LOW POWER SECURE SOC FOR IOT DEVICES USING LIGHTWEIGHT CRYPTOGRAPHY ACCELERATION

by

Hikmet Seha Öztürk

B.S., Electronics and Communication Engineering, Istanbul Technical University, 2019

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

Graduate Program in Electrical & Electronics Engineering
Boğaziçi University

2022

ACKNOWLEDGEMENTS

I would like to present my thanks to Assistant Professor Faik Başkaya for his motivation and guidance during the course of this thesis.

I want to thank my colleagues from TUTEL for their friendship. Working amongst them rekindled my passion for research and allowed me to obtain the skills that make this dissertation possible.

Finally, I would like to express my gratitude to my family for their unconditional support.

ABSTRACT

LOW POWER SECURE SOC FOR IOT DEVICES USING LIGHTWEIGHT CRYPTOGRAPHY ACCELERATION

In recent years, the proliferation of the Internet of Things (IoT) has led to a major increase in the quantity and type of devices involved in digital communications. Various Lightweight Cryptography (LWC) algorithms have been proposed to answer the need of cryptography in constrained devices. Although using separate algorithms for products with varying capacities is advantageous for optimization, it creates the risk that a single product may need to support multiple cryptographic primitives.

This thesis aims to find an efficient way of providing hardware acceleration for multiple cryptography algorithms in lightweight System-on-Chips (SoC). For this purpose, we present a design methodology that identifies the common portions across LWC algorithms and uses them to increase shared resources in the hardware. We explore two approaches to accelerator design: A fully-hardware approach and a hardware-software approach. Our observations indicate that the second approach, which employs an accelerator with a custom ISA, is more effective when designing for versatility.

We leverage the open-source PicoRV32 processor to construct a lightweight SoC which employs various accelerators supporting Ascon, TinyJAMBU, and PHOTON-Beetle LWC algorithms. To enable multi-algorithm support, we utilize hardware multiplexing of unshared resources, as well as Dynamic Partial Self-Reconfiguration (DPSR) on FPGA. These implementations are compared with each other and with dedicated accelerators in terms of energy efficiency, area, and throughput. The associated tradeoffs and the conditions in which each variant is useful are determined.

ÖZET

NESNELERİN İNTERNETİ İÇİN HAFİF KRİPTOGRAFİ HIZLANDIRICILI DÜŞÜK GÜÇ TÜKETİMLİ YONGA ÜSTÜ SİSTEM TASARIMI

Son yıllarda yaygınlaşan Nesnelerin İnterneti (IoT), sayısal haberleşmeye dahil olan aygıtların sayısında ve çeşitliliğinde önemli miktarda artışa sebep olmuştur. Özellikle güç ve donanım kabiliyetleri sınırlı cihazların haberleşme esnasında kriptografiye ihtiyaç duyması, son yıllarda birçok Hafif Kriptografi (LWC) algoritmaları önerilmesine yol açmıştır. Farklı kapasiteye sahip ürünler için farklı algoritmaların kullanılması her ne kadar optimizasyon için yararlı olsa da, sahadaki bir ürünün birden fazla kriptografi algoritması kullanmak durumunda kalması ihtimalini doğurmaktadır.

Bu tezin amacı, düşük güç tüketimli Yonga Üstü Sistemlerde (SoC) birden çok kriptografi algoritması destekleyecek donanım hızlandırıcıların tasarımını araştırmaktır. Ana fikir, farklı LWC algoritmalarının arasındaki benzerlikleri tespit ederek bu kısımların aynı donanımda gerçeklenmesini sağlamaktır. Hızlandırıcıların tasarımında tamamen donanıma dayalı ve donanım-yazılım işbirliği olacak şekilde iki yaklaşım denenmiş, hibrit yöntemin esnekliğe dayalı tasarımlar için daha uygun olduğu gözlemlenmiştir.

Açık kaynak kodlu PicoRV32 işlemcisi kullanılarak yaptığımız SoC tasarımı üzerinde Ascon, TinyJAMBU ve Photon-Beetle olmak üzere üç algoritma destekleyebilen farklı hızlandırıcı tasarımları enerji verimliliği, donanım alanı kullanımı ve hız açısından karşılaştırılmıştır. Çoklu algoritma desteği için donanımsal çoklama ve FPGA üzerinde Dinamik Kısmi Yeniden Kendini Belirleme (DPSR) yöntemleri kullanılmış, bu yöntemler birbiriyle ve tekil hızlandırıcılarla karşılaştırılarak her birinin faydaları belirlenmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\odot$	Multiplication in Galois Field
$\oplus$	Bitwise XOR operation
$\|\|$	Concatenation of bitstrings
$\ggg$	Bitwise right rotation

LIST OF ACRONYMS/ABBREVIATIONS

AD	Associated Data
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CBC	Cipher Block Chaining Mode
DES	Data Encryption Standard
DFX	Dynamic Function eXchange
ECB	Electronic Codebook Mode
FPGA	Field Programmable Gate Array
FF	Flip-Flop
GF	Galois Field
GUI	Graphical User Interface
ICAP	Internal Configuration Access Port
ISA	Instruction Set Architecture
I/O	Input / Output
IoT	Internet of Things
IV	Initialization Vector
KAT	Known Answer Test
LFSR	Linear Feedback Shift Register
LUT	Look-up Table
NIST	National Institute of Standards and Technology
NLFSR	Nonlinear Feedback Shift Register
PnR	Place and Route
PRR	Partially-Reconfigurable Region
RAM	Random Access Memory
ROM	Read Only Memory

RTL	Register Transfer Level
S-box	Substitution Box
SAIF	Switching Activity Interchange Format
SHA	Secure Hash Algorithm
SPN	Substitution-Permutation Network
SoC	System-on-Chip

1. INTRODUCTION

The volume of communication being carried out worldwide has increased drastically with the globalization brought about by technology. While this increasing flow of information has proved convenient and beneficial in multiple aspects of our lives, the necessity to safeguard data has never been greater. Consequently, communication security is no longer a concept limited to critical applications in the modern world. It now constitutes a substantial concern in our daily lives, whether it is about securing digital transactions in commerce, identity verification on online services, or preserving the privacy of personal information on social media.

In the past decade, especially with the proliferation of the Internet of Things (IoT), communication across diverse agents has grown increasingly prevalent. Constrained devices now play an active role in digital communication, and a significant number of them are not in a position to support standard cryptography primitives or unwilling to trade off performance for security. Lightweight cryptography (LWC) has emerged as a field to make cryptography accessible to all devices. The development and integration of lightweight cryptography primitives has become an active area of research in both industry and academia.

The National Institute of Standards and Technology (NIST) recognized the lack of standardization in this area and started the LWC project in 2013, followed by a call for algorithms for standardization in 2018. The process is an open-source round-based competition whose final stage is ongoing as of August 2022. Due to the extensive application range covered by LWC, it is expected that there will be multiple remaining algorithms at the end of the competition, approved to be used under varying conditions.

In this thesis, we design a low-power System-on-Chip (SoC), using hardware acceleration of LWC algorithms. We investigate whether the hardware acceleration can be used effectively in constrained and IoT devices, even though there could be multiple

LWC algorithms in use in the environment. In particular, we explore the feasibility and tradeoffs associated with providing hardware support for multiple algorithms in a single SoC. Instead of having a separate hardware accelerator for each algorithm, we take the similarities in operation flow for different LWC algorithms as a starting point and devise a design methodology that would allow us to accelerate multiple LWC algorithms on a single hardware. Our implementations focus on three of the NIST LWC finalists to provide a proof of concept, but the design methodology can be generalized to other LWC algorithms in the literature as well. We target Field-Programmable Gate Arrays (FPGA) for this thesis, which has a wide adaptation in the industry due to its configurability. Our designs aim to exploit partial reconfiguration in FPGAs, a technique that can be used to reduce the power consumption and implementation area.

The contributions of this thesis can be summarized as follows:

- We construct and present two generic Loosely-Coupled Accelerators (LCA) that can accelerate different LWC algorithms by utilizing partial reconfiguration. Their implementations for three of the NIST LWC Competition finalists: TinyJAMBU, Ascon, and PHOTON-Beetle, are presented and evaluated.
- We compare two different approaches used during the design of these accelerators: hardware-controlled and hardware-software codesign. Their compatibility to support multiple algorithms, as well as variants of a single algorithm, is discussed.
- We explore the feasibility of accelerating multiple cryptography algorithms in a lightweight SoC. The tradeoffs associated with having more than one hardware for different algorithms or having a single reconfigurable accelerator is elaborated, and the conditions in which using the partial reconfiguration would be beneficial are determined.

The thesis is constructed as follows. Chapter 2 provides background for the concepts and topics referred to in this thesis. The design methodology for the proposed accelerators and the SoC is described in Chapter 3, accompanied by a detailed expla-

nation of their operation. The design implementation and the Vivado flow to allow partial reconfiguration is explained in Chapter 4. Chapter 5 evaluates and compares the performances of various versions of the design, and reveals tradeoffs between different implementations. Finally, the conclusions of the dissertation are summarized in Chapter 6.

2. BACKGROUND

2.1. Cryptography

Cryptography is the study of manipulating a message such that any intercepting party cannot understand it without the knowledge of the algorithm and the key [1]. The message that is intended to be conveyed is referred to as the *plaintext*, and the altered version of this message is referred to as the *ciphertext*. The *key* refers to the information required to encrypt or decrypt the message. The algorithm that converts plaintext to ciphertext, or vice versa, using the key is defined as the *cipher*.

Throughout the course of history, numerous ciphers have been invented for the purpose of cryptography. However, its use was never as commonplace as it is now. As we enter the information era, the proliferation of computing systems and digital communications has made cryptography more crucial and its applications more ubiquitous. To cater to these newfound use cases, cryptography techniques have changed drastically in the last decade. As a result, modern cryptography differs significantly from the conventional meaning, as we will explain in the following subsection.

2.1.1. Modern Cryptography

Modern cryptography is the cornerstone of advanced computing and information security. Perhaps one of the most obvious differences between modern and traditional encryption is how they operate on the message. Unlike traditional methods that manipulate a set of characters (such as the characters in an alphabet), current cryptographic primitives operate on binary bit sequences. The algorithms in use are entirely dependent on mathematical principles like number theory, computational complexity theory, and probability theories, and there is no concept of security through obscurity [2]. Furthermore, traditional encrypted communication required an entire ecosystem of trusted agents, which is no longer the case in modern systems.

Despite the prevalent misconception that cryptography is synonymous with encryption, modern cryptography promises to provide the following four core information security services: Confidentiality, authentication, non-repudiation, and data integrity.

- Confidentiality: Confidentiality (also referred to as secrecy) is the ability to ensure that the message can be only disclosed by the intended recipient.
- Authentication: Authentication is the capacity to establish that a person or program is the intended sender of a communication, or who they claim to be.
- Non-repudiation: Non-repudiation is the guarantee that the sender of information is supplied with proof of delivery and that the recipient is provided with proof of the sender's identity.
- Data integrity: Integrity is the protection of a communication against unauthorized information alteration or deletion. Typically, the word incorporates assurances of validity and non-repudiation [3].

2.1.1.1. Modern Ciphers. Modern ciphers can be distinguished into two branches by the method they carry the encryption and decryption as symmetric and asymmetric ciphers. Symmetric ciphers are algorithms that use the same key for encryption and decryption. Asymmetric ciphers, on the other hand, require the use of multiple keys, namely one public key and one private key for each agent in communication. The messages are encrypted by the public key, which is known to all agents, but they can only be decrypted by the private key. Since each agent only has the information of their respective private keys, only the intended receiver can decrypt the message, guaranteeing confidentiality. A comparison of symmetric and asymmetric algorithms can be seen in Figure 2.1.

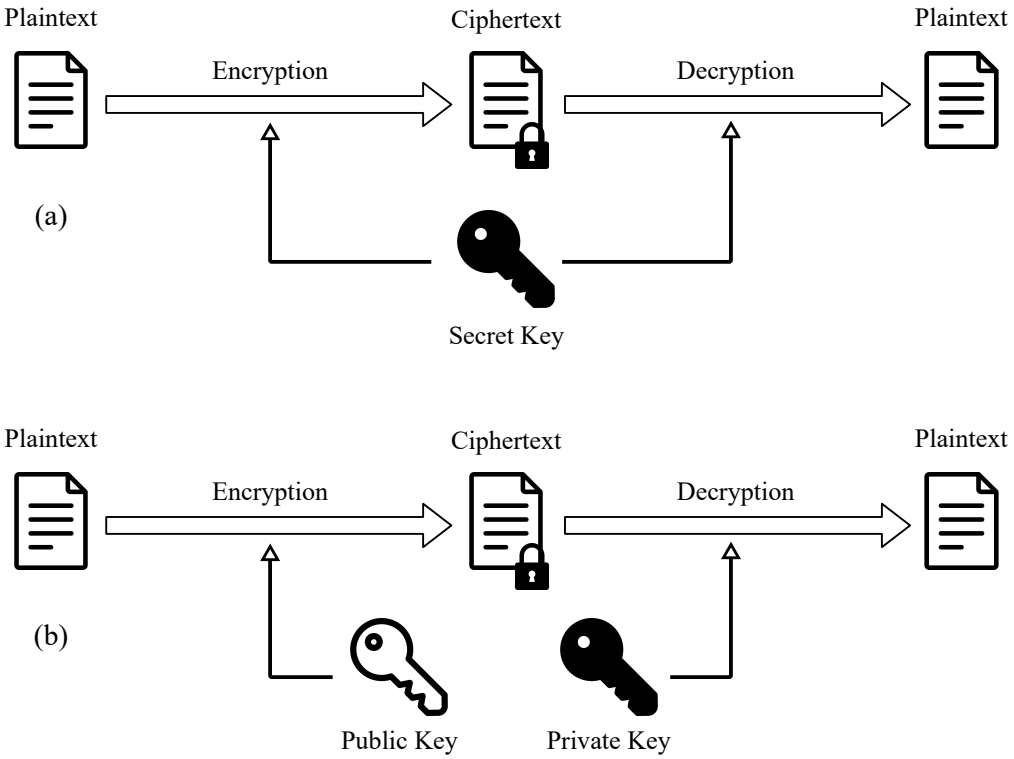


Figure 2.1. Overview of (a) Symmetric, (b) Asymmetric Cryptography.

The symmetric algorithms are significantly simpler and faster than asymmetric ones, but they require sharing a secret key. Such a secure channel may not be present between the agents in most cases. In practice, asymmetric algorithms are used to share this secret key; then, the communication is carried out using the more efficient symmetric encryption. This thesis will focus on symmetric cryptography and its lightweight applications, as explained in Section 2.1.2.

Two of the most prevalent types of symmetric ciphers are block ciphers and stream ciphers. A block cipher is a cryptographic scheme where the input message is processed in blocks of a pre-determined size [1]. They are widely employed in symmetric cryptography. The most prominent example is the NIST-recommended Advanced Encryption Standard (AES) algorithm [4], which uses blocks of 128-bits. AES lies within the class of Substitution Permutation Networks (SPN) category of block ciphers. Feistel network is another cryptographic scheme from which numerous block ciphers can be

derived. Data Encryption Standard (DES) is one such popular example. Both SPNs and Feistel Networks provide the confusion and diffusion required for security against statistical attacks [5]. The alternative to the block ciphers is the stream cipher, in which the input message is processed one bit at a time. After receiving an input bit and generating an output bit by a series of processes, the procedure is repeated for all bits of the message [1].

2.1.2. Lightweight Cryptography

Small computing devices, including RFID tags, sensor nodes, medical implants, smart cards, and industrial controllers, are becoming increasingly prevalent with the expansion of the IoT. The transition from desktop computers to mobile devices introduces numerous new security and privacy concerns. Implementing traditional cryptography protocols on small devices is often difficult because the tradeoff between security, speed and resource consumption are usually tuned for desktop and server implementations. Even if they are applied in resource-constrained devices, their performance will likely be unsatisfactory [6].

Lightweight cryptography is a branch of cryptography dedicated to developing solutions for devices with limited resources. As shown in Figure 2.2, it targets devices on the lower end of the spectrum. Various constraints in these devices include the limited amounts of memory space available in small microcontrollers, RFID tags with significantly small power budgets, sensor nodes with timing requirements, area-cost constraints, and so on.

It's worth noting that in many cases, high-end devices will also need to implement these lightweight algorithms. For example, many IoT and sensor nodes transfer data to an aggregator or a centralized device with higher computing capabilities. Therefore, the need for lightweight cryptography does not only arise from the limitations of a particular device but all agents in the network.

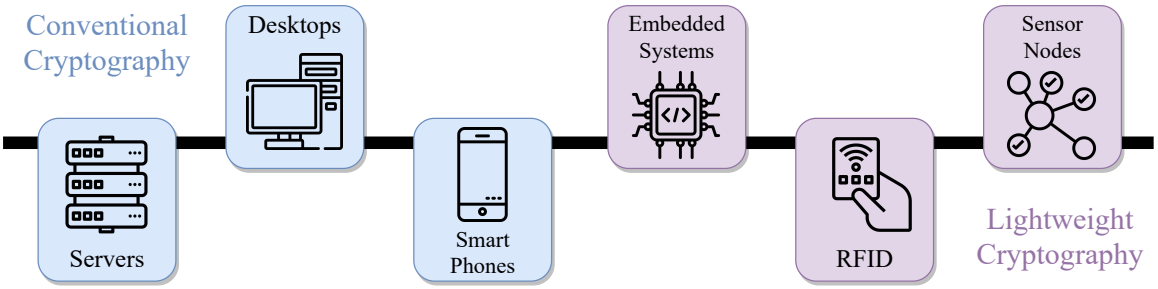


Figure 2.2. Cryptography Device Spectrum.

2.1.2.1. Authenticated Encryption with Associated Data. Authenticated encryption with associated data (AEAD) refers to encryption systems that offer confidentiality and integrity [7]. In addition to the traditional plaintext and key inputs, AEAD structures have Associated Data (AD) and nonce inputs [8]. The output ciphertext is also paired with an authentication tag. The tag is the message authentication code (MAC) that provides authenticity and can be generated using any of the authenticated encryption schemes: Encrypt-then-MAC, Encrypt-and-MAC, or MAC-then-encrypt.

AD is separated from the message input because, in several application configurations, we desire to not only encrypt and authenticate messages but also to contain auxiliary data that should be authenticated but not encrypted [8]. A network packet in which the data payload should be encrypted (and authenticated) but the header should be unencrypted is one such example [9].

The nonce input is used in AEAD protocols to accomplish semantic security. It is the sender's obligation to never reuse a nonce. The sender must maintain a counter or comparable state with a lengthy repetition time. The receiver is not required to have a replay-detection mechanism.

Both the nonce and the AD are required for decryption, despite not being considered part of the key or the ciphertext. How the receiver is made aware of the AD content lies outside the scope of the AEAD scheme.

2.1.3. NIST LWC Competition

National Institute of Standards and Technology (NIST) launched a lightweight cryptography project in 2013 to investigate the performance of existing cryptographic standards on restricted devices, determine the need for specialized lightweight cryptography standards, and host a transparent standardization procedure if one is needed.

In August 2018, NIST issued a call for algorithms for consideration in lightweight cryptography standards. They received 57 proposals for consideration; all except one were chosen as Round 1 Candidates after an initial examination. Thirty-two candidates were selected to advance to Round 2 from a pool of 56 in Round 1. NIST revealed ten finalists on March 29, 2021 [10].

