POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's Degree in Electronic Engineering

Compact Yet Fast: An Efficient d-Order Masked Implementation of Ascon

Supervisor:
Prof. Guido MASERA
Mattia MIRIGALDI

Candidate:
Nico PANINFORNI

Academic Year 2024/2025

Abstract

The proliferation of connected and resource constrained devices in everyday applications has made hardware security a critical requirement. Ensuring that cryptographic algorithms can resist physical attacks, such as side-channel analysis, is essential to guarantee trustworthy and privacy preserving communication in embedded systems. Side-channel attacks (SCA) represent a major threat to the secure deployment of cryptographic algorithms on embedded devices, with power analysis being particularly effective in extracting sensitive information from hardware implementations. Masking techniques are among the most widely adopted countermeasures, yet fully masked designs often incur significant area and latency overhead.

In this work, we present a generic side-channel protected design of Ascon, the NIST-selected lightweight cryptography standard, that achieves high efficiency by dynamically reconfiguring the hardware countermeasures during message processing. Exploiting Ascon's mode level structure, where bulk operations can be executed without full protection, we adopt a selective masking strategy, securing only the most critical phases (initialization and finalization), while accelerating unprotected bulk processing. To this end, we design a modified masking gadget with dual functionality: it acts as a countermeasure during sensitive operations and enables parallel processing paths for enhanced throughput during regular rounds.

Our architecture supports any configurable security order and instantiates only the minimum hardware resources needed to maximize throughput per round. We also evaluate an enhanced Ascon architecture based on the Changing of the Guards technique, which eliminates the need for fresh randomness. Security validation is performed using fixed-vs-random t-tests on both first- and second-order masked implementations.

Experimental results demonstrate that the proposed design achieves superior throughput-to-area ratios compared to state-of-the-art masked implementations, making it well suited for deployment in resource constrained environments where both performance and physical security are critical.

Index Terms—Ascon, Hardware, Side-channel attack, Domain- Oriented-Masking, Mode level implementation

“"It’s only through experiencing both victory and defeat, running wildly and weeping tears, that men truly grow. Don’t be afraid to cry". ”

Shanks

“Solo sperimentando sia la vittoria che la sconfitta, scappando e versando lacrime, un uomo diventerà un uomo. Non avere paura di piangere.”

Shanks

Contents

List of Tables

List of Figures

Acronyms

IoT	Internet of Things
SCA	Side-Channel Attack
NIST	National Institute of Standards and Technology
LWC	Lightweight Cryptography
AE	Authenticated Encryption
XOF	Extendable Output Function
AEAD	Authenticated Encryption with Associated Data
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
RFID	Radio Frequency Identification
TVLA	Test Vector Leakage Assessment
DOM	Domain-Oriented Masking
TI	Threshold Implementation
GF	Galois Field
FSM	Finite State Machine
MCU	Microcontroller Unit
RC	Round Constant
AAD	Associated Data
MSG	Message
LD	Linear Diffusion
MUX	Multiplexer
PLL	Phase-Locked Loop
IO	Input/Output

GE	Gate Equivalent
ASIC	Application-Specific Integrated Circuit
RTL	Register Transfer Level
SCALIB	Side-Channel Analysis LIBrary
PRNG	Pseudo-Random Number Generator
LFSR	Linear Feedback Shift Register
CW305	ChipWhisperer CW305 FPGA Board
SoA	State of the Art
TP/A	Throughput-to-Area
NDA	Non-Disclosure Agreement
MTU	Maximum Transmission Unit
MAC	Medium Access Control
UDP	User Datagram Protocol
IPv4	Internet Protocol version 4
LoRaWAN	Long Range Wide Area Network

