling rounds (*i.e.* having more rounds per clock cycle means less clock cycle are required to complete the operation), the maximum number of unrolled rounds might be limited by the clock period and further throughput gains would have to be attained by pipelining. The purpose of this section is to explore whether for a given number of unrolled loops, the degree of pipelining affects the total energy consumption.

Figure 5.17 depicts energy, area, and optimality as a function of *rpcc* for different values of *ccps* in both the AND-gate and latch glitch filtering schemes. For a given value of *rpcc*, the energy consumption remains relatively independent of *ccps*, as depicted in Figures 5.17(a) and 5.17(b). This makes sense because pipelining does not change the datapath of the operation; it simply allows the parallel execution of multiple messages. Whether there are two or four messages being encrypted simultaneously has no impact on the amount of work (*i.e.* energy) required to encrypt one message. Area, shown in Figures 5.17(c) and 5.17(d), is affected by both *rpcc* and *ccps*. For a given *ccps*, the area decreases as *rpcc* increases because the number of stages decreases. For a given *rpcc*, the area decrease as *ccps* increases also because the number of stages decreases. The global maximum optimality values, shown in Figures 5.17(e) and 5.17(f), correspond with the global minimum energy values. Figure 5.18 portrays the same data as Figure 5.17 except the lines are grouped by stages instead of *ccps*.

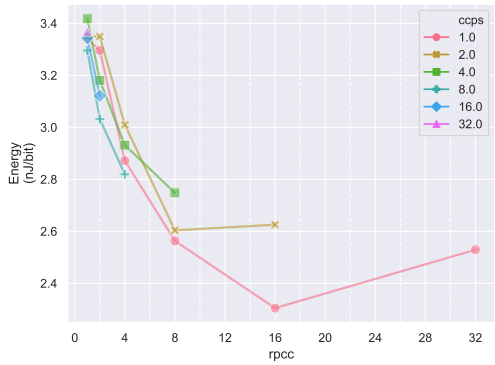
Since pipelining does not have much effect on energy consumption of the circuit (for a given *rpcc*), but does on the area and throughput, the *rpcc* and glitch filter scheme should be selected first to achieve the desired energy consumption, after which the the degree of pipelining can be selected to achieve the desired area and throughput.

5.5 Summary

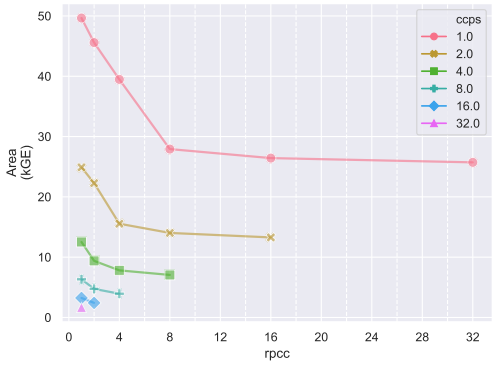
In this chapter, two glitch filtering schemes were analyzed for unrolled designs of PRESENT and AES block ciphers. One method uses AND-gates in between combinational rounds while the other used latches. Both methods rely on *enable* signals to 'activate' the rounds, allowing the propagation of signals only after they have stabilized. The energy consumption of PRESENT and AES was assessed as a function of *rpcc* (rounds per clock cycle) and gfs (glitch filter spacing). For PRESENT, two additional parameters, gfl (glitch filter location) and *ccps* (clock cycles per stage), were also investigated.