Unlike the AES process carried out between 1997 and 2000, NIST has declared that they are planning to announce multiple winners to create a portfolio of lightweight cryptography algorithms. Another key difference is that the winners of the LWC competition will be recommended for use under specific constraints instead of being approved for general use. In other words, it does not have a goal of replacing AES and setting a new encryption standard. While having more than one winner may appear to be unfavorable from a common implementation perspective, it is argued that there are various devices with different constraints under the lightweight category, and it is not possible for a single algorithm to perform optimally in all implementations.

2.1.3.1. Motivation. The LWC project aims to address the lack of standardization for cryptographic applications in constrained environments that are not well served by existing standards.

2.1.3.2. Scope. Block ciphers, authenticated encryption techniques, hash functions, message authentication codes, cryptographic permutations, and stream ciphers can be considered within the scope of the lightweight cryptography project. However, for

this competition, NIST specifically required each submission to implement the AEAD functionality using a single or a group of algorithms. The desired features of AEADs are explained in the following subsection. Each submission is required to include:

- Complete algorithm specification with all the necessary mathematical operations, equations, tables, and diagrams.
- Justification of important design decisions, such as constants and look-up tables utilized in the algorithm, if any.
- Realistic values for all adjustable parameters, as well as a study of how these settings affect algorithm performance and security.

While asymmetric cryptography is not within the focus of this competition, it can be considered within the broader context of NIST's LWC project [6]. It is mentioned that future lightweight public key cryptography schemes will have to rely on lightweight primitives to achieve known public key cryptography methods. They should also be resilient against attacks that are enabled by quantum computing.

2.1.3.3. Desired Features of AEADs. Desired characteristics of AEAD modes can be summarized as follows:

- Single-Pass: Makes only one pass over the data, doing everything necessary to preserve privacy and authenticity at the same time.
- Low State-size: Internal state that corresponds to the size of memory required should be kept small.
- Inverse-Free: Decryption algorithm does not require an inverse implementation of underlying primitives.
- On-line: Each plaintext can be encrypted on the fly without requiring the knowledge of subsequent plaintext blocks.
- High Rate: The number of message blocks processed on each primitive invocation is described as the rate. Constructions with a greater rate minimize latency and are especially useful for achieving higher speed.

- Optimal: Uses the smallest number of non-linear invocations possible to increase the efficiency for short messages and reduce the latency.
- Nonce Misuse Resistance: Security is preserved even if the nonce is reused or absent. Ideal for light-weight applications where keeping a counter or producing a random number is a challenge.
- Integrity under RUP: Limited resources may force the decryption algorithm to release plaintext before verification. If possible, exploitation of unverified plaintext for forgery should be prevented.

2.1.3.4. Metrics. Can be categorized as hardware and software. Software implementations of LWC algorithms are evaluated on the following criteria:

- Code size (bytes): Algorithm size is calculated by subtracting the code size of the empty wrapper from the total code size. Encryption-only and decryption-only versions compared separately.
- Timing (cycles/byte): Comparison between AEAD modes algorithms done for varying input sizes for plaintext and AD, from a few bytes up to 2 kB.

Hardware implementations of LWC algorithms and their various modes are evaluated using the following metrics:

- Area: Resource utilization of the LWC algorithm hardware, constructed in compliance with the NIST Application Programming Interface (API) [11]. Unit of measurement varies between Look-Up Tables (LUT), Flip-Flops (FF), Logic Elements (LE), and FPGA slices.
- Energy-per-bit (nJ/b): Energy consumption of LWC algorithm operation per a single bit of message/AD.
- Maximum Frequency (MHz): Maximum operating frequency comparison. Calculated from the critical path on each LWC hardware.
- Throughput (Kbit/s): The absolute maximum throughput that hardware can support. Calculated using the cipher's maximum frequency and cycles-per-bit.

2.1.3.5. Benchmarking. For performance analysis, benchmarks have been executed in software [12, 13] and hardware [14, 15] by the cryptography community. NIST has published benchmarking frameworks for both areas in order to enable efficient and fair comparison across algorithms, or various implementations of the same algorithm carried out by different parties. This wrapper is in the form of a header file and a common function wrapper in software [12], and an API description in hardware [11], both provided with example designs.

Software benchmarks are further classified by various processors, microcontrollers [12], and instruction set architectures (ISA) [13]. Hardware benchmarks are separated into two groups: FPGA implementations [14], and ASIC digital design flow results [15].

2.1.4. Sponge Function

Typically, encryption algorithms such as AES use block ciphers. Block ciphers encrypt plaintext by splitting it into fixed-length blocks and using a secret key to encrypt each block. Various modes of operations determine the relation between these blocks. For example, in the Electronic Code Book (ECB) mode of AES, different blocks are encrypted and decrypted independently of each other. In contrast, in the Cipher Block Chaining (CBC) mode, blocks are used to form a chain where the ciphertext of the previous block is used to generate the initialization vector of the next block. Recently, sponge structures have begun to be utilized as a substitute for the block cipher modes.

Sponge constructions are a class of cryptographic algorithms. Historically, they have been incorporated into hashing algorithms, with Secure Hash Algorithm (SHA-3) being the most prevalent example [16]. The fundamental building block of a sponge is a function f that maps bitstrings of a specified length to bitstrings of the same length, such that each different input results in a unique output.

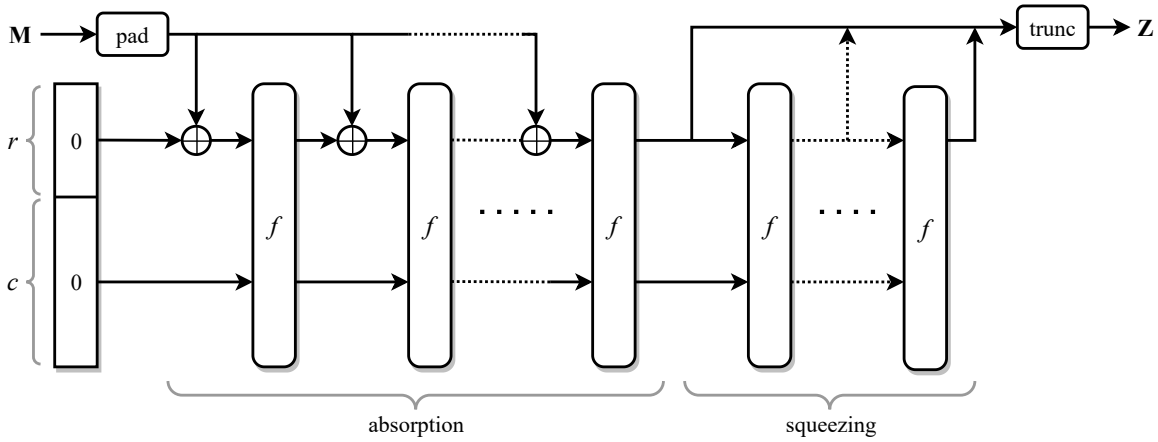


Figure 2.3. Generic Sponge Function.

Figure 2.3 shows the operation of a standard sponge function. The state is divided into the *bitrate* and *capacity* sections, denoted with r and c, respectively. The initial state goes through absorbing and squeezing phases (hence the name, sponge) to produce the output [17]. The input message is partitioned into blocks of r bits each. The first block of the input message is XORed with the first r bits of the state during the absorbing phase. The state is then modified using the function f. This procedure is repeated until all message blocks have been absorbed. If the message size is not an integer multiple of the bitrate, it is padded so it can be parsed into equal-sized blocks.

In the squeezing phase, the first r bits of the state are extracted, and the state is updated with the function f. This process is repeated until the desired number of output bits is received. The concatenation of all the extracted bits is the sponge function's output (i.e., the message digest). If the desired number of output bits is not an integer multiple of the bitrate, the output message can be truncated by discarding the excess bits [17].

One of the most advantageous characteristics of the sponge construction for hash functions is that it can take an arbitrary-length input and generate a message digest of any specified length. However, it was discovered that the sponge structure is highly adaptable and may be utilized to create a variety of cryptographic tools. In particular,

its suitability for AEAD modes, in addition to hashing, made it a preferred cryptography scheme for many NIST LWC contestants. Various implementations of sponges with additional features and tweaks are featured among algorithms in Section 2.2.

2.1.4.1. Duplex Construction. Similar to the sponge construction, the duplex construction employs a transformation or permutation with a defined length, bitrate and capacity parameters, and a padding rule to construct a cryptographic system [18] as shown in Figure 2.4. Instead of having separate absorb and squeeze phases, the duplex system is made up of duplex objects. Duplex construction is particularly useful for AEAD modes because it can return output blocks before the entire input message is absorbed, analogous to the operation modes in block ciphers.

The security of the duplex construction is mathematically shown to be equivalent to the security of the sponge function with the same components [18]. In addition, it only uses a single call of function f per input block. It should be noted that unlike the sponge function, the duplex construction outputs a digest that is the same size with the input message. In addition, similar to the sponge function, the parametric structure of the duplex construction allows simple tradeoffs on the algorithm level. While increasing the bitrate improves throughput, increasing the capacity enhances security. Due to these characteristics, it is used for the encryption of the message by many LWC algorithms, as we will explore in Section 2.2.

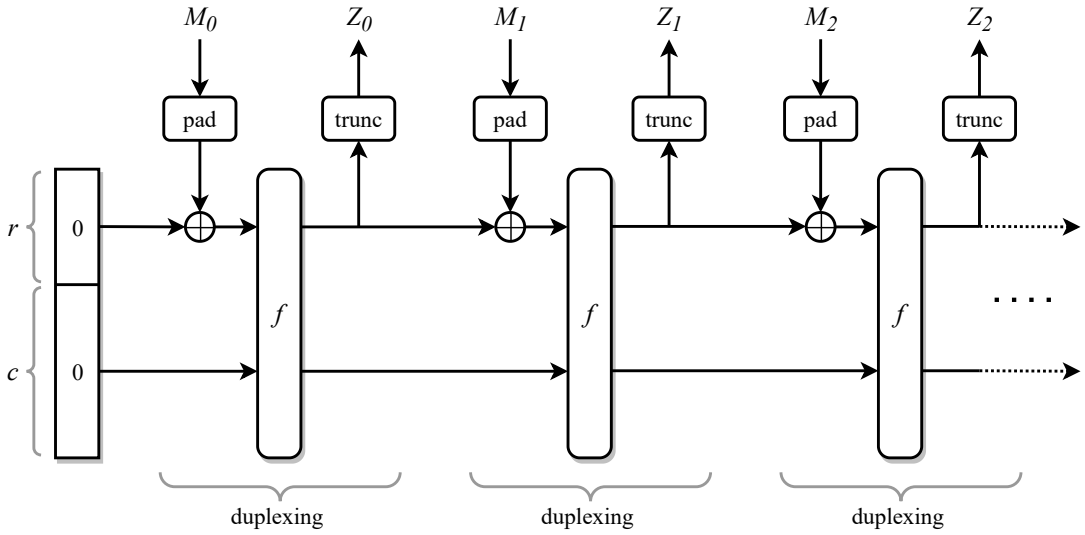


Figure 2.4. Duplex Construction.

2.2. LWC Algorithms

The NIST LWC competition has 10 submissions remaining as finalists as of August 2022. In particular, we focus on three of these algorithms in this thesis: ASCON [19], TinyJambu [20], and Photon-BEETLE [21]. While the proposed unified reconfigurable accelerator schemes in Chapter 3 are expected to work for many other finalists, we believe that implementation of these three algorithms is sufficient for proof-of-concept purposes. Our justification for selecting these algorithms are:

- Diversity: These algorithms are selected to be on the different regions of the performance-cost curve on both software and hardware implementations. According to the NIST LWC Round 2 Benchmarking Results, TinyJambu is the finalist with the lowest hardware and software footprint. ASCON implementations perform close to the top in terms of throughput and throughput per area. Finally, PHOTON-Beetle performs mediocre in state size, area, and performance compared to other applicants in hardware [14, 15].
- Compatibility: The selected algorithms are similar in their operation modes. In particular, their AEAD modes closely resemble sponge and duplex constructions,

cryptographic primitives which are widely adopted by NIST LWC Competition finalists.

2.2.1. ASCON

The Ascon cipher suite comprises a family of authenticated encryption plans, hash functions, and extendable output functions. Ascon's AEAD mode of action is based on Duplex Construction with a 320-bit state size. The encryption process is depicted in Figure 2.5.

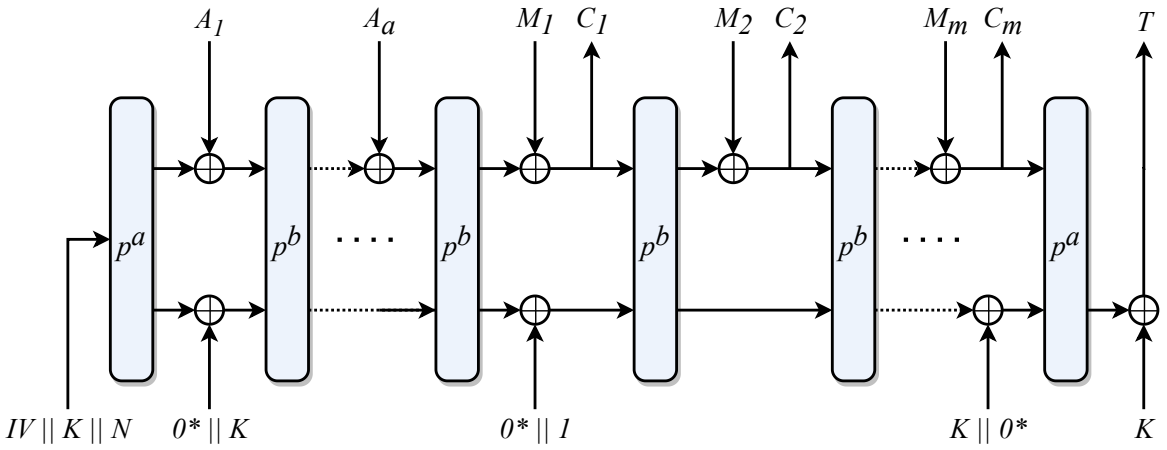


Figure 2.5. Ascon Encryption.

The initial state of Ascon is produced by concatenating K and N together with an Initialization Vector (IV) that specifies the algorithm parameters (key size k, the bitrate r, round numbers a and b, each allocating an 8-bit space). The initial state is updated with a rounds of permutation (p^a) followed by the addition of the secret key.

Ascon processes the AD and the message by dividing them into r-bit blocks, starting with the AD. At each step, a block of AD is XORed with the most significant bits of the state, and the state is updated using p^b. After processing the final AD block, a 1-bit constant is XORed to the state to separate the AD from the subsequent

message. This addition is done even if the AD is empty. Encryption and decryption are similarly performed by adding the message block to the bitrate portion of the state. The ciphertext then becomes a part of the new state in both cases. The resulting state is updated using p^b for each block apart from the final one. If the AD or the message is not divisible to r-bit blocks, they are padded by appending a single 1, followed by the least number of 0s to generate a multiple of r bits.

In the finalization stage, the secret key K is XORed to the internal state, followed by the permutation p^a. The authentication tag T is derived from the least significant 128 bits of the state XORed with the key.

2.2.1.1. Permutation. Ascon's permutation is defined on five 64-bit words and can be implemented by using only bitwise rotations and Boolean functions within words. The permutations repeatedly apply an SPN-based round transformation consisting of a round constant addition, a substitution layer, and a linear layer.

- Add round constant: adds round constant to the least significant bits of the middle word of the state. The round constants can be easily calculated using the current round number i, and total round numbers a and b.
- Substitution layer: Updates the state S with 5-bit S-boxes shown in Table 2.1. Each input bit is taken from the same position of a different 64-bit word, allowing parallel application of all S-boxes.
- Linear diffusion layer: Applies the linear functions

$$\begin{aligned} L(x_0) &= x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\ L(x_1) &= x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\ L(x_2) &= x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\ L(x_3) &= x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\ L(x_4) &= x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \end{aligned} \tag{2.1}$$

  to each 64-bit word of the state to provide diffusion.

Table 2.1. 5-bit S-box used in Ascon's permutation.

$S(x)$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c
1	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

2.2.1.2. Versions. Recommended parameter sets for AEAD mode of Ascon is given in Table 2.2 in the priority order, Ascon-128 being the primary recommendation.

Table 2.2. Recommended parameter sets for Ascon.

Variant	size (bits)				rounds	
	key	**nonce**	**tag**	**rate**	***a***	***b***
Ascon-128	128	128	128	64	12	6
Ascon-128a	128	128	128	128	12	8
Ascon-80pq	160	128	128	64	12	6

2.2.2. TinyJAMBU

The TinyJAMBU is a lightweight AEAD mode with a state of 128 bits. A 128-bit keyed permutation is used, and the message block size is 32 bits [20]. While it resembles a sponge construction with 3-bit constant additions (referred to as *FrameBits*) between blocks, it has been shown to provide marginally stronger security than the Duplex mode [20]. The *FrameBits* takes the hexadecimal values 1, 3, 5, and 7 for initialization, AD processing, plaintext processing, and finalization (tag generation) stages of the algorithm, respectively.

Figure 2.6 demonstrates the encryption with TinyJAMBU mode. Initially, the 128-bit state is set to zero and updated with P_a (by applying the permutation for a rounds). This is followed by the nonce setup. The *Framebits* of the nonce (binary value 1) is added to the state; then, we update the state using the keyed permutation P_b,

then 32 bits (a block) of the nonce are added to the most significant word of the state. This is repeated three times for the 96-bit nonce to be absorbed. The processing of AD is identical to the nonce absorption with two differences. The *FrameBits* value is 3, and the round number is changed to a. If the last block of AD is less than 32-bits, its number of bytes is added to the state.