1 Introduction

Motivation. The rapid proliferation of resource constrained devices, largely driven by the widespread adoption of the Internet of Things (IoT), has introduced new and pressing challenges in terms of security and privacy. Within the IoT-to-Cloud paradigm, devices typically perform partial local processing before encrypting and transmitting data to remote servers. As a consequence, cryptographic implementations must achieve an optimal balance: minimizing hardware footprint while sustaining sufficient throughput to enable the real-time encryption of substantial data volumes. At the same time, IoT devices often operate in physically exposed environments, rendering them particularly vulnerable to side-channel attacks (SCA), which exploit physical leakages—such as power consumption or electromagnetic emissions—to extract sensitive information. To mitigate such threats, cryptographic hardware must integrate dedicated countermeasures. Among the available techniques, *masking* has emerged as one of the most effective, providing provable resistance against side-channel leakage and offering scalability across different security levels.

1.1 Lightweight Cryptography: the Role of Ascon

To address the challenges posed by constrained devices, the National Institute of Standards and Technology (NIST) launched the Lightweight Cryptography (LWC) standardization process in 2015, with the goal of identifying algorithms capable of delivering essential security services while minimizing computational and memory requirements. After a multi-year evaluation, in February 2023 the **Ascon** family was selected as the winner of the LWC competition [1], thereby becoming the new standard for lightweight cryptographic applications.
Ascon is a versatile suite of cryptographic primitives that includes authenticated encryption (AE), hashing schemes, and extendable output functions (XOFs). This broad functionality makes it suitable for a wide range of applications, from embedded systems to high performance network protocols. A key feature of Ascon is its mode level design, which facilitates efficient implementations across both hardware and software platforms while also supporting physical protection mechanisms such as masking.

1.2 Authenticated Encryption and Standardization Efforts

Authenticated Encryption with Associated Data (AEAD) is a cryptographic primitive that simultaneously guarantees confidentiality and integrity. Unlike conventional encryption, which secures only message secrecy, AEAD additionally authenticates both the ciphertext and any unencrypted metadata. This dual functionality is essential in modern applications, particularly in constrained environments where robust yet efficient protection is required.
Two major international initiatives have shaped the AEAD landscape. The first was the **CAESAR competition** *(Competition for Authenticated Encryption: Security, Applicability, and Robustness)*, launched in 2013 to identify a portfolio of AEAD algorithms suitable for diverse use cases, from lightweight embedded devices to high throughput hardware accelerators. The second was the aforementioned **NIST LWC project**, which formally evaluated AEAD candidates (and optionally hashing functions) against four key criteria [2]:

1. **Security:** Strong resistance to cryptanalytic attacks and robustness under misuse conditions (e.g., nonce reuse).

2. **Implementation Efficiency:** Competitiveness on constrained platforms in terms of memory, energy, and area.

3. **Ease of Protection:** Support for efficient countermeasures against side-channel and fault attacks.

4. **Royalty-Free Availability:** Open and unrestricted adoption to maximize deployment.

Ascon emerged as one of the most promising candidates across both initiatives due to its balanced design philosophy. Not only does it achieve strong cryptographic security, but it is also tailored for efficient implementation and straightforward integration of countermeasures.