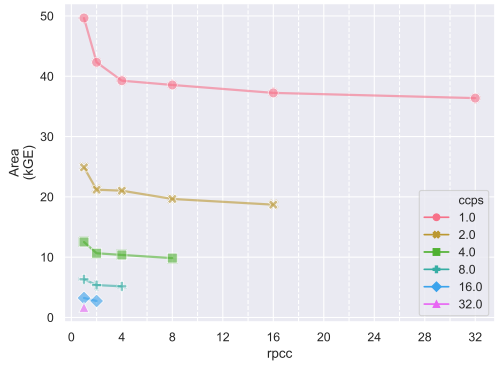
(a) AND-gate glitch filter energy



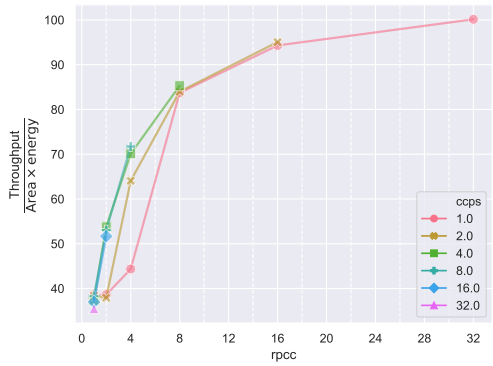
(b) Latch glitch filter energy



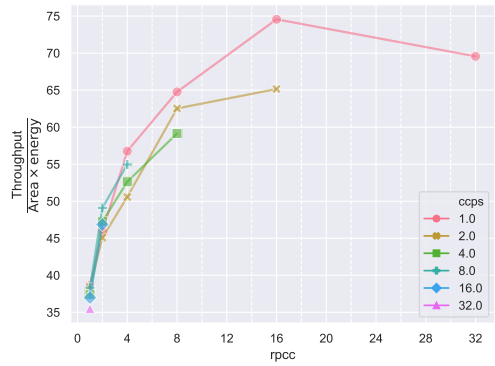
(c) AND-gate glitch filter area



(d) Latch glitch filter area

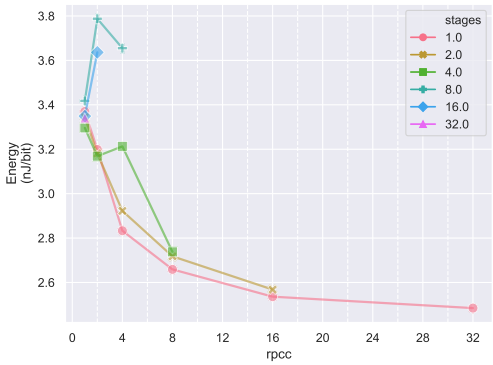


(e) AND-gate glitch filter optimality

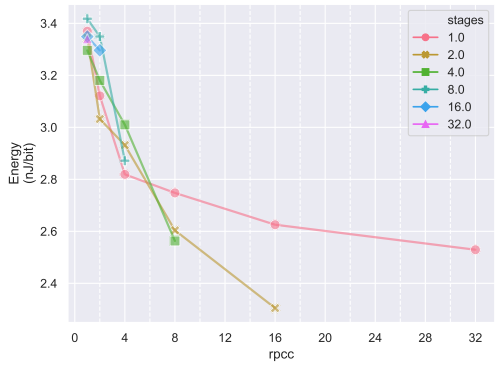


(f) Latch glitch filter optimality

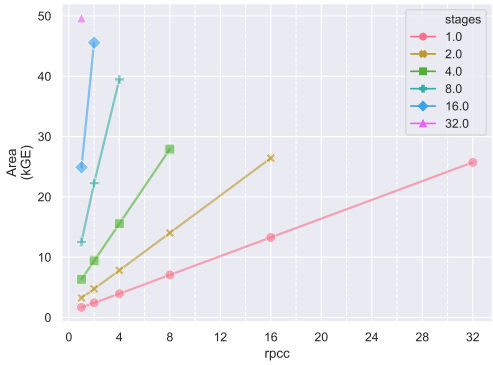
Figure 5.17: Varying degree of pipelining by *ccps*