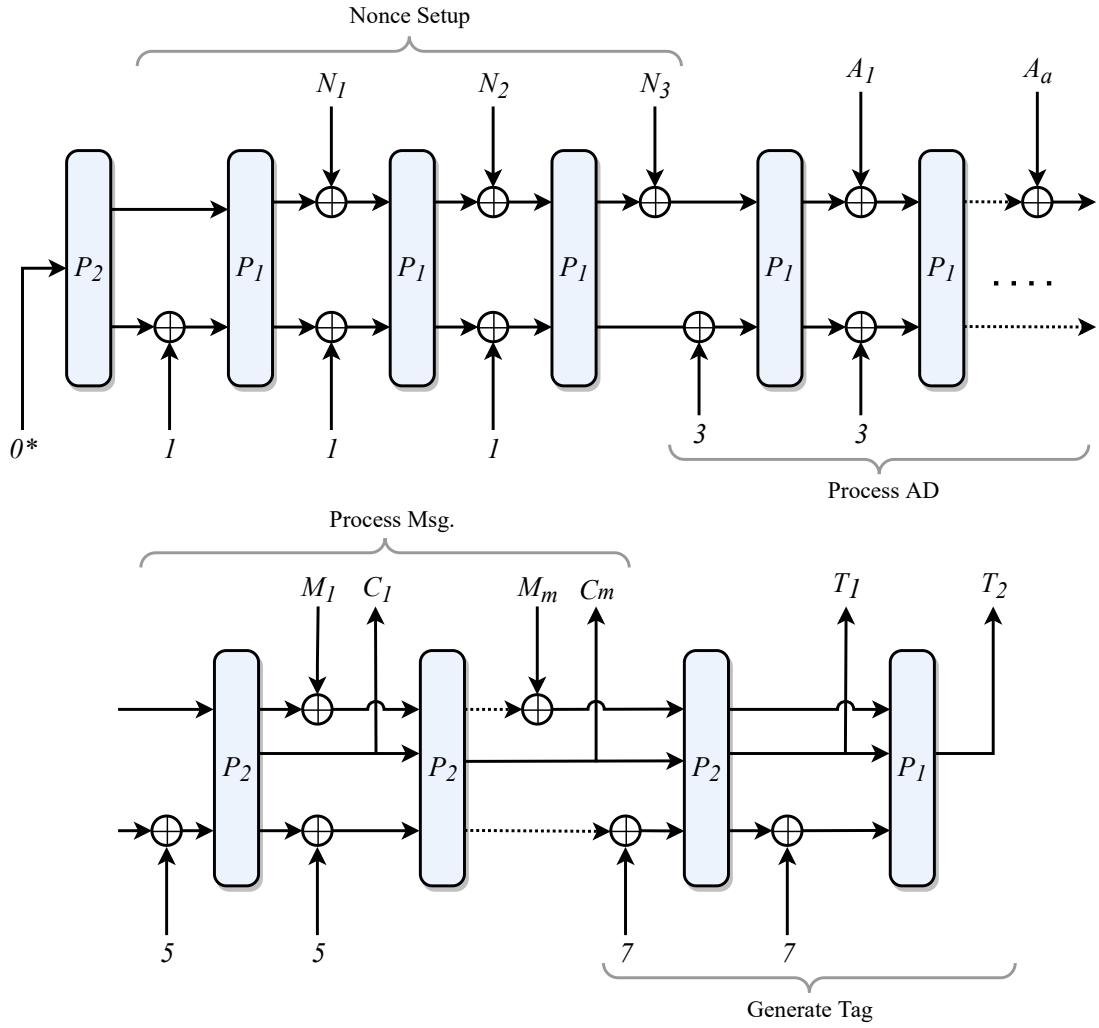


Figure 2.6. TinyJAMBU AEAD Mode of Operation.

The message encryption stage adds *FrameBits* to the state, updates it with P_b, then adds the message block to the state. Critically, ciphertext is not taken directly from the bitrate portion of state, instead it is calculated separately by XORing the plaintext with the second most significant word of the state. This, along with the

addition of *FrameBits*, separates this stage from the standard Duplex Construction. The number of bytes is again XORed to the state if the last block is partial.

Finally, the tag is generated by adding *Framebits*, updating the state with P_b, then reading the 2nd most significant word of the state. This is repeated again to gather a tag of 64-bits total, using a number of permutation cycles on the second step instead.

2.2.2.1. Permutation. TinyJAMBU uses a 128-bit keyed permutation. The 128-bit nonlinear feedback shift register (NLFSR) shown in Figure 2.7 uses a different bit of the input key at each round to update the state. The permutation is designed in a way such that 32 rounds can be computed in parallel.

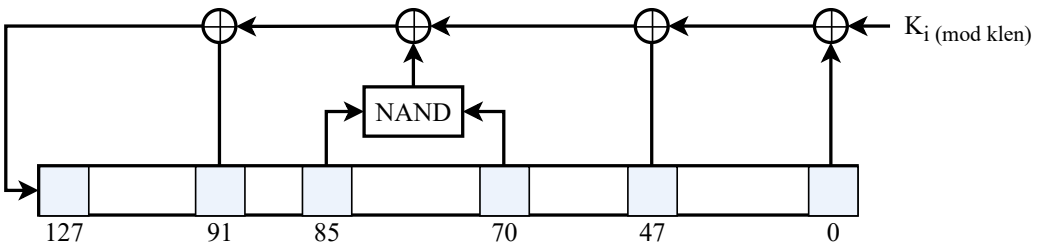


Figure 2.7. The NLFSR Used in TinyJAMBU's 128-bit Permutation.

2.2.2.2. Versions. Recommended parameter sets for TinyJambu is given in table 2.3. The primary variant is TinyJAMBU-128. Bitrate is fixed to $r = 32$ for all variants.

2.2.3. PHOTON-Beetle

PHOTON-Beetle is a series of authenticated encryption and hash functions that employs a sponge-based mode Beetle with the PHOTON_{256} being the internal permutation [21]. Figure 2.8 summarizes the operation of the AEAD mode.

Table 2.3. Recommended parameter sets for TinyJAMBU.

Variant	size (bits)				rounds	
	key	**state**	**nonce**	**tag**	***a***	***b***
TinyJAMBU-128	128	128	96	64	1024	640
TinyJAMBU-192	192	128	96	64	1152	640
TinyJAMBU-256	256	128	96	64	1280	640

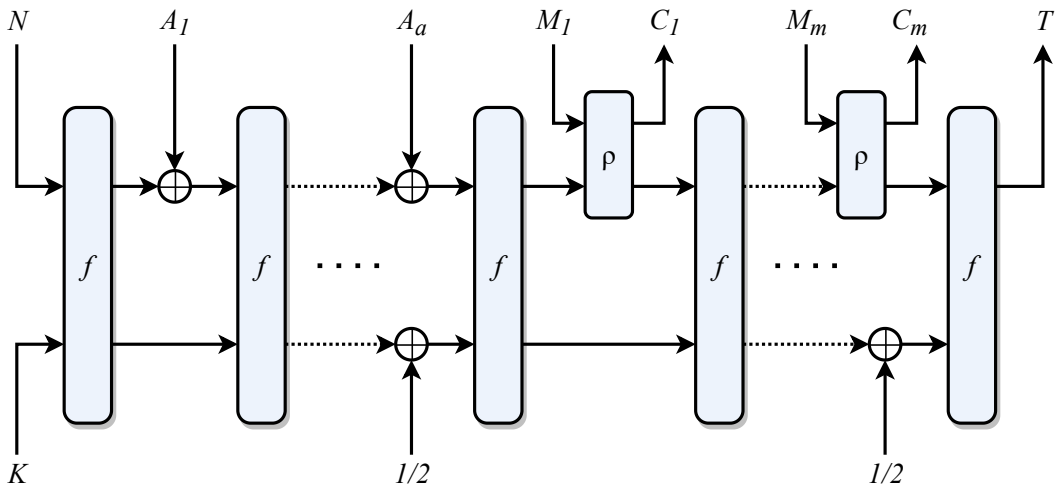


Figure 2.8. PHOTON-Beetle AEAD Mode Operation.

During encryption, the initial state is established by the concatenation of the nonce N and the key K. The AD is then processed in the same way as the original sponge mode: State is updated using PHOTON_{256} at each step, followed by the absorption of the AD block by XORing it with the first r bits of the state. This is repeated for all AD blocks.

After processing the AD, a similar operation is carried on the message. To produce the ciphertext block, the rate portion of the permutation output is shuffled and then XORed with the message block. This phase distinguishes PHOTON-Beetle from Sponge Duplex, in which the rate component of the next permutation input is out-

putted as the ciphertext block. During message processing, this state update and ciphertext creation are handled by the function ρ. The rate part of the state is returned as the authentication tag after processing the final message block.

In order to distinguish processing of AD and the plaintext, 3-bit constants are added to the state after the AD and message processing. These constants are different under empty AD and/or empty message cases. The decryption differs from the encryption only due to the fact that it uses the inverse of the linear function ρ to generate the plaintext from the ciphertext.

2.2.3.1. Linear Function. ρ is the linear function used during the message absorption. Its inputs are an r-bit state S and an r-bit input data U (padded to r-bits for partial blocks). It updates the state by XORing it with the input data, and returns an output V by XORing the input with the shuffled state. ρ^{-1} is the inverse function of ρ that is used for decryption, and reproduces the plaintext block U and the original state S from the current state and the ciphertext block V. Table 2.4 provides the descriptions for ρ and ρ^{-1} functions.

Table 2.4. Linear function used in PHOTON-Beetle during output generation.

$\rho(S, U)$	$\rho^{-1}(S, U)$	*Shuffle(S)*
V = Trunc(*Shuffle(S)*, $\|U\|$) $\oplus$ U $S = S \oplus \text{Ozs}_r(U)$ return (S, V)	U = Trunc(*Shuffle(S)*, $\|V\|$) $\oplus$ V $S = S \oplus \text{Ozs}_r(U)$ return (S, U)	$S_1 \| \| S_2 = S$ return $(S_2 \| \| (S_1 \ggg 1))$

2.2.3.2. Permutation. PHOTON-Beetle uses PHOTON_{256} as the underlying permutation. This 256-bit permutation is performed on the state, which is represented as 64 4-bit cells, organized as an 8-by-8 matrix, as shown in Figure 2.9. PHOTON_{256} contains AddConstant, SubCells, ShiftRows and MixColumnSerial layers, and operates by iterating all four layers in order [21].

- AddConstant: XORs pre-defined constants to the first column of the matrix.
- SubCells: Applies a 4-bit S-box to each element of the matrix.
- ShiftRows: Rotates the position of each cell in a row by the amount equal to its row number.
- MixColumnSerial: Applies a matrix multiplication on a 4th degree Galois Field (GF) on all columns to linearly mix them. The irreducible polynomial is x^4+x+1.

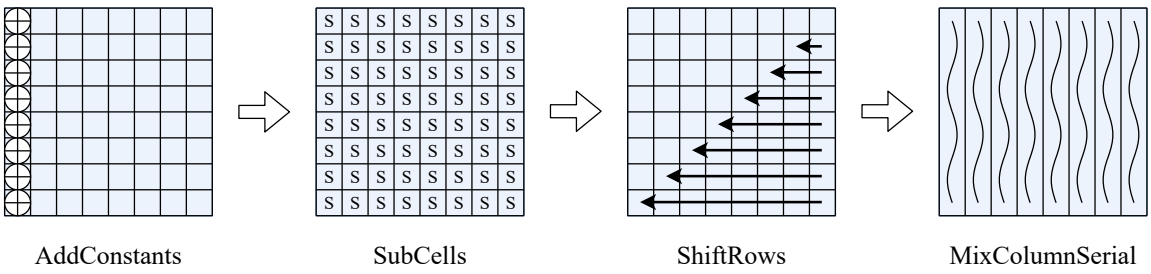


Figure 2.9. $PHOTON_{256}$ permutation.

2.2.3.3. Versions. Key, nonce, and tag length are fixed to 128-bits in the PHOTON-Beetle AEAD family. The rate of absorption r is modified depending on the target cipher version:

- PHOTON-Beetle-AEAD[128]: Primary implementation with $r = 128$. This design aims to achieve high throughput while allowing implementations with a low hardware footprint.
- PHOTON-Beetle-AEAD[32]: Reduces bitrate to $r = 32$ to trade off throughput for increased security while further reducing the hardware cost.

2.3. Partial Reconfiguration

Partial reconfiguration is the capacity to reconfigure specified regions of an FPGA at any moment after its initial configuration. Figure 2.10 shows the partial reconfiguration process, where mutually exclusive implementations of the reconfigurable module

are referred as *modes*. Using partial reconfiguration grants the design the following advantages:

- Enhanced system functionality. While a specified segment of the design is being modified, the remainder of the system may continue to function normally.
- Partial reconfiguration enables maintenance, service, and upgrade of hardware in the field with relative ease.
- The hardware sharing enables the execution of different applications on a single FPGA, hence decreasing the overall number of devices. This decreases total system energy consumption and device expenses.

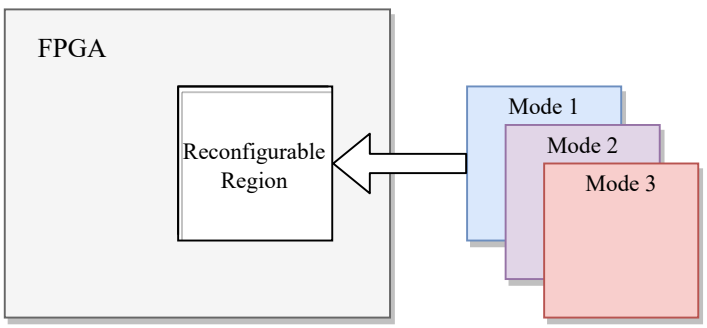


Figure 2.10. Partial Reconfiguration.

Partial reconfiguration has various degrees. The baseline is Static Partial Reconfiguration (SPR), where the device operation is suspended during reconfiguration. Dynamic Partial Reconfiguration (DPR) allows the static components of the design to be uninterrupted during the reconfiguration process. This function is supported by more recent Xilinx devices under the term Dynamic Function Exchange (DFX) [22]. In its simplest form, Partial Reconfiguration can be carried by the standard means of configuring the FPGA, i.e., by connecting the programming cable to the FPGA device and using the vendor-provided software. This requires an external connection with the programmable fabric to be present.

2.3.1. Dynamic Partial Self-Reconfiguration

Dynamic Partial Self-Reconfiguration (DPSR) is a more sophisticated form of partial reconfiguration in which the FPGA can reconfigure its programmable region at runtime without the need for external agents. Figure 2.11 compares several partial reconfiguration methods discussed.

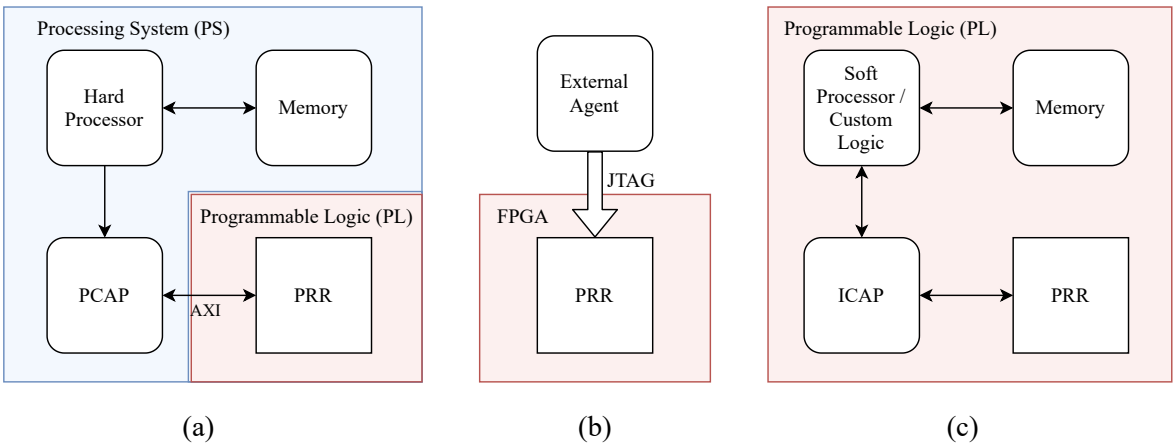


Figure 2.11. Partial Reconfiguration Using (a) PCAP, (b) External Agent, (c) ICAP.

Self-reconfiguration requires specific hardware resources to be present in the FPGA. Processor Configuration Access Port (PCAP) is one such primitive, which exists in some FPGAs with hard Processing Systems (PS), such as Zynq devices. It allows the PS to access and program the Programmable Logic (PL). A more ubiquitous resource is the Internal Configuration Access Port (ICAP), which enables the configuration of the PL from within the FPGA. Altera FPGAs employ a dedicated IP block called the Partial Reconfiguration IP (PR-IP) for this purpose [23]. It allows partial reconfiguration from the internal PR controller as well as external hosts when provided with adequate interfacing.

In this thesis, we will be using ICAP to maintain generality. The constructed ICAP controller hardware is explained in Section 3.3.2, and the Implementation details of the DFX flow is explained in Section 4.1.1.

3. LWC ACCELERATOR DESIGN

An accelerator is a specialized architectural substructure that is engineered for a particular category of applications. This design rationale allows them to improve performance and/or reduce the energy consumption of systems when integrated into general-purpose hardware [24]. Due to various types of hardware accelerators' availability, they have a variety of use cases ranging from high-performance computing and data centers to mobile and IoT devices. We investigate the hardware accelerators under two categories: Tightly-coupled accelerators (TCA) and loosely-coupled accelerators (LCA) [25].

TCAs are comprised of dedicated hardware computing units which can accelerate critical sections of a program, often integrated into the CPU's pipeline. This tight integration allows them to be used without any runtime overhead and effectively share the resources the CPU already has, such as the register file and the hierarchical memory access. However, from a hardware perspective, the integration of TCAs can prove to be a challenge. They often have no standardized interfacing due to the close integration and require the modification of the processor core itself. This may cause further complications in the design flow as the verification, and the timing analysis should be remade.

On the other hand, LCAs are external hardware that is often memory-mapped and can be accessed by the CPU via the on-chip interconnect. Since they are not constrained by the space and timing available within the CPU pipeline, they can have more sophisticated datapath structures, internal controllers, and even their own instruction set architectures (ISA). Although the term LCA is inclusive to the complex architectures that partially reside off-chip, or have their own memory hierarchy or scratchpad [26], we will be using this term to refer to structures that are similar to peripheral devices on a System-on-Chip (SoC) within the context of IoT.

In this thesis, we will be constructing a reconfigurable lightweight cryptography accelerator that can support different modes of an algorithm; or entirely different algorithms with the use of dynamic runtime partial reconfiguration on FPGA. We prefer an LCA structure because of its ease of integration to the readily deployed systems. Also, they do not require ISA modifications and are more in line with the theme of multi-purpose reconfigurability in general. Additionally, we realize that the data transfer overhead can be resolved in favor of the loosely-coupled accelerators if a proper direct memory access (DMA) structure is used to manage the data flow from the memory to the accelerator and vice versa.

In the following sections, we propose two different LCAs for LWC. The first one is a more hardware-oriented approach, where the accelerator is controlled by a number of control-status registers. The second approach is closer to hardware-software codesign, as we will propose an accelerator with its own custom ISA. We follow a similar process in designing both accelerators: Isolate the sections in LWC algorithms that are different from each other. This is primarily the controller part in the hardware approach and software in the ISA-based approach. The remaining common parts are constructed to support as many current and future LWC algorithms as possible.

3.1. Fully-Hardware Approach

For the design of this reconfigurable accelerator, we start by determining characteristics shared by different LWC algorithms, using the NIST competition finalists as our primary point of reference. The key observations we make are:

- All algorithms undergo the phases of initialization, AD processing, message processing, and finalization. Although there are minor differences in how they move across these stages, we anticipate that their hardware implementations will have similar finite state machines (FSM).
- All algorithms revolve around a fixed permutation with at most two different round numbers.

- All algorithms have the same I/Os per the competition rules, meaning they could have the same data I/O, control, and status registers if made into a bus-accessible accelerator.

Based on these points, we characterize accelerator hardware components as fully shared, partially shared, or not shared across algorithms. From a modular perspective, the accelerator consists of the register interface, controller, and cipher, as shown in Figure 3.1. The following subsections explore how these submodules are classified according to these criteria.