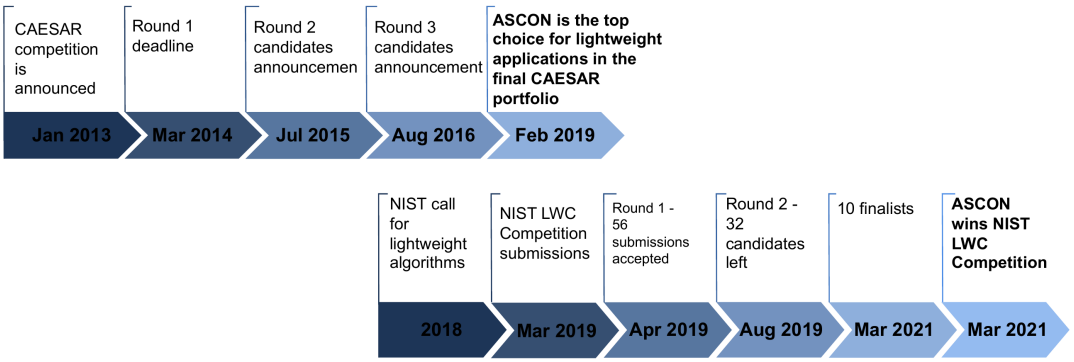


Figure 1.1: CAESAR and NIST LWC competitions timelines

1.3 Target Applications of Ascon

The principal application domains of AEAD schemes, and Ascon in particular, span a wide range of environments [3]:

- **High performance servers** and network infrastructures, where maximum throughput is required, often with hardware acceleration.
- **RFID tags**, especially passively powered ones, where ultra-low power and minimal area are critical due to energy harvesting constraints.
- **Wireless sensor nodes**, typically battery powered and long-lived, where energy efficiency is the dominant metric.
- **Embedded systems**, widespread in consumer and industrial markets, where efficiency is measured in throughput per unit of area.

Application	Optimization Goal	Interface
High performance computing	High throughput	Custom
RFID tags	Low power and low area	Custom
Wireless sensor nodes	Low energy	Memory-mapped
Embedded systems	Throughput per area (efficiency)	Memory-mapped

Table 1.1: Optimization goals for various authenticated encryption use cases

In addition to these performance driven constraints, modern cryptographic implementations must also address implementation level security. Side-channel resistance has thus become a mandatory requirement for practical deployments. Ascon's flexibility, combined with its ability to support efficient physical countermeasures such as masking, makes it a particularly strong candidate for secure and efficient use in embedded systems.

1.4 Thesis structure

This thesis has been developed with the aim of exploring a dedicated digital hardware architecture for the implementation of the ASCON algorithm, using the SystemVerilog hardware description language, the work focuses on the design and optimization of the ASCON structure, addressing both performance requirements and resistance against side-channel attacks.
After this introductory Chapter 1, the thesis is structured as follows:

- **Chapter 2** provides an overview of the state of the art, introducing the internal functioning of the ASCON algorithm and its main design principles.
- **Chapter 3** focuses on side-channel attacks (SCA), with particular attention to power analysis techniques and the Test Vector Leakage Assessment (TVLA) methodology for security evaluation.

- **Chapter 4** discusses masking countermeasures against SCA, with a dedicated focus on Domain-Oriented Masking (DOM) and its applicability to lightweight cryptographic designs.
- **Chapter 5** describes the proposed hardware core, detailing the design process, architectural optimizations, and the dedicated SystemVerilog implementation of the ASCON structure.
- **Chapter 6** presents the integration of the proposed design on FPGA, including the measurement setup and interfacing aspects. It also describes the experimental framework adopted for the security validation, with particular focus on the application of the TVLA methodology to assess the robustness of the implementation.
- **Chapter 7** discusses the main results of the thesis, highlighting performance metrics such as throughput and area occupation. Furthermore, it provides a comparison with state of the art implementations, emphasizing the advantages and trade offs of the proposed architecture.
- **Chapter 8** summarizes the main findings of the thesis, highlighting achieved results, limitations, and possible directions for future work.

2 ASCON

2.1 ASCON: Internal State and Mode of Operation

ASCON, as presented in the submission to NIST in [4] , is a cryptographic suite designed to provide authenticated encryption with associated data (AEAD) and hashing capabilities in low-power and resource constrained environments, such as embedded devices and IoT. The suite includes ASCON-128 and ASCON-128a for encryption, and ASCON-Hash and ASCON-Xof for fixed and extendable output hashing.
Both ASCON-128 and ASCON-128a offer 128-bit security and rely on a shared 320-bit permutation. This permutation, structured as a Substitution-Permutation Network (SPN), includes three steps per round: constant addition (pC), a nonlinear substitution layer (pS), and a linear diffusion layer (pL). A detailed breakdown of these transformation steps is provided later in chapter 2.3.
Authenticated encryption algorithms are typically defined by a set of parameters: the key length $k < 160$ bits, the data block size (also known as rate) r, and the number of internal rounds a and b. The encryption process can be represented as:

$$AE(K, N, A, P) = (C, T) \tag{2.1}$$

where:

- K is the secret key of k bits,
- N is the 128-bits nonce,
- A denotes the associated data (of arbitrary length),
- P is the plaintext message (also of arbitrary length).