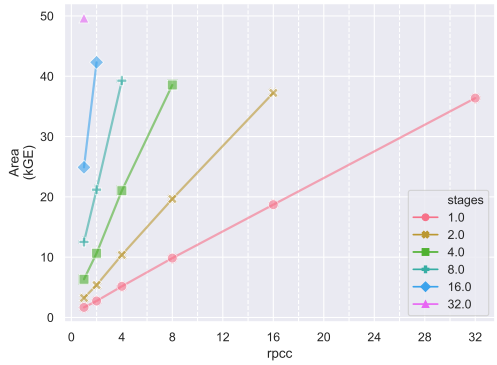
(a) AND-gate glitch filter energy



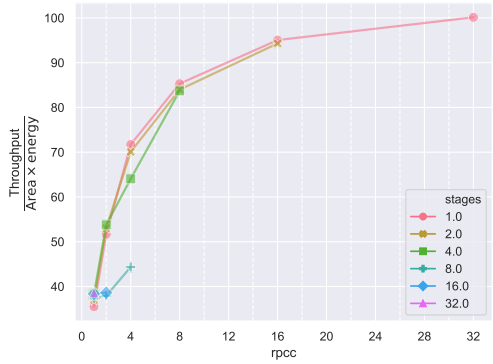
(b) Latch glitch filter energy



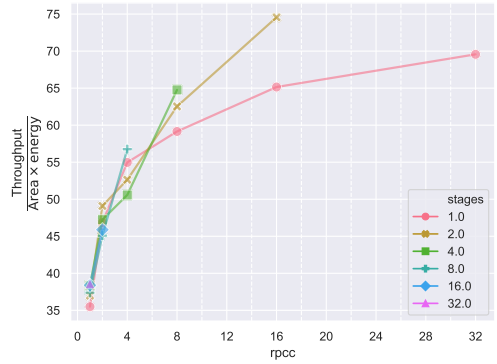
(c) AND-gate glitch filter area



(d) Latch glitch filter area



(e) AND-gate glitch filter optimality



(f) Latch glitch filter optimality

Figure 5.18: Varying degree of pipelining by stages

In general, the latch method achieved better energy savings than the AND-gate method. In PRESENT, the optimal energy savings occurred in 32 unrolled rounds with latch glitch filters spaced every two rounds. When the number of glitch filters is limited to a few, near-optimal energy savings can be obtained by placing the glitch filters at optimal locations. For $rpcc = 8$, latch glitch filters should be placed after the third, fifth, and sixth rounds. In AES, the latch glitch filters also provided the biggest energy savings. The most energy efficient design had 6 unrolled rounds with latch glitch filters after every round. Because AES is much larger, the energy overhead of the glitch filters, as a percentage, isn't as great as in PRESENT.

Chapter 6

Conclusion and Future Work

In this final chapter, the research work is summarized. Major points of this thesis are highlighted in Section 6.1 and ideas for future research work are presented in Section 6.2.

6.1 Summary

This thesis explored low-area and low-energy hardware optimizations for two block ciphers. PRESENT is a lightweight block cipher designed for low-power applications such as IoT or smart devices. AES is the most popular and widely adopted cipher, used in most smartphones and laptops and by the US government to protect highly classified information. Despite being much larger than PRESENT, AES is often still the preferred choice for embedded and resource-constrained applications because of its high level of security and reliability.

To offer AES as an option for even extremely resource-constrained devices, we provided a low-area ASIC implementation of its encryption algorithm in Chapter 4. We also implemented, analyzed, and compared three existing low-area AES encryption cores on a common technology (STMicro 65 nm). Using our taxonomy of architectural and component options for an 8-bit design, we showed how different design goals should motivate different design decisions. Our benchmark offers results in the metrics of area, energy, power, and optimality. Our architecture, Quark-AES, showed a 13% improvement in area and 8% improvement in throughput/area^2 compared to the runner-up. To illustrate the

extent of variations due to the use of different ASIC libraries, we also synthesized the designs on three additional technologies. Even for the same transistor size, different ASIC libraries produce different area results for the same RTL design. Knowing this, comparing the report area values from existing works can be unreliable and our benchmark remedies this issue. Because some applications require a more balanced tradeoff between throughput and area, we extended each architecture to higher datawidths, including 16-bit and 32-bit datawidths.

The second part of this thesis assessed two glitch filtering methods in unrolled designs of AES-128 and PRESENT. The first method used a bank of AND gates in between unrolled rounds and allowed signals to be propagated only after they had stabilized in the previous round. The output of the AND gates were controlled by *enable* signals (one for each unrolled round) which asserted in successive intervals. The second glitch filtering scheme worked in a similar fashion, but used latches in place of AND gates and whose transparent mode was controlled by the *enable* signals. In the analysis, a non-pipelined implementation with loop unrolling was first considered. When glitch filters are applied at every round in PRESENT, the baseline (no glitch filter) offered the lowest energy consumption for $rpcc \leq 4$ while the latch glitch filtering scheme provided the lowest energy consumption for $rpcc > 4$. In AES, the latch glitch filter was the best method for all values of *rpcc*. The energy consumption of each cipher was also affected by the glitch filter spacing. In PRESENT, applying latch glitch filters every other round provided the most energy savings. Applying them every round caused the energy of the glitch filter to dominate the energy it saved, while applying it less than every other round caused the glitching energy to dominate. In AES, it was most energy-efficient to apply latch glitch filters every round. When the number of glitch filters is restricted due to area requirements, significant energy savings can still be obtained by placing the available glitch filters at optimal locations.

6.2 Future Work

The work in this thesis certainly leaves room for further research. On the low-area front, this thesis only developed an encryption core of AES. Although Quark-AES still remains practical when used as part of cipher feedback, output feedback, or counter modes, it would benefit to have other options available by having a decryption core. The decryption support can be added by either building a merged encryption/decryption datapath or a separate decryption core.

While this thesis assessed architectural options for an 8-bit AES design, it didn't explore the details within a component, such as the internals of the S-box. Further research should be done to examine different composite field constructions and polynomials, such as Mathew did, to see (1) whether the architecture affects what the optimal polynomial is, and (2) whether the ASIC library affects what the optimal polynomial is. Hardware ciphers are often vulnerable to physical attacks such as side-channel attacks. It would be worth exploring a masking implementation for Quark-AES, like Moradi did for their cipher, as well as other side-channel countermeasure techniques.

The glitch filtering methods analyzed in this work were performed only for ASICs. It may be interesting to extend the analysis to FPGAs to see how they compare to the results in this thesis. If sufficient analysis is conducted on several different block ciphers, it may be possible to achieve a model that predicts the optimal number of glitch filters for a cipher depending on its size.