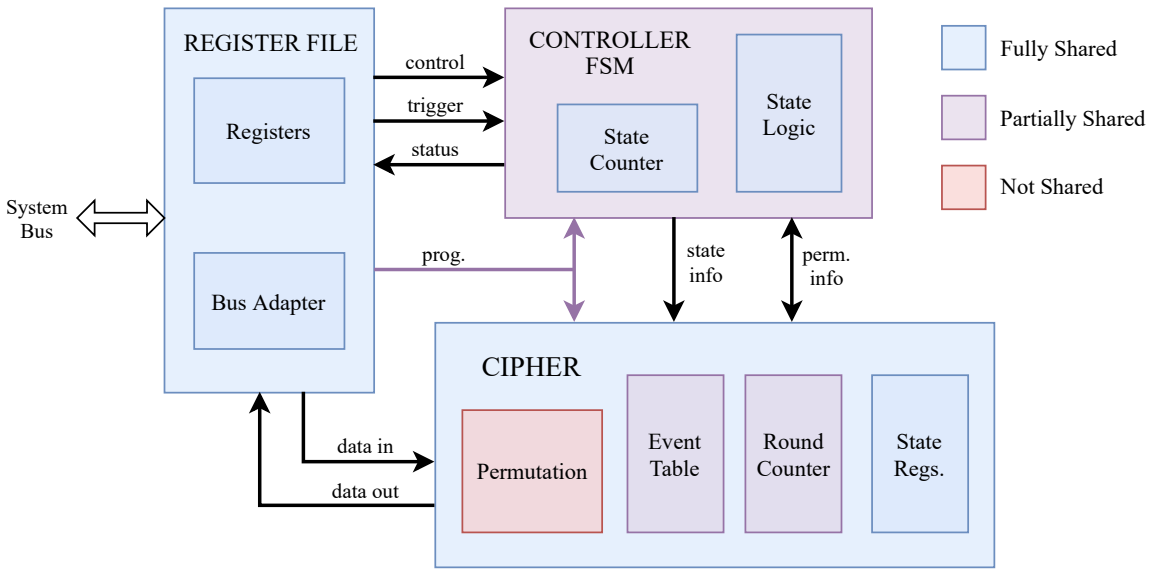


Figure 3.1. Top-Level Block Diagram of the Accelerator.

3.1.1. Register Interface

This is the accelerator component that interacts with the system bus. It consists of a bus adapter, CSRs, read and write data registers, and metadata registers. The entirety of the register interface is considered *fully shared*.

3.1.1.1. Control-Status Registers. The list of CSRs in the accelerator is provided in Table 3.1. Each 32-bit register is accessible by the CPU as a single memory location. Still, the underlying bit slices are used to monitor or control different aspects of the accelerator, as explained below. The following are the read-only status registers:

Table 3.1. List of Control-Status Registers.

Control Regs.		**Control2 Regs.**		**Status Regs.**	
Bit Slice	**Register**	**Bit Slice**	**Register**	**Bit Slice**	**Register**
[0]	Mode	[15:0]	AD length	[0]	Idle
[3:1]	Key length	[31:16]	Message len.	[1]	Stalled
[6:5]	-			[2]	Tag valid
[11:6]	Round num. a			[3]	Output valid
[17:12]	Round num. b			[4]	Input ready
[19:18]	Rate			[5]	Fault Alert
[21:20]	Nonce length				
[23:22]	Tag length				

- Idle: The accelerator is in the idle state.
- Tag valid: The encryption/decryption is completed and waiting for the authentication tag to be read.
- Output valid: An output block is generated and ready to be read. The accelerator is not allowed to process the next block until all words of the output block are read.
- Input ready: The accelerator is waiting for data inputs.
- Fault alert: Two or more erroneous read/write attempts have been made in succession.

Next, we have the control registers, which can be read at any time but can only be written when the accelerator is in the idle state:

- Mode: Select operation mode (encryption/decryption).
- Key length: Size of the secret key in words (between 32-256 bits with 32-bit increments).
- Round numbers (a,b): Number of rounds of permutation.
- Rate: Rate of absorption, i.e., The number of data or AD words absorbed in each step.
- Nonce length: Size of the public nonce input in words.
- Tag length: Size of the output authentication tag in words.
- AD length: Size of the AD input that will be processed in this accelerator run, in bytes.
- Text length: Size of the input plaintext/ciphertext in bytes.

3.1.1.2. Data Registers. Table 3.2 provides a list of data registers together with their respective read/write permissions. In addition to their directions, a read-write of these registers is restricted to specific intervals during the control flow of the accelerator. An AD input register, for instance, cannot be written until the accelerator has completed processing the previous AD block and is awaiting the next one. Similarly, output data can only be read if the processing of a message block is recently completed. Unauthorized read-write operations result in an error message being returned by the bus, and repeated attempts will trigger a fault.

Table 3.2. List of Data Registers.

Register	**Bit Size**	**Permissions**
Nonce In	128	Write-only
Key In	up to 256	Write-only
Text In	128	Write-only
AD In	128	Write-only
Text Out	128	Read-only
Tag Out	128	Read-only

Additionally, each data register is accompanied by metadata registers, which track the accesses to these registers on a word basis. This information is utilized to generate internal trigger signals to the accelerator. For example, an algorithm with a block size (bitrate) of 128-bits requires four input words to be written before processing and four output words to be read from data output registers before starting to process the next block.

3.1.1.3. Program Registers. Program registers, similar to control registers, can only be written before the start of the accelerator run. They allow re-programming of *partially shared* blocks in the accelerator without requiring FPGA reconfiguration.

- Counter: Indicates which of the round numbers (a, b) should be used for which permutation.
- FSM: Programs state transitions of the FSM (see Section 3.1.2).
- Events: Programs the content of *events* between permutations (see Section 3.1.3).

3.1.1.4. Bus Adapter. This module is the interface between the system bus and the accelerator registers. It translates the requests from the bus into read/write signals native to the registers and converts the responses from the accelerator into the appropriate bus format. Changing this module allows the accelerator to interface with different bus protocols (AXI [27], TileLink [28], etc.) in the design time.

3.1.2. Controller

The controller is considered *partially shared*, in the sense that the states are consistent across algorithms, but there are minor differences between state transitions. We exploit this by fixing the states and the *trigger* signals that induce state transitions while permitting the actual state transitions to be configured via the program registers.

In hindsight, this looks like a lot of resources should be allocated to keep a record of all possible state transitions. But upon analyzing LWC algorithms, we reveal that state transitions are severely limited, even across different algorithms. Figure 3.2 shows all state transitions required to realize seven different algorithms out of 10 LWC competition finalists, where each color denotes an algorithm-specific state transition, and the black connections indicate state transitions used in all algorithms. This limited mobility allows us to represent an algorithm's entire state transition table with only 48 bits, requiring only one and a half words of write overhead in operation.

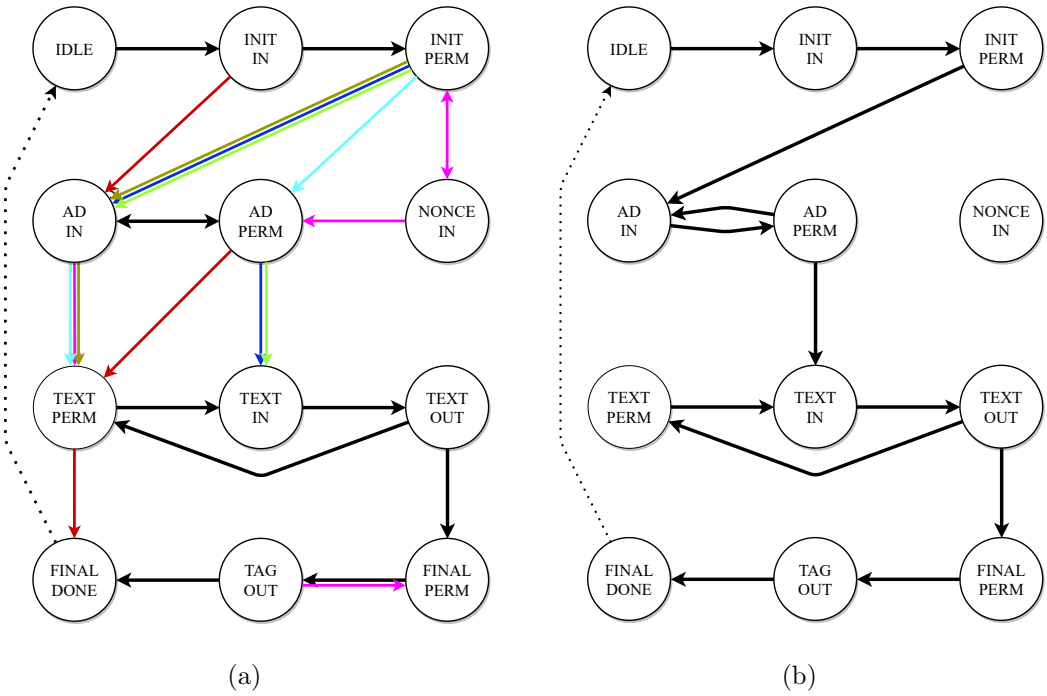


Figure 3.2. FSM Transitions for (a) Seven NIST LWC Algorithms, (b) Ascon.

The controller hardware is shown in Figure 3.3. As an example, the state transition table in the figure is filled for Ascon. The states are implemented in a one-hot fashion to reduce the power consumption and allow simple transitioning between stages with the help of a shifter. The shift amounts after each trigger are determined by the FSM program registers (see 3.1.1.3).

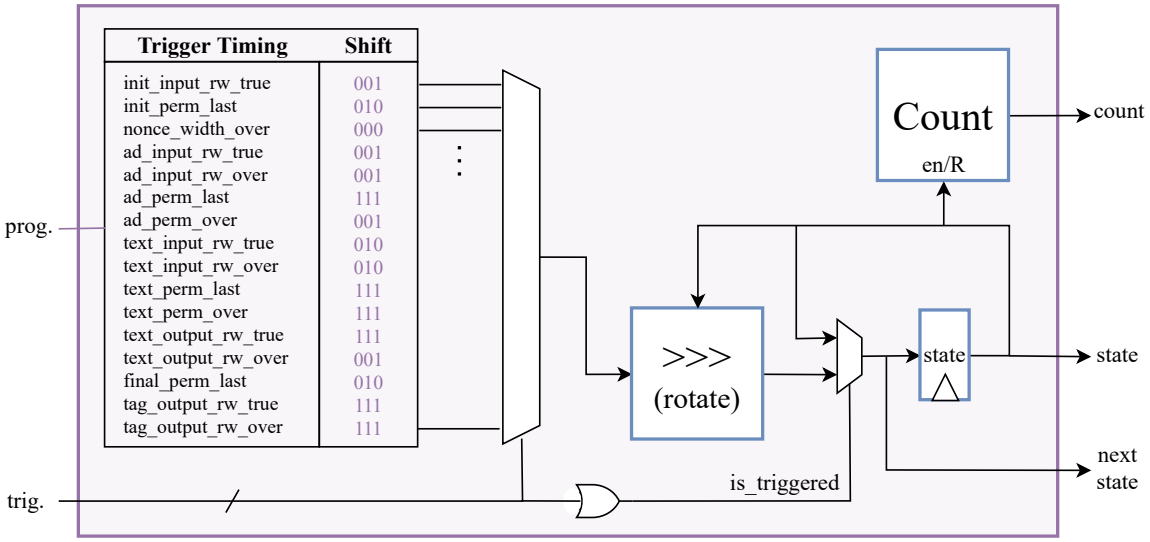


Figure 3.3. Controller Block Diagram.

The underlying permutation is marked *not shared* across algorithms. It needs to be reconfigured at the runtime using FPGA resources (see Section 2.3) to support multiple algorithms. For the remainder of the computing logic, we capitalize on the fact that there is still some shared logic for different ciphers, thanks to the similarities in their operation flow.

3.1.3. Cipher

The cipher module block diagram is given in Figure 3.4. This module is the computation engine of the accelerator. It consists of internal state registers, a permutation module, an event generator, a counter, and the connection logic including various multiplexers.

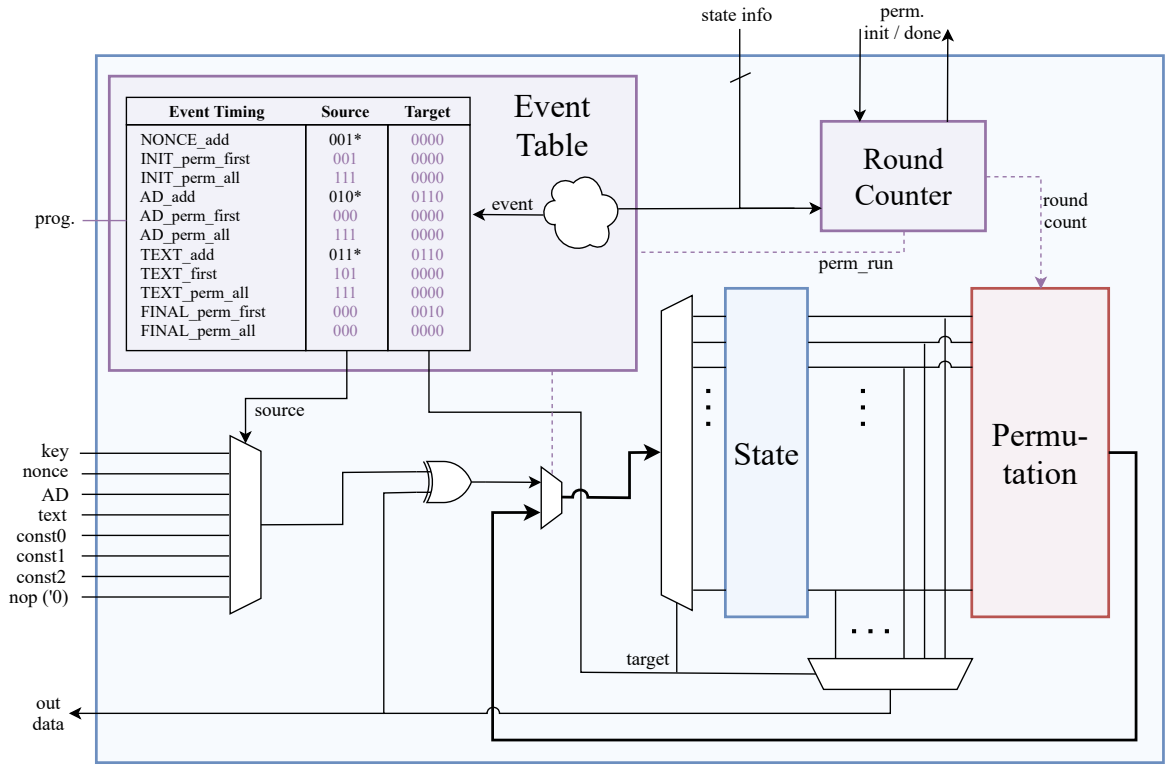


Figure 3.4. Cipher Module Block Diagram.

3.1.3.1. Permutation. This is the only compute-intensive part of the accelerator. It is also the module that is subjected to partial reconfiguration. The main idea is that by dynamically reconfiguring this part alone into an LWC permutation hardware, we are able to obtain an accelerator that is suitable for that particular algorithm. This allows the entire permutation round (or, depending on the logic depth, multiple rounds) to be completed in a single cycle, which is a massive improvement compared to using multiple generic micro-operations.

As explained in Section 2.3, partial reconfiguration requires module I/O's to be fixed. The permutation module inputs are the internal state registers, current round count, and an optional mode input to allow different modes of operations within one configuration. The output of the permutation is written back to the state registers if the permutation is running.

3.1.3.2. Event Table. For any given LWC algorithm, we define the addition or absorption of any input or a constant value as an *event*. Our observation is that for the majority of the finalists in the NIST LWC competition, the entire algorithm definition can be expressed as a combination of *events* and *permutation invocations*. Furthermore, each *event* can be uniquely characterized by its *event timing* and its *event action*, where an *event action* is XORing some value (any one of the data inputs, or a constant) to a specified portion of the internal state. Note that *event timings* depend strictly on state transitions and can be calculated by a fixed logic. Consequently, we can characterize all events of an LWC algorithm by programming source and target registers for all possible *event timings*, which are less than a dozen. The event table is the piece of hardware that keep the information of *event actions*. The table within the block diagram depicted in Figure 3.4 is programmed for Ascon as an example.

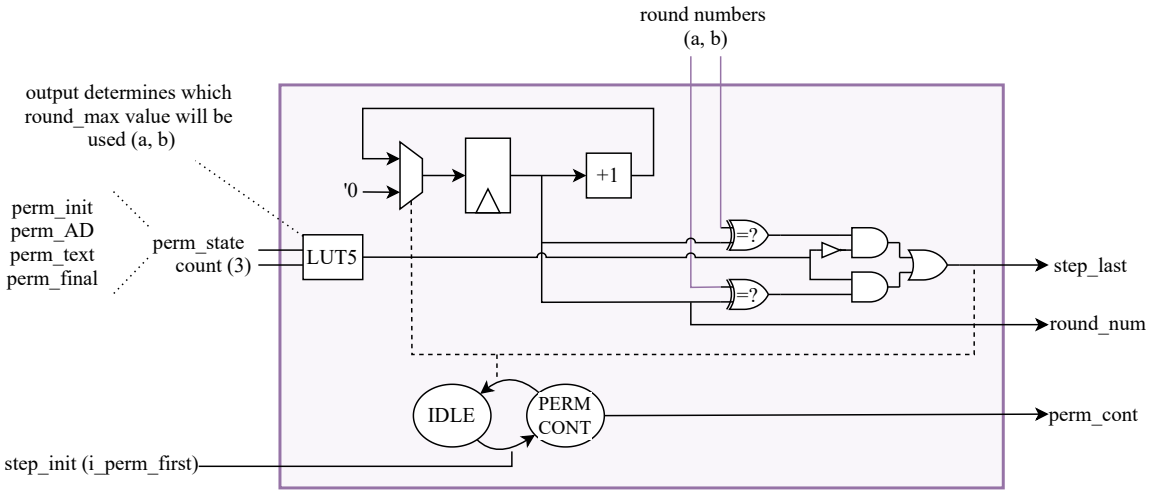


Figure 3.5. Round Counter Block Diagram.

3.1.3.3. Round Counter. Finally, the cipher includes a counter to keep track of rounds processed by the permutation. It starts counting when initialized by the controller and causes the state to be updated until the target number of rounds is reached. This total round number is a or b, depending on the current state information. When the permutation is complete, the controller is notified, and the counter is reset. The round counter's block diagram is shown in Figure 3.5.

3.1.4. Direct Memory Access Interface

Direct Memory Access (DMA) Interface is a separate module added between the system bus and the LWC accelerator. The motivation for this module is to reduce the indirection in data movement. The input data moves from the memory to the CPU, and then to the accelerator, which is also in a memory-mapped region. In addition, the instructions to carry this data movement also indirectly causes memory accesses themselves. The DMA module integrates the accelerator to the system bus as a master (see Section 3.3), allowing it to access the main memory directly, saving a significant amount of time and energy.

The CPU writes the start addresses of AD, plaintext, and the ciphertext in the main memory. The DMA controller keeps track of the internal state of the accelerator, and sends read/write commands to the accelerator whenever it is awaiting, reads/writes to the memory, and increments the address after each memory access. The effect of DMA in terms of performance and resource utilization is discussed in Chapter 5. The key and nonce inputs of the cipher should be provided to the accelerator as usual without the use of a DMA.

3.2. Hardware-Software Co-Design Approach

For this approach, we try to balance the workloads of hardware and software in the acceleration of the LWC algorithm. Our observation is that the previous design allocated a considerable amount of hardware resources, even outside the reconfigurable region, to allow flexibility on the controller.

Instead of programming some memory-mapped registers in obscure ways to describe a control flow, we propose using a set of instructions tailored for this purpose. The algorithm is described in this instruction set, and the binary is placed into any readable memory region. The accelerator may begin operating by executing instructions starting from a specified address. This allows us to handle the control flow in

software without occupying the CPU and reduces the costly message traffic between the processor core and the accelerator.

A formal description of our ISA for this accelerator is made in Section 3.2.1, accompanied by the design rationale and use cases of the instructions. Section 3.2.2 explains the hardware resources that execute these instructions to realize LWC algorithms.