References

[1] Khaleel Ahmad, M. N. Doja, Nur Izura Udzir, and Manu Pratap Singh. *Emerging Security Algorithms and Techniques*. CRC Press, 2019.

[2] Kate Kochetkova. Shock at the wheel: your jeep can be hacked while driving down the road. *Kaspersky Lab*.

[3] Alex Drozhzhin. Black Hat USA 2015: The full story of how that Jeep was hacked. *Kaspersky Lab*.

[4] The 5 worst examples of IoT hacking and vulnerabilities in recorded history. *IoT For All*.

[5] Jessica Bianchi. 5 examples of innovative uses for RFID technology in retail. *Shopify Blogs*.

[6] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. AES implementation on a grain of sand. *IEEE Proceedings - Information Security*, 152(1):13–20, Oct 2005.

[7] P. Hämäläinen, T. Alho, M. Hännikäinen, and T. D. Hämäläinen. Design and implementation of low-area and low-power aes encryption hardware core. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*, pages 577–583, Aug 2006.

[8] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88, 2011.

[9] S. Mathew, S. Satpathy, V. Suresh, H. Kaul, M. Anders, G. Chen, A. Agarwal, S. Hsu, and R. Krishnamurthy. 340mv–1.1v, 289gbps/w, 2090-gate NanoAES hardware accelerator with area-optimized encrypt/decrypt $GF(2^4)^2$ polynomials in 22nm tri-gate cmos. In *2014 Symposium on VLSI Circuits Digest of Technical Papers*, pages 1–2, June 2014.

[10] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES: A compact implementation of the AES encryption/decryption core. volume 10095, pages 173–190, 12 2016.

[11] T. Järvinen, P. Salmela, P. Hämäläinen, and J. Takala. Efficient byte permutation realizations for compact aes implementations. In *2005 13th European Signal Processing Conference*, pages 1–4, Sep. 2005.

[12] Eslam Gamal Ahmed, Eman Shaaban, and Mohamed Hashem. Lightweight Mix Columns implementation for AES. In *Proceedings of the 11th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering*, MMACTEE'09, pages 48–53, 2009.

[13] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Exploring energy efficiency of lightweight block ciphers. In *Revised Selected Papers of the 22Nd International Conference on Selected Areas in Cryptography - SAC 2015 - Volume 9566*, pages 178–194, Berlin, Heidelberg, 2016. Springer-Verlag.

[14] S. Banik, A. Bogdanov, F. Regazzoni, T. Isobe, H. Hiwatari, and T. Akishita. Round gating for low energy block ciphers. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 55–60, May 2016.

[15] Siva Nishok Dhanuskodi and Daniel Holcomb. Energy optimization of unrolled block ciphers using combinational checkpointing. *IACR Cryptology ePrint Archive*, 2016:1093, 2016.

[16] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[17] Jean-Marc Couveignes and Reynald Lercier. Fast construction of irreducible polynomials over finite fields. 2009.

[18] R.W. Marsh. *Table of Irreducible Polynomials Over GF(2) Through Degree 19*. U.S. Department of Commerce, Office of Technical Services, 1957.

[19] John Kerl. Computation in finite fields, 2004.

[20] Ming-Deh Huang and Anand Narayanan. Finding primitive elements in finite fields of small characteristic. 04 2013.

[21] Gary L. Mullen and Daniel Panario. *Handbook of Finite Fields*. Chapman & Hall/CRC, 1st edition, 2013.

[22] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, USA, 1996.

[23] Roberto Maria Avanzi. A salad of block ciphers. *IACR Cryptology ePrint Archive*, 2016:1171, 2016.

[24] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, Oct 1949.

[25] Morris J. Dworkin. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, Gaithersburg, MD, United States, December 2001.

[26] National Institute of Standards and Technology (NIST). FIPS PUB 197: Advanced Encryption Standard (AES), November 2001.

[27] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[28] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, pages 239–254, 2001.

[29] D. Canright. A very compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 441–455, 2005.

[30] M. S. Wamser and G. Sigl. Pushing the limits further: Sub-atomic AES. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, Oct 2017.

[31] N. K. Dumpala, S. B. Patil, D. Holcomb, and R. Tessier. Energy efficient loop unrolling for low-cost fpgas. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 117–120, April 2017.

[32] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient rijndael encryption implementation with composite

field arithmetic. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 171–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[33] B. Sunar, E. Savas, and C. K. Koc. Constructing composite field representations for efficient conversion. *IEEE Transactions on Computers*, 52(11):1391–1398, Nov 2003.

[34] X. Zhang and K. K. Parhi. On the optimum constructions of composite field for the AES algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 53(10):1153–1157, Oct 2006.

[35] Christof Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*. PhD thesis, University of Duisburg-Essen, 1994.

[36] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. Smashing the implementation records of AES S-box. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):298–336, May 2018.

[37] Michael Muehlberghuber. AES Canright S-box. https://github.com/mbgh/aes128-hdl/blob/master/src/vhdl/sboxCan.vhd, 2014.

[38] Rei Ueno, Naofumi Homma, Yukihiro Sugawara, Yasuyuki Nogami, and Takafumi Aoki. Highly efficient $GF(2^8)$ inversion circuit based on redundant gf arithmetic and its application to aes design. In Tim Guneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 63–80, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[39] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.10.2*, 2019.

[40] Thomas W. Cusick and Pantelimon Stanica. Chapter 2 - fourier analysis of boolean functions. In Thomas W. Cusick and Pantelimon Stanica, editors, *Cryptographic Boolean Functions and Applications (Second Edition)*, pages 7 – 29. Academic Press, second edition edition, 2017.

[41] Lidong Chen and Guang Gong. *Communication System Security*. Chapman & Hall/CRC, 1st edition, 2012.

[42] A. Ajane, P. M. Furth, E. E. Johnson, and R. L. Subramanyam. Comparison of binary and LFSR counters and efficient LFSR decoding algorithm. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, Aug 2011.

[43] G. B. Fitzpatrick. Synthesis of binary ring counters of given periods. *J. ACM*, 7(3):287–297, July 1960.

[44] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[45] Maria George and Peter Alfke. *Linear Feedback Shift Registers in Virtex Devices*. Xilinx, April 2007.

[46] D. W. Clark and Lih-Jyh Weng. Maximal and near-maximal shift register sequences: efficient event counters and easy discrete logarithms. *IEEE Transactions on Computers*, 43(5):560–568, May 1994.

[47] Jaromir Kolouch. LFSR counters with arbitrary cycle length. 11 2019.

[48] Jean-Pierre Deschamps, Jose Luis Imana, and Gustavo D. Sutter. *Hardware Implementation of Finite-Field Arithmetic*. The McGraw-Hill Companies, 2009.

APPENDICES

Appendix A

Finding a Transformation Matrix Between Two Fields

A.1 Method

A finite field $GF(2^n)$ can be regarded as an n-dimensional vector space over $GF(2)$. Every vector space has a basis; that is, a set of elements that are linearly independent. Every element in the vector space can be represented as a linear combination of the basis vectors. A vector space can have more than one bases.

Typically, finite fields are represented in one of three bases: standard (or canonical or polynomial), normal, and dual [35, Chapter 2.1.3]. In this section, we are only concerned with the polynomial basis, whose definition is given below:

Definition A.1.1 (Polynomial Basis) *If α is a root of the irreducible polynomial $P(x)$ of degree n, then the set $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$ forms a polynomial basis for the field $GF(q^n)$.*

This representation is equivalent to the representation presented in Section 2.1. That is, the polynomial representation $A(x) = a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0$ is equivalent to $A(\alpha) = a_{n-1}\alpha^{n-1} + \cdots + a_2\alpha^2 + a_1\alpha + a_0$.

Because a composite representation has different defining polynomials, it will also have a different basis. The purpose of a transformation matrix is to perform a change of basis; that is, to perform an isomorphic mapping of field elements represented with respect to the binary field to elements represented with respect to the composite field. To obtain the representation of an element, E_α, in basis α, perform matrix-vector multiplication of the transformation matrix, M, with the representation of element E_β in basis β: $E_\alpha = M \cdot E_\beta$.

Paar's algorithm for finding an isomorphic mapping applies only to irreducible polynomials that are primitive [35, Chapter 2.2]. The AES polynomial is not primitive, thus his algorithm cannot be applied. Zhang [34] proposed an algorithm that works for polynomials that are non-primitive. If α is a root of the irreducible polynomial $P(x) = x^8+x^4+x^3+x+1$ used in AES, then the set $\{1, \alpha, \alpha^2..., \alpha^7\}$ forms a polynomial basis for the field [34]. The basic idea is to find eight base elements, $1, \beta, \beta^2, \ldots, \beta^7$ of the composite field that map to the base elements $1, \alpha, \alpha^2, \ldots, \alpha^7$ of the binary field representation. The transformation matrix is constructed by taking the binary representation of β^j as the j^{th} column entry. A simplified version of Zhang's algorithm is the following. Iterate over all the elements in the composite field and check if the current element ω satisfies $P(\omega) = 0$, where the computation is carried out according to the arithmetic rules of the composite field. If the condition is satisfied, then a base element β is found. Zhang's algorithm is a more efficient version of this as it removes elements of the same conjugacy class as ω, if ω doesn't satisfy $P(\omega) = 0$. Their algorithm requires $(2^q - 1)/2q$ checkings, on average, for a field of order 2^q.