3.2.1. ISA Specification

The instruction set defined in this section consists of eight 32-bit instructions, classified as memory access instructions, immediate instructions, control flow instructions, and a permutation instruction. Note that the accelerator does not have traditional general-purpose registers. Instead, each 32-bit portion of the state is treated as a separate register when reading from and writing, avoiding loss of time and energy due to internal data transfers. This requires some internal resources to be allocated for control flow variables that normally reside in a software-accessible register file. These resources are explained in Section 3.2.2.

3.2.1.1. Memory Access Instructions. The data transfer between the accelerator and the memory is carried out by the memory access instructions.The instruction set defines three such instructions: Add load value (ADD), write load value (WRT), and store read value (RD) as shown in Figures 3.6, 3.7, and 3.8.

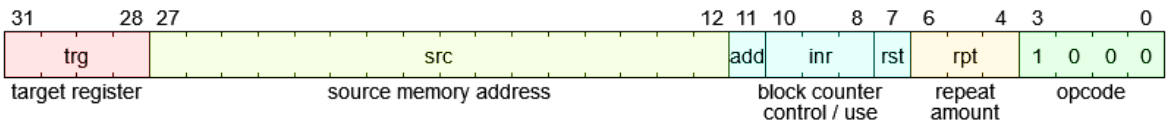


Figure 3.6. Add Load Value (ADD) Instruction.

The ADD instruction reads the memory address specified by *src*, XORs the 32-bit value read from memory with the target (*trg*) register, and writes the result to the same target register. If the *add* flag is set to 1, the value of the *block counter* is added to the *src* field to calculate the memory read address. If the *inr* field is set to a non-zero value, *block counter* is incremented by *inr* amount when the execution of the instruction is completed. If the *rst* bit is set, the *block counter* is reset to zero instead. Finally, the repeat amount uses a *subblock counter* to additionally repeat the instruction by *rpt* amount, incrementing the *src* and decrementing the *trg* value after each repetition.

This instruction is useful for the addition/absorption of cipher inputs, such as the AD or the message. The *block counter* and *repeat* mechanisms allow the reading and processing of successive words from the memory without calling a new instruction. Repeat mechanisms are particularly useful when the algorithm needs to digest more than one word at a time, for example, for reading four consecutive memory addresses for the 128-bit AD block of Ascon. The *block counter*, on the other hand, is often used when the instruction is nesting in a loop (see 3.2.1.3). It allows the same instruction binary to target vastly different memory addresses by an accumulating value, which is mostly helpful for incrementally targeting different input data blocks.

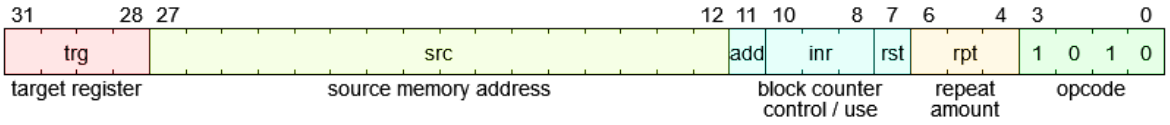


Figure 3.7. Write Load Value (WRT) Instruction.

WRT instruction operates similar to the ADD instruction, but the value read from the memory is directly written to the target register. This is useful when setting initialization vectors or decryption cases where the ciphertext input overwrites some state words. The *block counter* and *repeat* bitfields are identical to those of the ADD instruction.

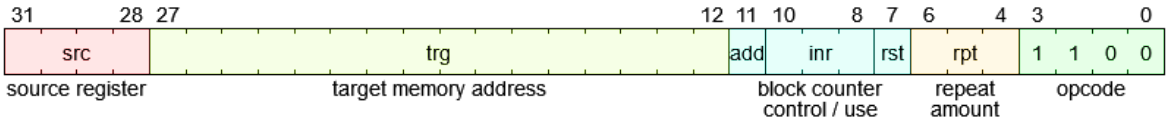
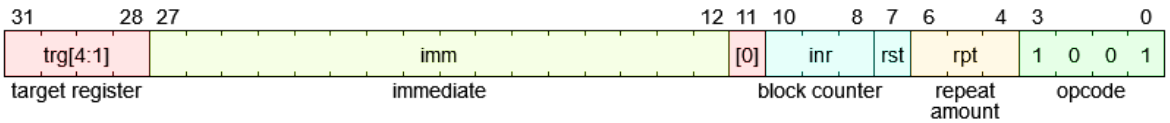


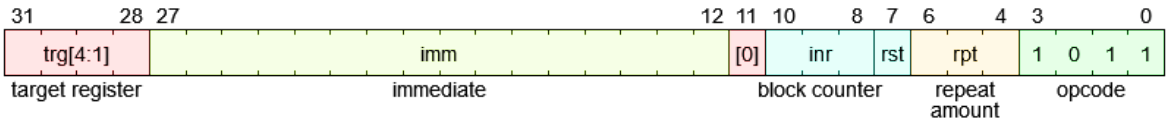
Figure 3.8. Store Read Value (RD) Instruction.

RD instruction reads the *src* register and writes the result into the *trg* memory address. The control flow is identical to ADD and WRT instructions.

3.2.1.2. Immediate Instructions. Instructions in this class, add immediate (ADDI) and write immediate (WRTI), are shown in Figure 3.9.



(a)



(b)

Figure 3.9. (a) Add Immediate (ADDI). (b) Write Immediate (WRTI).

ADDI and WRTI instructions add and overwrite the 16-bit immediate value in the instruction (*imm*) to the target register (*trg*), respectively. ADDI instruction is useful for constant additions to the state (such as the domain separation in Ascon), whereas WRTI is more frequently used for initialization purposes.

The *inr* and *rst* bitfields can be used to modify the block counter without causing any memory accesses, and the repeat amount is useful for applying the same operation to multiple registers. Because the immediate values are 16-bits, immediate instructions

modify the registers in a half-word granularity. Target register bitfield (trg) has an extra bit on the least-significant end to allow separate accesses to the lower and upper half-words of a register.

3.2.1.3. Control Flow Instructions. This class of instructions include branch (BRN) and branch-immediate (BRNI), displayed in Figure 3.10.

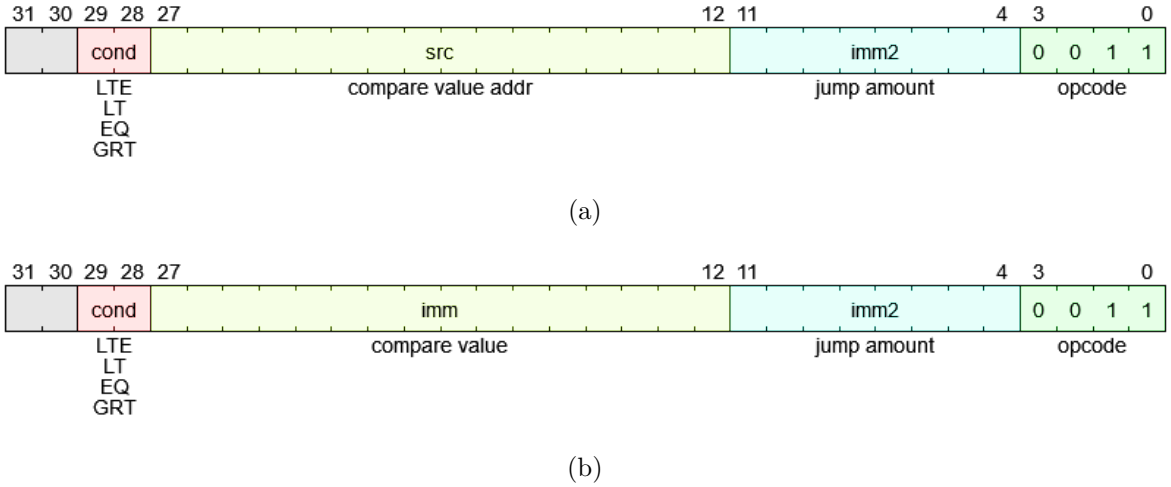


Figure 3.10. (a) Branch (BRN). (b) Branch-Immediate (BRNI).

BRN reads the source memory address (src) and compares the value with the block counter. If the condition evaluates to true, the program counter is incremented by $imm2$ (sign-extended). Possible conditions are: Less than or equal to (LTE), less than (LT), equal to (EQ), and greater than (GRT); and the block counter value is always the first operand. BRNI instruction uses the 16-bit immediate value embedded into the instruction binary for comparison instead of reading the value from the main memory. Both instructions can be used for conditionally executing or repeating certain portions of the code. BRN instruction is particularly useful when the repetition amount is a variable that is not deterministic at the programming time (for example, AD or message length) or for fixed values that require more than 16-bits. On the other hand, BRNI is faster for a fixed number of iterations or simple jumps because it does not require a memory access.

3.2.1.4. Permutation Instruction. Permutation instruction (PERM) shown in Figure 3.11 runs the permutation block in the cipher.

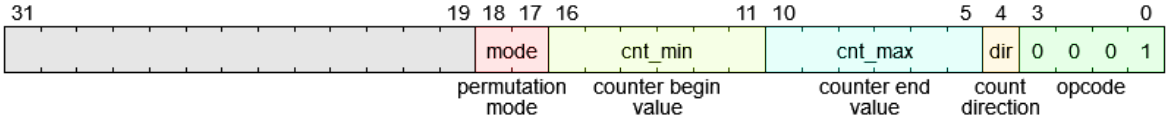


Figure 3.11. Permutation (PERM) Instruction.

PERM instruction updates the internal state by executing the permutation and writing the result back to the state. This is repeated for the number of rounds specified in the instruction, starting with *cnt_min* and finishing when the round counter equals *cnt_max*. The *dir* flag indicates the direction of the count. The *mode* bitfield offers flexibility in the use of the permutation block. It is used to select different modes of the same permutation on execution time without requiring dynamic reconfiguration. Alternatively, if the algorithm requires another computational block in addition to the permutation (such as the linear function in PHOTON-Beetle, explained in 2.2.3.1, it can be implemented into the permutation block as a new *mode* (see Section 3.2.2).

3.2.2. Top-Level Design

The accelerator's block diagram is depicted in Figure 3.12. As mentioned earlier, there is no general-purpose register file. The data transfers are efficiently performed directly into the state registers of the cipher and constitute no complications. However, the lack of a general-purpose register file necessitates the placement of specialized hardware in order to store different pointer values and loop variables. These variables are, thankfully, limited in number and consistent across all LWC algorithms. We determine three loop variables that need to be placed in order to describe an LWC algorithm: round number, block number, and subblock number. Each one of these variables is maintained using a dedicated counter within the controller, as explained in Section 3.2.2.1.

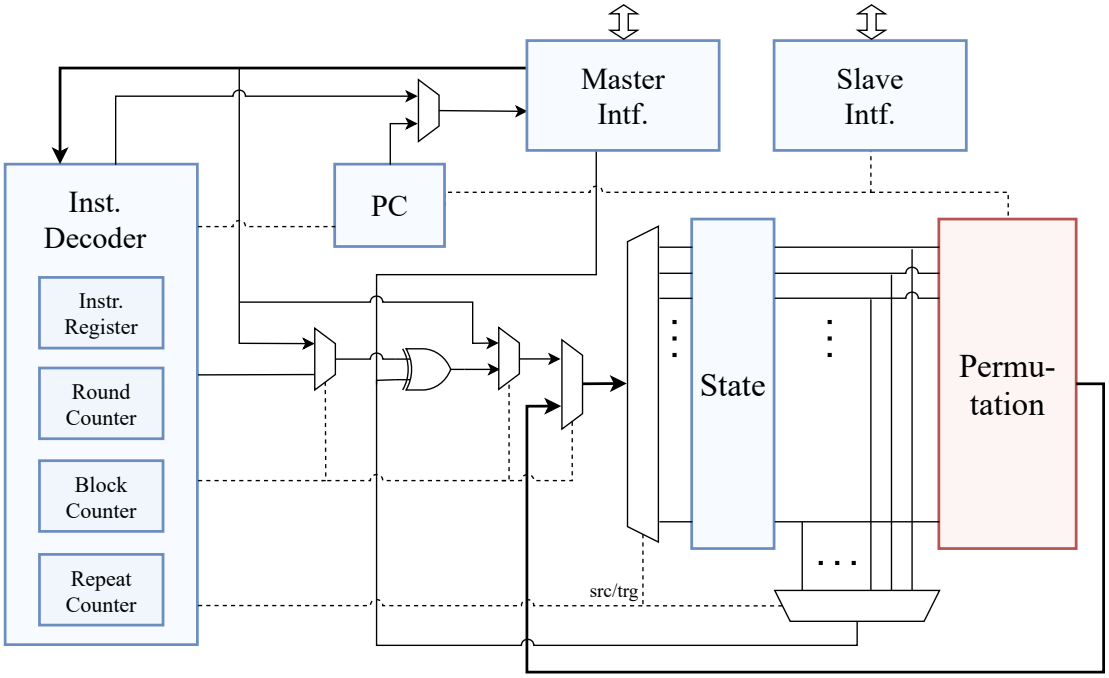


Figure 3.12. Top-Level Block Diagram of the Accelerator.

For address pointers, the accelerator has three memory-mapped registers that can be written to by the processor via the slave interface:

- Instruction start address register.
- Data read base address register.
- Data write base address register.

Base address registers only keep the upper half of the data addresses. The lower half of the data location in memory is indicated by the *src* and/or *trg* bitfields of the memory access instructions (3.2.1.1). When all three of these registers are written, the accelerator begins operation by reading instructions from the start address register.

3.2.2.1. Instruction Decoder. The instruction decoder is the main controller of the accelerator. It consists of decoder logic, a 32-bit instruction register, and various counters:

- Decoder logic: Decodes the instruction and generates various control signals for the accelerator. Controls the data flow and handles memory transactions through the master interface if the instruction requires memory access.
- Instruction register: Stores the instruction word received from the memory until the execution is completed.
- Round counter: Counts the permutation rounds based on the minimum and maximum counter values specified by the PERM instruction.
- Block counter: Keeps a counter value that is optionally incremented with memory access and immediate instructions. This value is used as a loop variable in programming, allowing branches using BRN(I) (see 3.2.1.3). It is also added to the memory address during read/write operations, allowing adjacent addresses to be accessed with the same instruction. The block counter does not reset unless an instruction explicitly makes this call using the rst flag.
- Repeat counter: Counts the number of times the current instruction is repeated, and alerts the controller when the repetition is completed. Used by memory access (Section 3.2.1.1) and immediate (Section 3.2.1.2) instructions. Repeat counter value is added to the address during memory accesses.

3.3. System-on-Chip Integration

We propose a simple, lightweight SoC structure for the complete design, as shown in Figure 3.13. The SoC consists of a lightweight processor core, an I/O device peripheral, a general-purpose main memory, two Read-Only Memories (ROM), a ROM programmer module to allow software updates, an ICAP controller to handle DPSR, and one of the proposed LWC accelerators. Our structure is based on PicoSoC, an open-source SoC equipped with PicoRV32 CPU [29]. The SRAM and Universal Asynchronous Receiver Transmitter (UART) modules are taken from the PicoRV32 repository. Our additions to this SoC structure include the accelerator, the boot ROM, the DPSR ROM, ROM programmer, ICAP controller, and modifications to the system bus (a PicoRV32-native interface) to enable multi-master support, allowing the accelerator to access memory without needing the CPU.

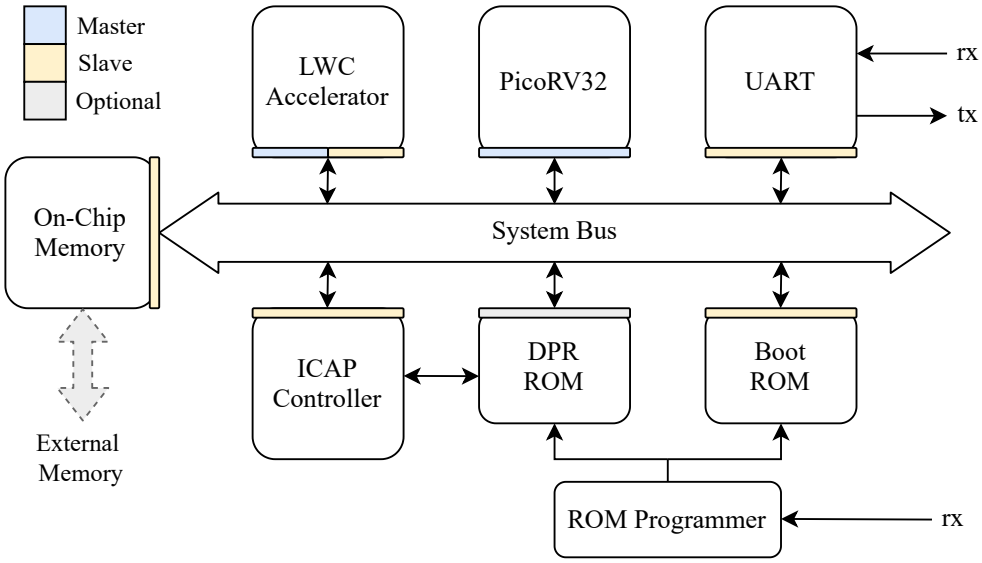


Figure 3.13. Top Level Connections.

3.3.1. PicoRV32

PicoRV32 is a free and open-source CPU core provided under the ISC license [29]. It implements the RV32IMC instruction set. It is intended for use as an auxiliary processor in FPGA and ASIC designs, with a tiny hardware footprint and a configurable native memory interface. Due to its high maximum frequency, it can be included in the majority of current designs without crossing clock domains. It can also be optimized for power consumption when run at lower frequencies, thanks to the ease of timing closure. Although its Cycles per Instruction (CPI) average is roughly 4, this core is suitable for lightweight environments that do not require significant computing power. The core is supplied with various configurable parameters, including an optional co-processor interface. Nevertheless, we avoid using such core-specific structures in order to maintain generality.

3.3.2. ICAP Controller

As explained in Section 2.3.1, we will be using ICAP to accomplish DPSR. Vivado Design Suite provides an ICAP controller IP that can be incorporated into Register-

Transfer Level (RTL) designs. However, several works in the literature demonstrated that it is possible to construct a substantially quicker and more efficient ICAP controller by manually instantiating the hardware primitive (ICAPE2) in a custom controller [30, 31]. Our ICAP controller permits the bitstream to be written at the theoretical maximum throughput of 32 bits per cycle.

Typically, reading from or writing to ICAP is accomplished by issuing a series of commands to configuration registers of the FPGA. Specific packet formats used in communication to the configuration registers are explained in [32]. Usually, a transaction starts with bus width detection and synchronization words, continues by issuing read/write commands, and finishes with a termination of communication using a desynchronization command. In the case of reconfiguring a region in FPGA fabric, write instructions involve a number of operations, including setting configuration options and frame registers, writing the configuration binary, and performing a Cyclic-Redundancy Check (CRC) to validate the correctness of the written configuration.

We design a simple ICAP controller with two modes. In the single-transaction mode, the processor issues a simple read or write request to a memory location dedicated to ICAP Controller. The least significant bits of the address are used to determine the target configuration register. A read/write operation is carried out by executing the steps explained in Figure 3.14. The timing of the ICAP inputs (chip enable, read/write select, and 32-bit data) are managed by a finite state machine in hardware, and a response is returned when the ICAP output is valid.