For a given field construction, there are eight possible isomorphic mappings that map the elements in $GF(2^8)$ to elements in the composite field [34], each one differing in hardware complexity. In Appendix A.2, the default bases (that GAP employs interally) is used. GAP (Groups, Algorithms, and Programming) is a free software package for computation in discrete abstract algebra [39]. Particularly useful for this section are the functions provided on Galois fields [39, Chapter 59].

A.2 GAP Code

```
toBinList := function(x)
    local i, str;
    str := [];

    for i in x do
```

```
        if i = 0*Z(2) then
            Add(str, 0);
        else
            Add(str, 1);
        fi;
    od;

    return str;
end;


# Create indeterminate
x := X(GF(2), "x");

# Construct AES Galois Field
aes_gf := GF(GF(2), x^8 + x^4 + x^3 + x + 1);

# Construct Mathew's ground field
ground := GF(GF(2), x^4 + x^3 + 1);
y := X(ground, "y");

alpha := LinearCombination(Basis(ground), [0,1,1,0]);
beta := LinearCombination(Basis(ground), [1,0,0,1]);

# Construct Mathew's extension field
extension := GF(ground, y^2 + alpha * y + beta);

# Find transformation matrix using the basis vectors

aes_bas := BasisVectors(Basis(aes_gf));
M := [];

for i in aes_bas do

    i_ext := Coefficients(Basis(extension),i);
    app := [];
    for j in i_ext do
        c := Coefficients(Basis(ground), j);
        Append(app, c);
    od;
```

```
    tmp := toBinList(app);
    Add(M, tmp);
od;

M:= TransposedMat(M);
Minv := Inverse(M) mod 2;
```

A.2.1 Output Matrices

The transformation matrix and its inverse produced by the GAP code are

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad M^{-1} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

where the least significant bit is in the upper left corner.

Appendix B

LFSR Counters

A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous states [40]. LFSRs are often used for generating pseudorandom sequences for applications like communicaton systems, cryptographic algorithms, and hardware design [41, Chapter 2.1]. They are also commonly used as event counters, serving as an alternative to binary counters. While binary counters generally use registers, half adders and a high-speed carry chain, LFSR counters typically require only a few XOR or XNOR gates in addition to the registers [42]. Moreoever, the delay in a binary counter is related to the number of bits, whereas the delay in LFSRs is independent of number of bits. Because of its low area and delay, LFSRs are often an appealing choice for a counter.

An LFSR with m registers has a *degree* of m and can generate at most $2^m - 1$ unique states, where the state refers to the sequence of bits contained in the registers. The *period* of an LFSR is the number of shifts that happen before the initial state reoccurs. The initial state of the LFSR is known as the *seed*.

An LFSR is defined by a *characteristic polynomial*, which has a degree of m. If we let the registers represent all but the highest degree terms in the characteristic polynomial, then a register representing a term with a coefficient of '1', known as a *tap*, has an XOR gate between it and the previous register, illustrated in Figure B.1. Taps are bit positions that are involved in the feedback function and that affect the next state. They correspond to the terms in the characteristic polynomial with a coefficient of '1'. The only exception is the register representing x^0: there is no XOR gate associated with it even when the coefficient is '1'. The register shifts in the direction of increasing exponent.

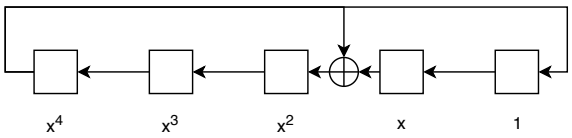


Figure B.1: LFSR with characteristic polynomial $x^5 + x^2 + 1$

The period of an LFSR depends on the characteristic polynomial. Furthermore, more than one possible period may be obtained depending on the initial state of the counter [43]. If the characteristic polynomial is *primitive*, then the LFSR will cycle through all $2^n - 1$ non-zero states (for an XOR feedback LFSR) and the LFSR is known as a *maximum-length* LFSR [44].

LFSRs can have two types of feedback: XOR feedback or XNOR feedback. A state with all zeroes is illegal when using XOR and a state with all ones is illegal when using an XNOR feedback. These states are considered illegal because the counter would remain stuck in the state [45]. A table of tap positions to achieve maximal-length XNOR-feedback LFSR counters for degrees of 3 through 168 can be found in [45].

The LFSR configuration shown in Figure B.1 is called an *internal* or *Galois* LFSR. In such LFSRs, the taps are XORed with the output bit before being stored in the next position and the other bit positions are shifted as a regular shift register. Another type of LFSR, called an *external* or *Fibonacci* LFSR, has the feedback function as a serial chain of XOR or XNOR gates, and only the input bit is a function of the feedback function. All remaining registers operate as a regular shift register. An external LFSR with the same characteristic polynomial as in Figure B.1 is shown in Figure B.2. While an external LFSR might have many XOR gates connected in serial, an internal LFSR has only one XOR gate between registers and therefore internal LFSRs usually have lower delays.

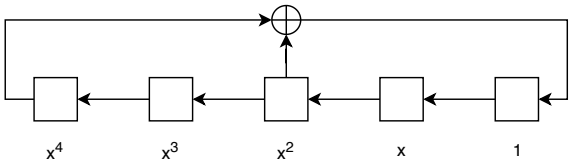


Figure B.2: External LFSR with characteristic polynomial $x^5 + x^2 + 1$

For Galois LFSRs, the state is commonly treated as the binary coefficients of a polynomial and a register shift may be thought of as a polynomial multiplication by x, modulo the shift register's characteristic polynomial [46]. For example, the state 11001 represents

$x^4 + x^3 + 1$. Using the LFSR in Figure B.1, one shift of the state 11001 is equivalent to the multiplication $(x^4 + x^3 + 1)x = x^5 + x^4 + x \mod (x^5 + x^2 + 1) = x^4 + x^2 + x + 1$ and the resulting state is 10111.

Because most applications require maximal-length LFSRs, there has not been much research surrounding the generation of LFSRs with a specific period. Fitzpatrick [43] and Kolouch [47] both explored this problem. Having an LFSR of a specific period is desirable because it can simplify the control logic of a design. If the desired period cannot be obtained, it is always possible to generate a counter of a specific cycle using a maximal-length LFSR: choose the smallest (*i.e.* least number of bits) maximal-length LFSR that has a period greater than the desired maximum count value, n. Pick a seed. Shift the LFSR n times and note the resulting state, s. Assert a control signal when the LFSR state equals s.

Given a state, finding the number of shifts that has occurred since the initial state is called the *discrete logarithm problem*. It will not be discussed here, but additional information may be found in [46], [42].

Appendix C

Mathew Composite Field Derivations

The bases used in this thesis to implement Mathew's composite field are listed in Section C.1. Section C.2 and Section C.3 provide the hand calculations for functions in the S-box and MixColumns operations.

C.1 Notations and Constructions

Mathew's [9] composite field defining polynomials:

$f(x) = x^4 + x^3 + 1$

$g(x) = x^2 + \alpha x + \beta, \quad \alpha, \beta \in \mathbb{F}_{2^4}$

Let $f(\omega) = 0$ and $g(\lambda) = 0$. The polynomial bases (PB) of the composite field are the following:

$\text{PB}_{\mathbb{F}_{2^4}/\mathbb{F}_2} = \{\omega^3, \omega^2, \omega, 1\}$

$PB_{\mathbb{F}_{(2^4)^2}/\mathbb{F}_{2^4}} = \{\lambda, 1\}$

$PB_{\mathbb{F}_{(2^4)^2}/\mathbb{F}_2} = \{\lambda\omega^3, \lambda\omega^2, \lambda\omega, \lambda, \omega^3, \omega^2, \omega, 1\}$

Reductions:

$$\begin{aligned}
\omega^4 &= \omega^3 + 1 \\
\omega^5 &= \omega^3 + \omega + 1 \\
\omega^6 &= \omega^3 + \omega^2 + \omega + 1 \\
\omega^7 &= \omega^2 + \omega + 1 \\
\omega^8 &= \omega^3 + \omega^2 + \omega \\
\omega^9 &= \omega^2 + 1
\end{aligned}$$

$$\begin{aligned}
\lambda\omega^4 &= \lambda\omega^3 + \lambda \\
\lambda\omega^5 &= \lambda\omega^3 + \lambda\omega + \lambda \\
\lambda\omega^6 &= \lambda\omega^3 + \lambda\omega^2 + \lambda\omega + \lambda \\
\lambda^2 &= \lambda\omega^2 + \lambda\omega + \omega^3 + 1 \\
\lambda^2\omega &= \lambda\omega^3 + \lambda\omega^2 + \omega^3 + \omega + 1 \\
\lambda^2\omega^2 &= \lambda + \omega^3 + \omega^2 + \omega + 1 \\
\lambda^2\omega^3 &= \lambda\omega + \omega^2 + \omega + 1 \\
\lambda^2\omega^4 &= \lambda\omega^2 + \omega^3 + \omega^2 + \omega \\
\lambda^2\omega^5 &= \lambda\omega^3 + \omega^3 + 1 \\
\lambda^2\omega^6 &= \lambda\omega^3 + \lambda + \omega^3 + \omega
\end{aligned}$$

C.2 S-box

Let $s(\omega) \in \mathbb{F}_{2^4}$, $\alpha = \omega^2 + \omega$, $\beta = \omega^3 + 1$.