The stream mode is more straightforward in its operation and is used to partially reconfigure the FPGA. The processor initiates reconfiguration by writing the bitstream length, and the ICAP controller starts writing the data in the DPSR memory to the ICAP as a stream. The partial bitstream already contains all the necessary commands in the correct sequence to complete a transaction, so no further control is required.

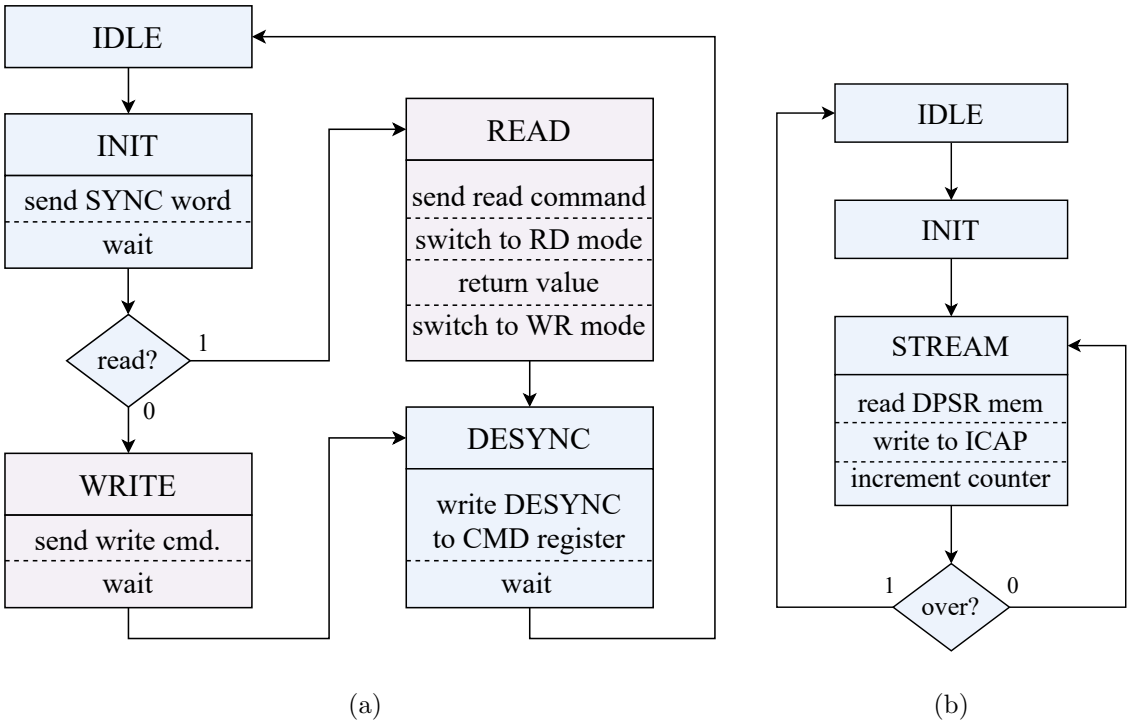


Figure 3.14. ICAP Controller Diagram (a) Single Mode, (b) Stream Mode.

The implementation of DPSR flow is explained in detail in Section 4.1.1.

3.3.3. ROM Programmer

This module is used as the boot mechanism of the SoC. It uses a UART interface to receive data from outside of the SoC and update Boot Memory and the DPSR Memory. These modules are read-only from the CPU's perspective, but they are implemented as BRAMs in FPGA and can be updated. ROM Programmer consists of a simple FSM that constantly checks the value of the serial input. If the pre-designated input sequence is detected, the succeeding data is written to either one of the read-only memories. On the software side, we use a simple python script to start the transaction by sending the ASCII sequence "LWC THESIS", then read binary data from a file and stream it through the serial interface.

The boot mechanism is not integral to the engineering purpose of our SoC; therefore, it can be safely replaced with any other boot mechanism such as SPI flash during the design time.

3.4. Software Support

A number of elements are needed on the software side of the stack to make this SoC work in a user-friendly manner. These allow software written in C language to be compiled with a RISC-V compiler in such a way that it utilizes different components on the SoC:

- A linker script that describes memory regions, their origin and lengths, permissions, and various attributes.
- A start assembly code to handle the system initialization and pointer assignments based on memory regions.
- UART driver containing the C functions required for serial terminal communication with the SoC. These primarily consist of print, scan, and datatype conversion (AXII hexadecimal string to integer, or vice versa) functions.
- LWC drivers that enable the utilization of accelerators from the software. It comprises small C functions that initiate reads/writes to memory-accessible registers and more extensive functions that execute the complete encryption/decryption process by calling these smaller functions in correct succession.

```
int crypto_aead_encrypt(
  unsigned char *c, unsigned long long *clen,
  const unsigned char *m, unsigned long long mlen,
  const unsigned char *ad, unsigned long long adlen,
  const unsigned char *nsec,
  const unsigned char *npub,
  const unsigned char *k);
```

Figure 3.15. C Wrapper For AEAD Modes of LWC Algorithms.

The top functions for the LWC drivers are formatted identically to the AEAD encryption wrapper provided by the NIST specification shown in Figure 3.15. This allows any previous software using LWC functions to be migrated into the accelerator use without causing any complications. The drivers are presented separately for different LWC algorithms. The addition of a new algorithm will require drivers to be re-written. As discussed earlier, this process is simple and extremely intuitive for the codesign approach, thanks to the ISA definition. On the other hand, the fully-hardware solution will require some hand-calculated values to be entered into a few registers. Still, we argue that this is a negligible non-recurring engineering effort compared to the hardware design.

4. IMPLEMENTATION OF LWC ACCELERATORS

The RTL design of the lightweight SoC with hardware acceleration is carried out using the Verilog and SystemVerilog hardware description languages. Xilinx Vivado Design Suite (v2020.1) is utilized for the implementation. The following section explains various implementation steps when converting the RTL codes into a functioning design in the FPGA, including the partial reconfiguration flow for dynamically switching between different ciphers.

4.1. FPGA Implementation

The hardware target for the implementation is Nexys A7 board, featuring 7-series programmable logic Artix-7 XC7A100T. The complete implementation setup consists of RTL design files, test benches, and a simple constraints file which includes clock and IO constraints. The clock frequency is set to 50 MHz for the SoC, and it is generated in the RTL from the 100 MHz internal clock of the FPGA by a simple clock divider. The frequency is kept at a reasonable but low value in order to allow tool optimizations to focus on reducing area and energy consumption without being tightly constrained by the timing budget. We differentiate our implementation flow from Vivado Design Suite's usual GUI-based bitstream creation to allow partial reconfiguration. A detailed explanation of the implementation flow with partial reconfiguration is provided in Section 4.1.1.

In this thesis, we make numerous implementations on slightly different SoC structures to enable the comparison of various use cases of LWC algorithms and isolate the benefits of proposed architectures. The implemented versions of the SoC are:

- Software-only: This is the SoC structure with no hardware acceleration. It is used as a baseline for evaluating the benefits of the accelerators in the following SoC versions. LWC algorithms are executed on software using PicoRV32.

- Dedicated accelerator: This SoC version includes a non-configurable hardware accelerator for a single LWC algorithm. This implementation is exclusively carried for Ascon, TinyJAMBU, and PHOTON-Beetle. The same accelerator structure is implemented without running the partial reconfiguration flow for all three algorithms. Partially shared portions of accelerators are also fixed for their algorithm-specific values (such as the program registers or event table in Section 3.1.1.3) to allow further optimizations. The comparison of this version with later ones reveals the true overhead of supporting multiple algorithms, covering both design and implementation aspects.
- Multiplexed-accelerator: This is the SoC version that can accelerate all three LWC algorithms without partial reconfiguration. Instead of designating the permutation block for partial reconfiguration, this design instantiates permutations from all three algorithms and allows their selection via multiplexers. The inclusion of this variant aims to articulate the tradeoff of having all permutation modes present in hardware at all times instead of using partial reconfiguration.
- Reconfigured accelerator: This is the final SoC structure we propose in this thesis, including the PicoRV32 CPU, an on-chip memory, two ROMs, UART device, one of the proposed LWC accelerators, and an ICAP module with DMA functionality. Allows acceleration of all three LWC algorithms by utilizing DPSR on the permutation block.

All three variants except the software-only mode are replicated for both accelerators that were proposed in Section 3. Other parameters are kept consistent across implementations to maintain fairness. The sizes for the main memory and the boot ROM are fixed at 16 kB, which is sufficient to accommodate the software implementation with the largest code size among the three algorithms.

4.1.1. Partial Reconfiguration Flow

The modification of the Vivado design flow for partial reconfiguration is outlined in Figure 4.1. It begins with hierarchically distinguishing the parts that are intended to be reconfigurable and contain them into a reconfigurable module. Initially, this module is designated as a black-box, and the design is synthesized. This step produces a netlist (.dcp) for the static region.

Next, variants of the reconfigurable module (modes) are subjected to out-of-context synthesis individually. This is a method used for synthesizing sub-modules of a design and allows post-synthesis netlists to be generated without the presence of IO buffers. The Partially-Reconfigurable Region (PRR) is then floorplanned by designating a p-block into the reconfigurable module. The mode with the highest resource utilization should be taken into consideration when designating a p-block. It is crucial to note that the partial reconfiguration bitstream size will be directly proportional to the size of the PRR, meaning larger PRRs require more time and energy to reconfigure and more memory space to store. For the FPGA we use, a PRR needs to have the height of at least one clock region in the FPGA layout. We designate a p-block that contains 400 slices, hence 1600 LUTs for the permutation module.

A combination of the static region and a valid mode for all PRR regions is referred to as a *configuration*. Since we only have a single PRR region, the number of configurations is equal to the number of modes. To generate a valid configuration, the netlist of the static region is opened in a Vivado project, and the netlist of one of the modes is imported to replace the black-box. We designate the module as reconfigurable, and the combined netlist is subjected to Place-and-Route (PnR) steps, going through timing closure. The obtained post-PnR netlist is our first configuration.

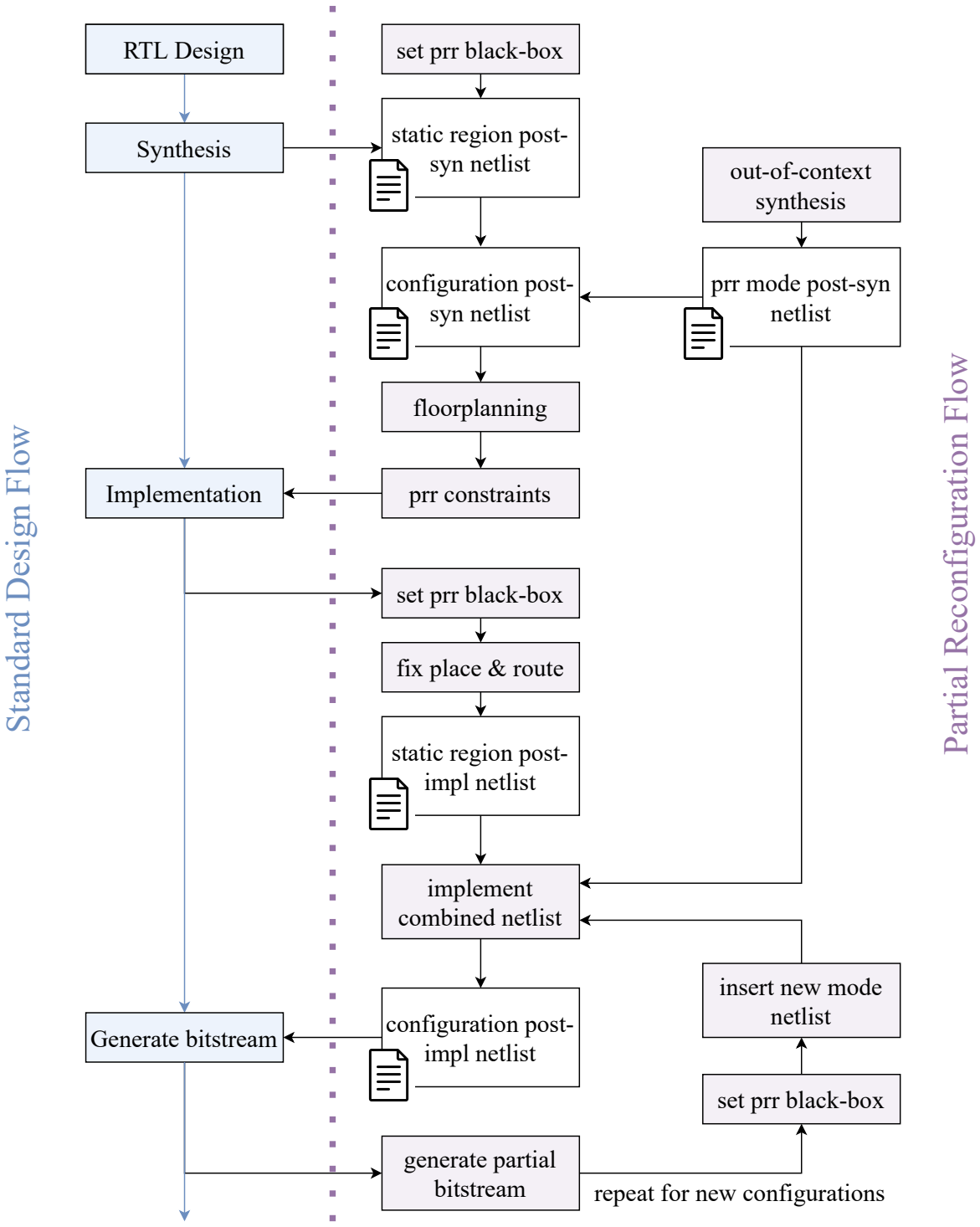


Figure 4.1. Design flow for partial reconfiguration.

To allow the implementation of other nodes, PRR is once again set as a black-box, this time on the post-PnR netlist. The resulting post-PnR netlist of the static region

is locked in the routing level, meaning any subsequent implementation runs will not modify the placement and routing of the static region's hardware resources. The black-box may now be replaced by a post-synthesis netlist of one of the modes. Running PnR on the combined netlist generates a valid configuration, as long as the new mode can be routed to meet the timing constraints. Figure 4.2 displays the FPGA layout at this step of the flow, where the orange cells are the fixed static region, and the PRR region is depicted with purple boundaries. Finally, we generate FPGA programming bitstreams for all configurations and partial bitstreams for all subsequent modes. A complete bitstream is only needed when programming the FPGA for the first time because the partial reconfiguration only needs the partial bitstreams.

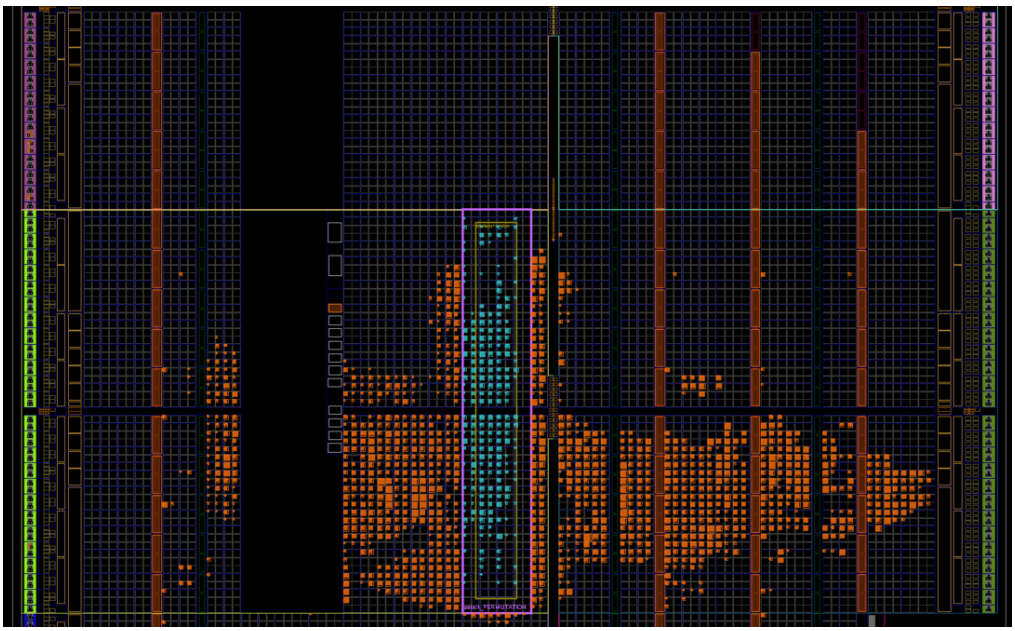


Figure 4.2. Post-Implementation FPGA Layout.

In order to dynamically reconfigure PRR, the partial bitstream should be written to the ICAP primitive. First, we transfer the partial bitstream into the DPSR memory using the ROM programmer module explained in 3.3.3. This process is analogous to remotely making a software update to the SoC, but does not halt the CPU operation. The CPU then starts DPSR by writing to the ICAP controller, which directly accesses the DPSR data to reconfigure the PRR using the stream mode (see 3.3.2).

4.2. Design Verification

For the verification of the implemented LWC algorithms, we use *cryptotvgen*. It is a suite of python scripts designed to iteratively run C implementations of the LWC algorithms with different inputs, write the input/output values of the runs to text files in a specific format, and compare them with the outputs of the NIST-recommended hardware API [11]. We use cryptotvgen solely for the generation Known-Answer Tests (KAT).

There are two steps to the verification process: behavioral verification and hardware verification.

4.2.1. Behavioral Verification

The behavioral verification consists of running a behavioral full-system simulation over a SystemVerilog testbench. As previously stated, both LWC accelerator drivers and reference C implementations of algorithms have the same input/output (IO) structure in which all inputs ($key, nonce, ad, message$) are supplied as char pointers, and their sizes ($mlen, adlen$) as integers. For simulation, we initialize input values into fixed memory addresses and pass these addresses into the functions as C pointers to execute encryption. This aims to imitate a real-world scenario where the inputs are obtained as a consequence of some computation or retrieved from some peripheral device and saved into the memory prior to the encryption. This approach also eliminates the overhead of writing data into memory at each program startup, allowing us to focus our benchmarks solely on the LWC algorithm execution.

We separately compile C codes using different drivers and the reference implementations, using the software setup explained in Section 3.4. The outputting program code is used to initialize boot ROM in hexadecimal format. We use a custom SystemVerilog testbench to run encryption using accelerators with different input values. The correctness of the outputs are verified by comparing them with KAT values.

4.2.2. Hardware Verification

The hardware verification is done by loading the bitstream into the Nexys A7 Board. The rx and tx pins of the SoC are assigned to the FPGA pmod IO using a constraints file (.xdc), and the UART peripheral is used for external communications. The UART baud rate is set to 115200, the maximum value supported by our setup in default configurations. An FTDI cable is used to establish a connection between the USB port of the computer and the serial IO of the FPGA. We use a serial terminal to access the processor via this link. UART driver functions allow us to control the processor through this terminal and observe the contents of any readable memory location.

We use slightly modified versions of LWC accelerator drivers, which allow the algorithm inputs to be taken from the terminal as hexadecimal strings. The hardware tests are performed manually by writing all input values from the terminal or passing locations (pointers) of the initialized values in the main memory into the driver. The contents of the output location are printed out on the terminal and compared with KAT results.