Multiplication by constant α:

$$\begin{aligned}
s(\omega) \cdot \alpha &= (a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0) \cdot (\omega^2 + \omega) \\
&= a_3\omega^5 + a_2\omega^4 + a_1\omega^3 + a_0\omega^2 + a_3\omega^4 + a_2\omega^3 + a_1\omega^2 + a_0\omega \\
&= a_3(\omega^3 + \omega + 1) + a_2(\omega^3 + 1) + a_1\omega^3 + a_0\omega^2 + a_3(\omega^3 + 1) + a_2\omega^3 + a_1\omega^2 + a_0\omega \\
&= \omega^3(a_1) + \omega^2(a_1 + a_0) + \omega(a_3 + a_0) + (a_2)
\end{aligned}$$

Multiplication by constant β:

$$\begin{aligned}
s(\omega) \cdot \beta &= (a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0) \cdot (\omega^3 + 1) \\
&= a_3\omega^6 + a_2\omega^5 + a_1\omega^4 + a_0\omega^3 + a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0 \\
&= a_3(\omega^3 + \omega^2 + \omega + 1) + a_2(\omega^3 + \omega + 1) + a_1(\omega^3 + 1) + a_0\omega^3 + a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0 \\
&= \omega^3(a_2 + a_1 + a_0) + \omega^2(a_3 + a_2) + \omega(a_3 + a_2 + a_1) + (a_3 + a_2 + a_1 + a_0)
\end{aligned}$$

Square and multiplication with constant β:

$$\begin{aligned}
s(\omega)^2 \cdot \beta &= (a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0)^2 \cdot (\omega^3 + 1) \\
&= a_3\omega^6 + a_2\omega^4 + a_1\omega^2 + a_0) \cdot (\omega^3 + 1) \\
&= a_3\omega^9 + a^2\omega^7 + a_1\omega^5 + a_0\omega^3 + a_3\omega^6 + a_2\omega^4 + a_1\omega^2 + a_0 \\
&= a_3(\omega^2 + 1) + a_2(\omega^2 + \omega + 1) + a_1(\omega^3 + \omega + 1) + a_0\omega^3 \\
&+ a_3(\omega^3 + \omega^2 + \omega + 1) + a_2(\omega^3 + 1) + a_1\omega^2 + a_0 \\
&= \omega^3(a_1 + a_2 + a_1 + a_0) + \omega^2(a_2 + a_1) + \omega(a_3 + a_2 + a_1) + (a_1 + a_0)
\end{aligned}$$

Multiplication in $GF(2^4)$: classic two-step [48, Chapter 7.1.1].

Affine transformation:

Original: $Y(x) = Ax + B$

New: $Y'(x) = M \cdot A \cdot M^{-1} \cdot x + M \cdot B$

C.3 MixColumns

Constants $0x02$ and $0x03$ in AES original basis become $0x6E$ and $0x6F$ in the composite field basis.

Multiplication by constant $6E$:

$$\begin{aligned}
6E \cdot s(\omega, \lambda) &= (\lambda\omega^2 + \lambda\omega + \omega^3 + \omega^2 + \omega)(a_7\lambda\omega^3 + a_6\lambda\omega^2 + a_5\lambda\omega + a_4\lambda + a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0) \\
&= \lambda\omega^3(a_6 + a_1) \\
&\quad + \lambda\omega^2(a_6 + a_5 + a_1 + a_0) \\
&\quad + \lambda\omega(a_5 + a_4 + a_3 + a_0) \\
&\quad + \lambda(a_7 + a_4 + a_2) \\
&\quad + \omega^3(a_7 + a_6 + a_5 + a_3 + a_2 + a_0) \\
&\quad + \omega^2(a_4 + a_3 + a_1 + a_0) \\
&\quad + \omega(a_7 + a_2 + a_0) \\
&\quad + (a_7 + a_6 + a_3 + a_1)
\end{aligned}$$

Multiplication by constant $6F$:

$$
\begin{aligned}
6F \cdot s(\omega, \lambda) &= (\lambda\omega^2 + \lambda\omega + \omega^3 + \omega^2 + \omega + 1)(a_7\lambda\omega^3 + a_6\lambda\omega^2 + a_5\lambda\omega + a_4\lambda + a_3\omega^3 + a_2\omega^2 + a_1\omega + a_0) \\
&= \lambda\omega^3(a_7 + a_6 + a_1) \\
&\quad + \lambda\omega^2(a_5 + a_1 + a_0) \\
&\quad + \lambda\omega(a_4 + a_3 + a_0) \\
&\quad + \lambda(a_7 + a_2) \\
&\quad + \omega^3(a_7 + a_6 + a_5 + a_2 + a_0) \\
&\quad + \omega_2(a_4 + a_3 + a_2 + a_1 + a_0) \\
&\quad + \omega(a_7 + a_2 + a_1 + a_0) \\
&\quad + (a_7 + a_6 + a_3 + a_1 + a_0)
\end{aligned}
$$