Note that the hardware implementation is made for verification and demonstration purposes and is not used for benchmarking. Accurate real-time power consumption measurements often require specialized hardware, such as the SAKURA-G board [33] or the products from the NewAE ChipWhisperer series [34], which are widely adopted for the generation of power traces and side-channel analyses. Newer FPGA boards, such as the Xilinx Ultrascale series, also employ a dedicated Power Management Bus for this purpose [35], which is not the case in our FPGA. In addition, the performance is impacted significantly by the communication overhead in a real-world scenario. Not only is the data transfer through the UART substantially slower than the operation of the CPU or the accelerator, but there are also intermediate steps that degrade performance, such as prints, scans, and various data conversion functions used for ease of communication. As explained in the following section, performance and power mea-

surements are carried out using post-implementation simulations on Vivado Design Suite.

4.3. Power Analysis

For power assessments, we utilize the power analysis tool in Vivado Design Suite. To increase accuracy, Vivado allows the incorporation of a Switching Activity Interchange Format (SAIF) file into its power report functionality. A SAIF file is exported after a simulation and used to estimate the power consumption of the design for that particular test case. It contains toggling counts of the underlying signals and their timing information like the time spent on logic high, logic low, or unknown states. SAIF files can be generated for specific timing intervals during the simulation, which allows us to investigate the power consumption of the design when different hardware components are engaged in operations. The power consumption values for various architectures and test cases presented in Chapter 5 are obtained using individual SAIF files and simulation runs for each configuration. Only the dynamic power consumption is considered, as the device static depends more on the FPGA chip than the implemented design.

5. EXPERIMENTS AND RESULTS

In this chapter, we compare the various designs described in Section 4.1. In each scenario, we utilize a single execution of an LWC algorithm as a benchmark. As comparative metrics, we use hardware area, code size, throughput, and energy per bit. The timing and power consumption information displayed is generated from the Vivado simulator, as explained in Section 4.3. The area and resource utilization values are obtained from the Vivado implementation reports. Code size refers to the size of the encryption function in memory, excluding firmware related to the rest of the SoC. The partial bitstream size is also included in the DPSR variant.

The test cases used during comparisons are selected to be similar to the NIST benchmarks [15]. For each algorithm, we use three distinct message lengths for encryption: A short message containing 16 bytes of plaintext and AD, a medium-sized message containing 64 bytes of plaintext and AD, and a long message with 1536 bytes of plaintext and AD.

5.1. Benchmarking Results

This section compares all four implementations (see 4.1) for three LWC algorithms. Among these implementations, CPU is the implementation of the *PicoRV32* processor without hardware acceleration. *Dedicated accelerator* (DA) refers to the acceleration of a single algorithm. *Multiplexed* (MUX) refers to the implementation variant where all three permutations are simultaneously present in hardware and can be selected at runtime. Finally, DPSR refers to the mode in which the permutation blocks can be reconfigured at runtime. Finally, we repeat the comparisons for both accelerator designs in Chapter 3. The hardware approach is referred to as versionH, whereas the mixed approach (HW-SW codesign) is referred to as versionM.

Note that the results reported in this section should not be used to compare algorithms with each other but rather to compare different SoC versions and accelerator design approaches. This is because the designs are not made and optimized for individual algorithms but for the entire SoC. For example, a single round of Ascon permutation requires substantially less timing and power budget than a round of PHOTON, but they are both limited by a critical path in the SoC. Similarly, a significant amount of time is spent between data transfers, which may conceal the performance differences between individual accelerators.

For a comparison of hardware performance between algorithms that uses different implementations, including several rolled versions of permutations, refer to NIST Benchmarking Reports [14, 15].

Table 5.1. Progressive Comparison of ASCON Implementation versionH.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	4029	5998	4569
Area (FFs)	-	1416	2571	2768	2884
Code Size (bytes)	-	40996	624	728	210388
Throughput (Mbit/s)	Short	0.402	5.267	4.547	4.547
Energy/bit (nJ/bit)		84.46	3.796	4.618	4.178
Throughput (Mbit/s)	Medium	0.661	8.974	8.386	8.386
Energy/bit (nJ/bit)		51.26	2.228	2.742	2.265
Throughput (Mbit/s)	Long	0.837	11.63	11.59	11.59
Energy/bit (nJ/bit)		40.60	1.890	2.156	1.181

Table 5.1 compares various implementations of versionH accelerator on Ascon, from least developed to most complex. It can be observed that resource utilization rises as complexity increases in general. The MUX version of the SoC uses more resources than the DPSR version, meaning that the additional permutations in the MUX version are significantly more hardware-costly than the ICAP controller and the

DPSR memory added in the DPSR version. The multiplexed variant can support three LWC algorithms for the cost of a 50% increase in LUT count and less than 10% increase in Flip-Flops compared to the DA implementation. It is significantly more efficient than having three separate accelerators, thanks to the reusability of common hardware provided by our design methodology.

Comparing the DA and DPSR versions reveals that supporting DPSR incurs a hardware overhead of approximately 10% in terms of LUT and FF count. This includes DPSR memory, the ICAP controller, their bus connections, and any additional resource overhead that the DFX flow may have produced. While this is a relatively small price, one must consider the significant overhead in code size (memory requirement) required to store the partial bitstream.

In terms of throughput and energy efficiency, accelerator versions provide 10 to 20 times improvement for medium and long-sized messages, as anticipated from specialized hardware. Even for the minor message and AD sizes of 16 bytes, hardware acceleration is at least an order of magnitude faster and more power-efficient than the CPU. The DA version has a slightly higher throughput for short and medium-sized messages because it takes fewer instructions to set up the accelerator (due to the lack of various program registers), but this advantage diminishes for longer messages. The MUX variant has the highest power consumption between the accelerators due to switching caused by unused permutations' hardware. It is possible to prevent this unnecessary logic switching by integrating additional hardware for masking. There is no significant difference in power consumption between the DA and DPSR modes in this version as the DPSR version also only has a single permutation active at a time.

Tables 5.2 and 5.3 present comparison of SoC versions for TinyJAMBU and PHOTON-Beetle respectively. We recognize that the majority of our observations regarding Ascon also apply to these two algorithms, with PHOTON-Beetle benefiting slightly more from hardware acceleration due to its computation-intensive round permutation.

Table 5.2. Progressive Comparison of TinyJAMBU Implementation versionH.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	3708	5998	4519
Area (FFs)	-	1416	2482	2768	2884
Code Size (bytes)	-	7630	724	828	210488
Throughput (Mbit/s)	Short	0.263	4.383	4.155	4.155
Energy/bit (nJ/bit)		125.0	4.790	7.046	4.616
Throughput (Mbit/s)	Medium	0.377	7.757	7.534	7.534
Energy/bit (nJ/bit)		87.52	2.707	3.981	2.521
Throughput (Mbit/s)	Long	0.436	10.27	10.26	10.26
Energy/bit (nJ/bit)		75.53	2.140	3.215	1.949

Table 5.3. Progressive Comparison of PHOTON-Beetle Implementation versionH.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	4452	5998	5223
Area (FFs)	-	1416	2567	2768	2884
Code Size (bytes)	-	13012	624	728	210388
Throughput (Mbit/s)	Short	0.068	5.099	4.555	4.555
Energy/bit (nJ/bit)		479.1	4.510	4.829	4.829
Throughput (Mbit/s)	Medium	0.092	8.819	8.386	8.386
Energy/bit (nJ/bit)		356.8	2.607	2.742	2.742
Throughput (Mbit/s)	Long	0.103	11.66	11.63	11.63
Energy/bit (nJ/bit)		317.6	2.229	2.235	2.149

In the case of PHOTON-Beetle, the overhead associated with adding support for various algorithms to the SoC (whether by multiplexing or DPSR) is substantially smaller. This is expected because it has the highest hardware footprint among all three algorithms. The power overhead for the multiplexed version is minimal in PHOTON-Beetle because the other two permutation blocks spend substantially less power than

PHOTON. Finally, the difference between DA and MUX versions is most remarkable in TinyJAMBU because it has the lowest-hardware dedicated implementation cost. Its high permutation round count also increases the power consumption in MUX versions by an excessive amount if the logic gates of the unused permutations are allowed to switch freely.

Next, we evaluate the performance of the accelerator versionM, starting with ASCON. We observe that the accelerator reduces the energy per bit processed by around 100 times for medium and long encryptions compared to the CPU-only implementation. Similarly, the throughput is increased by two orders of magnitude, which is significantly more improvement than versionH. Because the mixed accelerator has master access to the bus, a significant amount of time and energy is saved on the input/output data transfer. Additionally, it can operate without waiting for read-writes from the core, reducing idle time. The increase in energy efficiency and throughput as the message length increases is also more apparent in this version. The codesign approach (versionM) takes slightly longer to set up but is much more efficient once programmed.

We observe that almost all metrics follow a similar trend to versionH as the complexity increases. Multiplexed implementation has the highest resource utilization and spends significantly more power than the DA mode. DPSR mode introduces around 10% area overhead but requires more than 209 kB extra memory space compared to the DA version. All accelerator variants have more or less the same throughput for long messages.

One notable difference is that versionM presents a significant power consumption overhead in the case of DPSR, consuming more than twice as much energy as the dedicated accelerator version. Using a hierarchical power report, we confirm that this difference is, indeed, caused by the accelerator itself. It is not surprising that the increased complexity caused by routing a permutation to a pre-fixed static region results in increased power consumption, but the difference is more prominent in the mixed

Table 5.4. Progressive Comparison of ASCON Implementation versionM.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	3340	4925	3716
Area (FFs)	-	1416	2072	2072	2189
Code Size (bytes)	-	40996	392	392	210052
Throughput (Mbit/s)	Short	0.402	19.10	19.10	19.10
Energy/bit (nJ/bit)		84.46	1.099	1.570	2.407
Throughput (Mbit/s)	Medium	0.661	62.43	62.43	62.43
Energy/bit (nJ/bit)		51.26	0.352	0.608	0.768
Throughput (Mbit/s)	Long	0.837	219.6	219.6	219.6
Energy/bit (nJ/bit)		40.60	0.118	0.314	0.264

approach. A contributing factor is that in the mixed approach, the energy consumption is more dominantly determined by the accelerator, whereas the CPU and bus constitute a greater portion of total energy in the hardware-only accelerator. Additionally, the state registers of the cipher permutation are connected to the system bus to enable faster communication with the memory in the mixed approach. The same registers also serve as I/Os to the PRR region. It is possible that this overloading of internal state registers complicated placement and routing in the mixed approach, resulting in increased dynamic power.

Finally, it should be noted that the code sizes for the mixed version of the accelerator are smaller than the hardware-only version despite the adoption of a proprietary ISA.

Table 5.5. Progressive Comparison of TinyJAMBU Implementation versionM.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	3231	4925	3653
Area (FFs)	-	1416	2072	2072	2189
Code Size (bytes)	-	7630	416	416	210076
Throughput (Mbit/s)	Short	0.263	12.61	12.61	12.61
Energy/bit (nJ/bit)		125.0	1.506	3.647	1.823
Throughput (Mbit/s)	Medium	0.377	27.97	27.97	27.97
Energy/bit (nJ/bit)		87.52	0.714	2.180	0.893
Throughput (Mbit/s)	Long	0.436	45.76	45.76	45.76
Energy/bit (nJ/bit)		75.53	0.437	2.010	0.568

Table 5.6. Progressive Comparison of PHOTON-Beetle Implementation versionM.

	Msg. Len.	**CPU**	**DA**	**MUX**	**DPSR**
Area (LUTs)	-	2123	4079	4925	4283
Area (FFs)	-	1416	2072	2072	2189
Code Size (bytes)	-	13012	316	316	209976
Throughput (Mbit/s)	Short	0.068	23.27	23.27	23.27
Energy/bit (nJ/bit)		479.1	1.203	1.417	2.449
Throughput (Mbit/s)	Medium	0.092	70.62	70.62	70.62
Energy/bit (nJ/bit)		356.8	0.566	0.679	1.118
Throughput (Mbit/s)	Long	0.103	191.1	191.1	191.1
Energy/bit (nJ/bit)		317.6	0.355	0.455	0.685

Tables 5.5 and 5.6 demonstrate the results obtained using the algorithms TinyJAMBU and PHOTON-Beetle respectively. Their performances exhibit a similar pattern across versions. In TinyJAMBU, the power consumption of the multiplexed version is exceedingly high due to an increased number of rounds in permutation. As noted before, it can be reduced with the addition of masking logic. In contrast, the MUX

variant consumes less energy than the DPSR version in Photon-BEETLE because it has the most complex permutation block of all three.

5.2. Evaluation

In this section, we overview comparisons of the accelerator versions using the measurement results in the previous section. Figure 5.1 compares the average energy consumption of the fully-hardware (versionH) and mixed (versionM) accelerator architectures. Error bars indicate the least and most energy per bit values across algorithms.

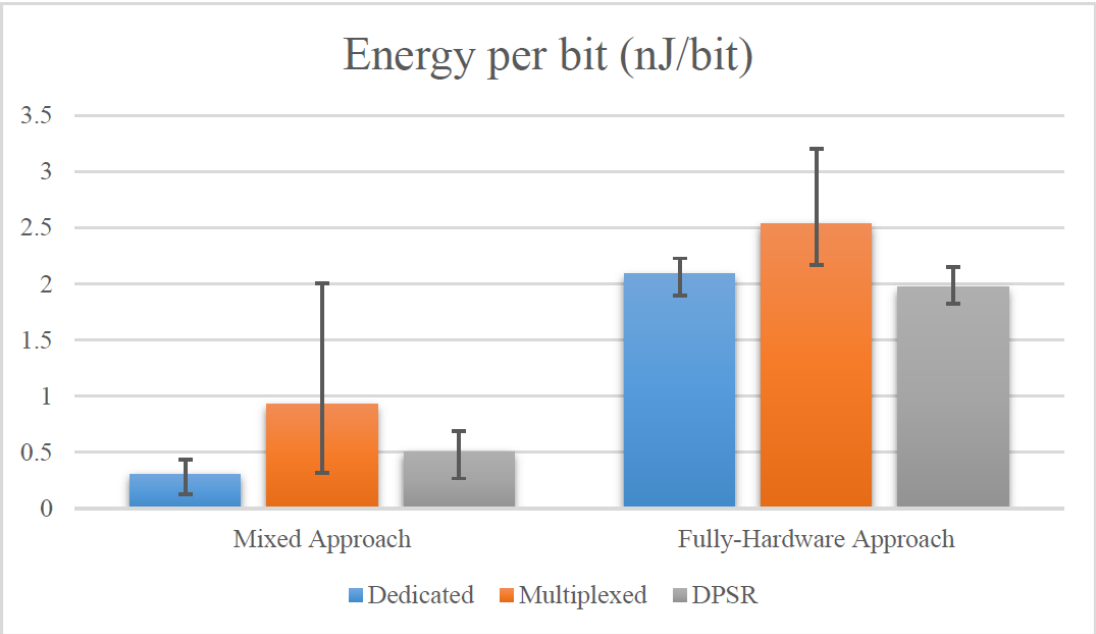


Figure 5.1. Energy efficiency comparison between versionM and versionH.

Evidently, the mixed approach is much more energy-efficient than the fully-hardware approach. We already mentioned that a significant portion of this advantage could be attributed to decreased data transfer overhead and reduced dependency on CPU triggers. Figure 5.2 compares the accelerator variants in terms of resource utilization. The provided values are for the complete SoC implementations. The CPU-only implementation of the SoC is included for reference.

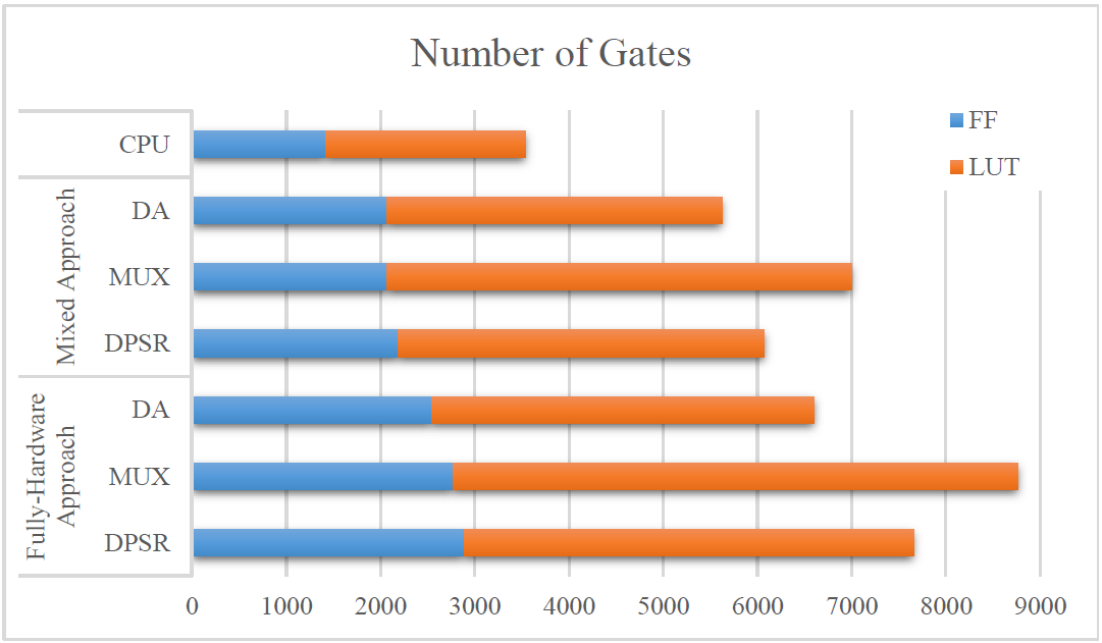


Figure 5.2. Resource utilization comparison between versionM and versionH.

The mixed approach uses significantly fewer resources than the fully-hardware approach. The information related to the control flow of the algorithm is kept in hardware registers (see Section 3.1.1.3) in versionH, which significantly increases the number of FFs. In contrast, this information is embedded into the program code in the mixed approach, which is stored in the main memory (BRAM) at no extra cost. The FF count in the mixed approach is further reduced because of its access to the memory, which allows the elimination of the memory-mapped IO registers by directly interacting with the state registers. The reduced number of registers automatically reduces the number of LUTs, because each register or register word is accompanied by LUTs that control read-writes to those registers. The LUT count has decreased further in the mixed approach thanks to its simple instruction decoder-based structure.

Next, we compare the dedicated, multiplexed, and reconfigurable accelerator implementations of the mixed approach. Figure 5.3 compares the energy efficiency of implementations ,and Figure 5.4 compares the hardware area. In this comparison, the MUX version is accompanied by a masking logic to reduce the power consumption.

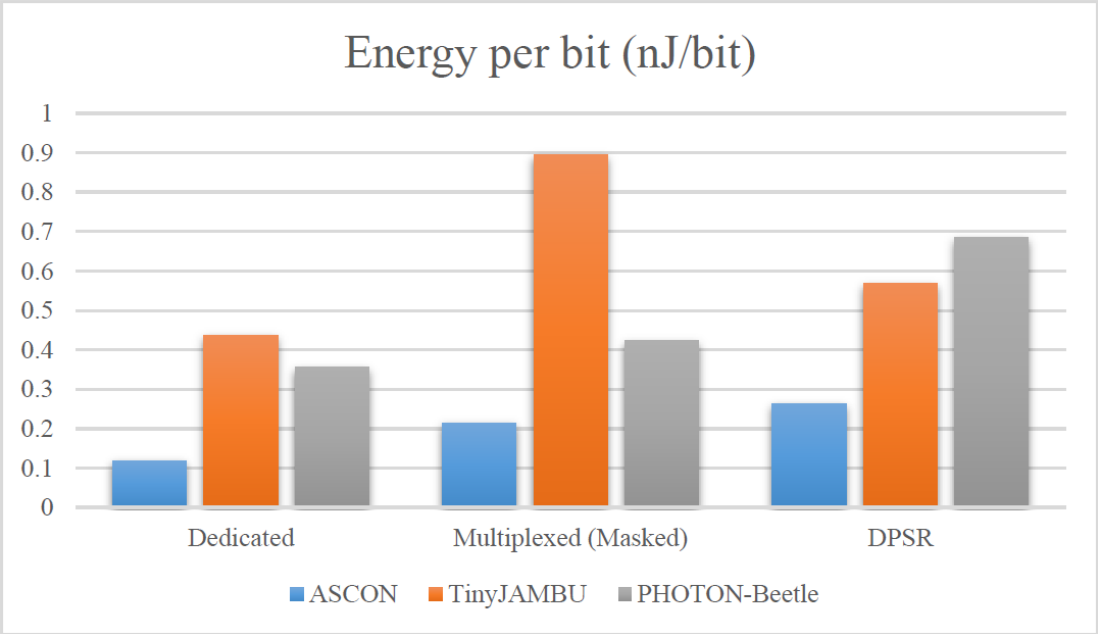


Figure 5.3. Energy efficiency comparison between DA, MUX, and DPSR implementations.

A dedicated accelerator is significantly more efficient regarding area and energy. The multiplexed mode also consumes substantially less power than the DPSR implementation when used with masking, although it is still not as low as a DA implementation. Nonetheless, it offers a significant improvement over having individual accelerators for algorithms and should be preferred if support for multiple algorithms is required. Note that the MUX approach is only viable because of the lightweight nature of algorithms: It would be troublesome to employ three hardware with implementation sizes similar to that of AES. Different parameter sets of LWC algorithms are supported by both DA and MUX versions, so long as the underlying permutation block remains unchanged.

The DPSR implementation provides maximum flexibility at the expense of energy efficiency. It permits the adjustment and updating of both algorithm parameters as well as the permutation, which may improve the life-cycle of a product after its deployment. Despite being outperformed by other implementations, DPSR mode is still orders of magnitude better than having no accelerator. It can also be preferred

over the multiplexed mode if the area constraint is severe. Additionally, the leftover area in the PRR in smaller algorithms can be used for performance improvement using aggressive parallelization. However, the overhead caused by the reconfiguration process should also be considered when using the DPSR approach, which is addressed in the following section.

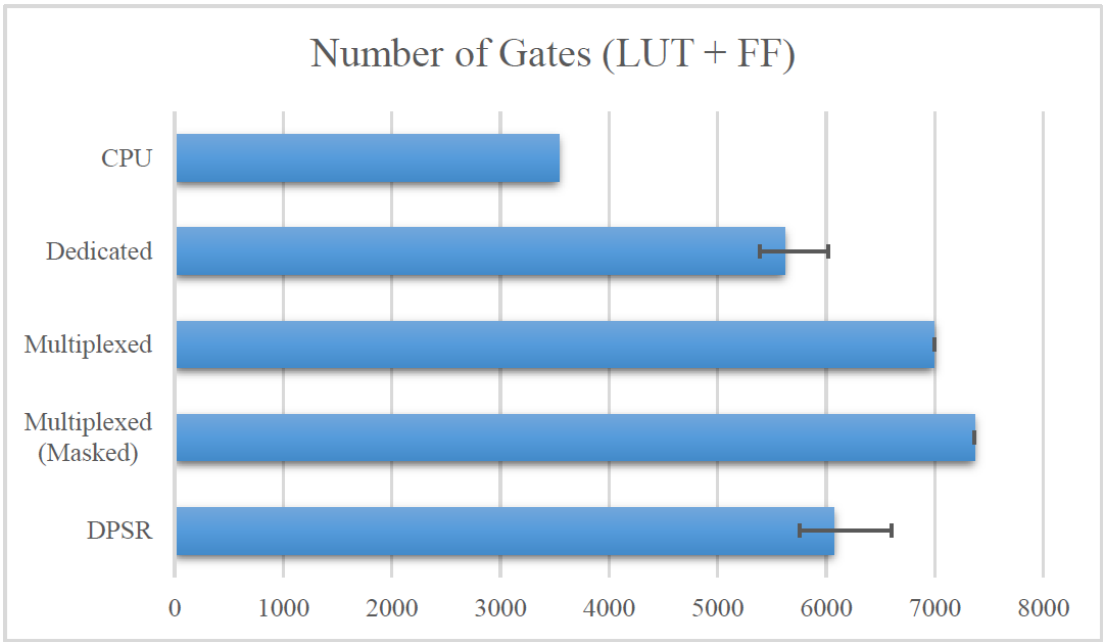


Figure 5.4. Resource utilization comparison between DA, MUX, and DPSR implementations.

5.3. DPSR Overhead

Comparisons should account for the time and energy necessary to reconfigure an FPGA fabric, which is the greatest drawback of the DPSR approach. In our case, the designated PRR p-block contains 400 slices, causing the bitstream size to be fixed at 205 kB. ICAPE2 primitive takes 32-bits at each cycle, so DPSR completes after 52415 cycles, taking slightly longer than 1ms at 50MHz frequency. The power consumption overhead is more tricky to compute because it cannot be directly extracted from the Vivado Simulator. While the simulator calculates the power consumption for moving

the data from BRAM to the ICAPE2 primitive, the power consumed by the logic that is being reconfigured needs to be estimated separately. For the estimation, we use the Medium-Grained Model provided in [36], and average out the power consumption for the duration of the reconfiguration is calculated as

$$P_{MG} = P_{FPGA} + P_{controller} + \frac{P_{before} + P_{after}}{2} \tag{5.1}$$

where P_{FPGA} is the idle power consumption of the FPGA, $P_{controller}$ is the power consumed by the ICAP controller, and P_{before} and P_{after} are the idle power consumption of the PRR before and after reconfiguration. The total on-chip power when the SoC is idle is 104 mW. From the hierarchical power reports, we obtain the value 20 mW for $P_{controller}$, and 1 mW for P_{before} and P_{after}. Low power consumption in the PRR region is expected because its inputs are connected to static registers, which are unchanged during DPSR. Other related works [37, 38] confirm that the power consumption overhead of the DPSR process does not exceed 20% on average.

The total energy consumption is calculated as $125mW \times 1048\mu s = 131\mu J$. Results presented in the previous section indicate that using the DPSR accelerator saves at least 40 nJ/bit energy compared to using the CPU, meaning single long encryption or a few medium-sized encryptions are enough to prefer running DPSR over the CPU. The difference in throughput is even more apparent: A medium-sized encryption takes longer than 1.5 μs for all algorithms, which is enough time to set up the reconfigurable region and use the accelerator. The difference between the CPU and the accelerators is so substantial that the overhead of the DPSR pales in comparison for most cases.

6. CONCLUSION

The spreading of IoT and small computing devices has caused constrained devices to take an active role in digital communications, hence cryptography. LWC emerged as a field in response to the demand for cryptographic primitives tailored for such devices. To solve the vast variance in solutions from academia and industry, NIST has initiated a standardization process: But it is still expected that multiple algorithms will be recommended for adoption to various environments. While this provides flexibility in optimization, it can also become a problem if the same agent is required to execute separate algorithms at different times.

In this thesis, we analyze and evaluate different methods of providing hardware acceleration for multiple cryptography algorithms in a single SoC. Our design methodology identifies the common components of different LWC algorithms and promotes resource sharing in hardware. Using this concept, two unique accelerator architectures are carried out for three LWC algorithms. Each combination is used on three distinct implementation variants (Dedicated, multiplexed, and reconfigurable accelerator) to generate an SoC. With the addition of the non-accelerated version, a total of 19 SoC implementations were compared. The designs are implemented on an Artix-7 XC7A100T FPGA with a clock frequency of 50 MHz, using the Vivado Design Suite. SAIF files generated from post-implementation timing simulations are utilized to increase the accuracy of power consumption reports. Each implementation is tested on three different encryption message and AD lengths, using the same metrics for evaluation with NIST benchmarking of hardware implementations.

We compare two accelerator design approaches under various conditions: The first is a hardware-only solution in which the accelerator is controlled by the CPU using the memory-mapped IO, program, and control registers. The second approach relies on hardware-software codesign and defines a custom ISA for the accelerator. Once initiated, the accelerator is capable of operating independently of the core. Our mea-

surements and analysis revealed that the mixed approach is significantly more efficient than the hardware approach when designing for flexibility. Not only is designing a reconfigurable hardware FSM costly, but it also relies on an obscure set of register writes to allow re-programming of the controller. On the other hand, a bespoke instruction set provides a simple and intuitive interface for programming. It is also substantially more versatile than the fully-hardware approach because its control flow is not limited to a set of pre-determined states and events. In fact, the mixed accelerator versions are so adaptable that they can be used to accelerate the hash modes of LWC algorithms despite the fact that they are initially designed for AEAD modes.

Next, we investigate the tradeoffs associated with supporting hardware acceleration for multiple algorithms in an SoC. We compare three accelerator modes in particular. The dedicated accelerator mode has a hardware accelerator for a single LWC algorithm. The multiplexed implementation also employs a single accelerator, but it can support three LWC algorithms using hardware multiplexing of the unshared resources. Finally, the DPSR version allows the dynamic modification of the entire accelerator, including the unshared permutation block.

The results indicate that the DA mode is the most efficient and should be preferred if the SoC will strictly utilize a single LWC algorithm in its lifetime. The MUX mode is significantly more area-efficient than having multiple accelerators but consumes slightly more power than a dedicated accelerator, even with masking. It should be considered if a few pre-determined LWC algorithms may need to be accelerated during the SoC operation. Lastly, the DPSR version of the system offers unequaled versatility. It enables the complete modification of accelerated algorithms using a similar method to provide a software update, which is a significant advancement in forward compatibility. It is less efficient if additional flexibility is not necessary; however, it may be preferable over the MUX version if the environment is severely area-constrained. The DPSR overhead should also be evaluated when reconfiguring the accelerator for a new algorithm.

Future work includes using this design methodology to develop side-channel resistant accelerators. Numerous side-channel countermeasures described in the literature rely on the adjustment of algorithm permutation hardware to conceal power usage, which DPSR can perform during the runtime. Another idea is to utilize the unused permutation hardware in the multiplexed version to generate noise, which, in turn, should obfuscate the power trace of the used algorithm, although it requires a thorough analysis.

REFERENCES

1. Easttom, C., *Modern Cryptography: Applied Mathematics for Encryption and Information Security*, McGraw-Hill Publishing, New York, 2015.

2. Petitcolas, F. A. P., "Kerckhoffs' Principle", H. C. A. van Tilborg and S. Jajodia (Editors), *Encyclopedia of Cryptography and Security*, p. 675, Springer, 2011.

3. Paulsen, C. and R. Byers, "Glossary of Key Information Security Terms", National Institute of Standards and Technology, 2018.

4. NIST FIPS PUB, "197: Advanced Encryption Standard (AES)", *Federal Information Processing Standards Publication*, Vol. 197, No. 441, p. 311, 2001.

5. Shannon, C. E., "Communication Theory of Secrecy Systems", *The Bell System Technical Journal*, Vol. 28, No. 4, pp. 656–715, 1949.

6. McKay, K., L. Bassham, M. Sönmez Turan and N. Mouha, "Report on Lightweight Cryptography", National Institute of Standards and Technology, 2016.

7. Rogaway, P., "Authenticated-Encryption with Associated-Data", *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 98–107, Washington D.C., USA, 2002.

8. Kehrer, P., "Authenticated Encryption", 2019, https://cryptography.io/en/latest/hazmat/primitives/aead/, accessed on June 15, 2022.

9. Black, J., "Authenticated Encryption", *Encyclopedia of Cryptography and Security*, Springer, 2005.

10. Turan, M. S., K. McKay, D. Chang, C. Calik, L. Bassham, J. Kang and J. Kelsey,

"Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process", *National Institute of Standards and Technology Internal Report*, Vol. 8369, No. 10.6028, 2021.

11. Kaps, J.-P., W. Diehl, M. Tempelmeier, E. Homsirikamol and K. Gaj, "Hardware API for Lightweight Cryptography", George Mason University, 2019.

12. Calik, C., M. Hasan and K. Jinkeon, "Benchmarking Round 2 Candidates on Microcontrollers", *NIST Lightweight Cryptography Workshop*, 2020.

13. Campos, F., L. Jellema, M. Lemmen, L. Müller, D. Sprenkels and B. Viguier, "Assembly or Optimized C for Lightweight Cryptography on RISC-V?", *International Conference on Cryptology and Network Security*, pp. 526–545, Springer, Vienna, Austria, 2020.

14. Mohajerani, K., R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps and K. Gaj, "FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results", Cryptology ePrint Archive, 2020, https://eprint.iacr.org/2020/1207, accessed in June 2022.

15. Mohajerani, K., R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps and K. Gaj, "Hardware Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process", *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 164–169, 2021.

16. Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Federal Information Processing Standards, National Institute of Standards and Technology, Gaithersburg, MD, 2015.

17. Guido, B., D. Joan, P. Michaël and V. Gilles, "Cryptographic Sponge Functions", Citeseer, 2011.

18. Bertoni, G., J. Daemen, M. Peeters and G. V. Assche, "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications", *International Workshop on Selected Areas in Cryptography*, pp. 320–337, Springer, Toronto, Canada, 2011.

19. Dobraunig, C., M. Eichlseder, F. Mendel and M. Schläffer, "Ascon v1.2", *Submission to the NIST Lightweight Cryptography Standardization Process*, 2021, https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf, accessed in June 2022.

20. Wu, H. and T. Huang, "TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms (version 2)", *Submission to the NIST Lightweight Cryptography Standardization Process*, 2021, https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf, accessed in June 2022.

21. Bao, Z., A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin and K. Yasuda, "PHOTON-Beetle Authenticated Encryption and Hash Family", *Submission to the NIST Lightweight Cryptography Standardization Process*, 2021, https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf, accessed in June 2022.

22. "UG909 Vivado Design Suite User Guide: Dynamic Function eXchange, version 2021.2", Xilinx, 2022.

23. UG-Partrecon, "Partial Reconfiguration IP Core", *Intel*, 2019, https://www.intel.com/content/www/us/en/docs/programmable/683404/current/partial-reconfiguration-ip-core.html, accessed on August 24, 2022.

24. Patel, S. and W. H. Wen-mei, "Accelerator Architectures", *IEEE Micro*, Vol. 28, No. 4, pp. 4–12, 2008.

25. Cota, E. G., P. Mantovani, G. Di Guglielmo and L. P. Carloni, "An Analysis of Accelerator Coupling in Heterogeneous Architectures", *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, San Francisco, California, USA, 2015.

26. Shao, Y. S. and D. Brooks, *Research Infrastructures for Hardware Accelerators*, Springer International Publishing, Cham, 2016.

27. "AMBA AXI and ACE Protocol Specification", ARM Cambridge, UK, 2011.

28. Cook, H. M., *Productive Design of Extensible On-Chip Memory Hierarchies*, University of California, Berkeley, 2016.

29. Clifford, W., "PicoRV32- A Size-Optimized RISC-V CPU", 2017, https://github.com/YosysHQ/picorv32, accessed on 24 August, 2022.

30. Vipin, K. and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq", *IEEE Embedded Systems Letters*, Vol. 6, No. 3, pp. 41–44, 2014.

31. Duhem, F., F. Muller and P. Lorenzini, "Farm: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA", *International Symposium on Applied Reconfigurable Computing*, pp. 253–260, Springer, Belfast, UK, 2011.

32. "7 Series FPGAs Configuration User Guide, version 1.13.1", Xilinx, 2018.

33. Guntur, H., J. Ishii and A. Satoh, "Side-Channel Attack User Reference Architecture Board SAKURA-G", *IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pp. 271–274, Tokyo, 2014.

34. O'flynn, C. and Z. D. Chen, "Chipwhisperer: An Open-Source Platform for Hardware Embedded Security Research", *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pp. 243–260, Springer, Paris, 2014.

35. "UltraScale Architecture System Monitor User Guide, version 1.10.1", Xilinx, 2021.

36. Bonamy, R., D. Chillet, S. Bilavarn and O. Sentieys, "Power Consumption Model for Partial and Dynamic Reconfiguration", *International Conference on Reconfigurable Computing and FPGAs*, pp. 1–8, Cancun, Mexico, 2012.

37. Nafkha, A. and Y. Louet, "Accurate Measurement of Power Consumption Overhead During FPGA Dynamic Partial Reconfiguration", *International Symposium on Wireless Communication Systems (ISWCS)*, pp. 586–591, Poznan, 2016.

38. Rihani, M. A., F. Nouvel, J.-C. Prévotet, M. Mroue, J. Lorandel and Y. Mohanna, "Dynamic and Partial Reconfiguration Power Consumption Runtime Measurements Analysis for ZYNQ SoC Devices", *International Symposium on Wireless Communication Systems (ISWCS)*, pp. 592–596, Poznan, 2016.

APPENDIX A: Partial Reconfiguration Flow Commands

Below are the tcl commands used in Vivado during the partial reconfiguration flow. The comments in the below code specify the functionality of respective code snippets, combined with some intermediate steps that are handled using the GUI.

```
synth_design
update_design -cell [get_cells coproc/PERMUTATION] -black_box
# save netlist after synthesizing (prr as black box)
write_checkpoint <prjdir>/dfx/netlists/static/syn.dcp

#synthesize desired mode(s) as out of context
synth_design -mode out_of_context -top permutation_reconfig
write_checkpoint \
<prjdir>/dfx/netlists/prr/<mode1>/permutation_reconfig.dcp

#PBLOCK properties
set_property RESET_AFTER_RECONFIG 1 [get_pblocks pblock_PERMUTATION]
set_property SNAPPING_MODE ON [get_pblocks pblock_PERMUTATION]

#open netlist of the static region.
#read a netlist of a prr mode, and combine it with current netlist.
read_checkpoint -cell [get_cells coproc/PERMUTATION] \
<prjdir>/dfx/netlists/prr/<mode1>/permutation_reconfig.dcp

#set the merged netlist reconfigurable
set_property HD.RECONFIGURABLE 1 [get_cells coproc/PERMUTATION]

# save combined netlist (optional)
write_checkpoint <prjdir>/dfx/netlists/config/syn_<mode1>.dcp
```

```
#run physical flow.
opt_design
place_design
route_design

#save combined routed netlist
write_checkpoint <prjdir>/dfx/netlists/config/impl_<mode1>.dcp
#write full bitstream
write_bitstream -raw_bitfile <prjdir>/dfx/bitstreams/<mode1>.bit

#remove the netlist of the prr block.
update_design -cell [get_cells coproc/PERMUTATION] -black_box
#save netlist of the remaining (static portion only).
lock_design -level routing
write_checkpoint <prjdir>/dfx/netlists/static/impl.dcp

#now we generate other configurations.
#<loop>
read_checkpoint -cell [get_cells coproc/PERMUTATION] \
<prjdir>/dfx/netlists/prr/<mode2>/permutation_reconfig.dcp
opt_design
place_design
route_design
write_checkpoint <prjdir>/dfx/netlists/config/impl_<mode2>.dcp
write_bitstream -raw_bitfile <prjdir>/dfx/bitstreams/<mode2>.bit
update_design -cell [get_cells coproc/PERMUTATION] -black_box
#</loop>
```