The output of the encryption consist of:

- C, the ciphertext (same length as P),
- T, a 128-bits authentication tag.

The corresponding decryption process is defined as:

$$AD(K, N, A, C, T) \in \{P, \perp\} \tag{2.2}$$

where $\perp$ denotes a failed authentication. If the tag verification succeeds, the plaintext P is returned; otherwise, the output is $\perp$.

The Table 2.1 presents the reccommended parameter sets for authenticated encryption, as proposed in [4].

Name	**Bit size of**						**Rounds**	
	State (S)	**Rate** (S_r)	**Capacity** (S_c)	**Key** (K)	**Nonce** (N)	**Tag** (T)	p^a	p^b
ASCON-128	320	64	256	128	128	128	12	6
ASCON-128a	320	128	192	128	128	128	12	8

Table 2.1: Parameters for recommended authenticated encryption schemes

Compared to ASCON-128, ASCON-128a increases the data block size (rate) from 64 to 128 bits and increases the number of intermediate rounds from 6 to 8, resulting in higher throughput with only a marginal increase in hardware area.
All variants of the ASCON cipher suite operate on a 320-bit internal state S, which is iteratively updated using two permutations: p^a, applied for a rounds during initialization and finalization, and p^b, applied for b rounds during associated data and plaintext processing. The permutation based design offers several advantages [5], including a fixed and well defined state size, the elimination of complex key scheduling procedures, and low decryption overhead—since encryption and decryption rely on the same underlying permutation. The state S is logically divided into two parts: an **outer part** S_r of r bits, and an **inner part** S_c of $c = 320 - r$ bits, known as the capacity.
For implementations and round transformation purposes, the state S is represented as five 64-bit words x_0, x_1, x_2, x_3, x_4, such that:

$$S = S_r||S_c = x_0||x_1||x_2||x_3||x_4 \tag{2.3}$$

ASCON's design also accounts for physical attack resilience, particularly against side-channel attacks (SCA). Its logical simplicity and low algebraic complexity make it well suited for masked implementations with minimal overhead in both hardware area and power consumption. The protocol structure supports targeted protections such as masking only the initialization and finalization phases, while keeping the intermediate steps lightweight and efficient.
Thanks to this balance of security, efficiency, and modularity, ASCON is a robust and scalable solution for applications where lightweight devices must perform cryptographic operations securely and interact with high-performance systems.

2.2 ASCON: Authenticated Encryption

ASCON's AEAD mode is based on a duplex mode of operation, inspired by constructions like MonkeyDuplex [6]. In the duplex mode, data is absorbed into the state and then squeezed out (sponge structure). This eliminates the need for a separate key scheduling process, allowing for high-speed implementations and less memory requirements.
The encryption and decryption operation are illustrated in Figure 2.1 and specified in Algorithm 2.2.
The core idea is that the output of the previous permutation is reused: the outer

part (rate) is combined with the plaintext to produce the ciphertext and also serves as input for the next round, while the inner part (capacity) is carried over unchanged between rounds. This approach enables both absorption and output generation within a unified structure.
The encryption process in ASCON, shown in Figure 2.1a, consists of four main phases:

- **Initialization -** The initial state is built using key, nonce and initialization vector (predefined constant).
- **Associated Data Processing -** the associated data is absorbed into the state.
- **Plaintext Processing -** the plaintext is encrypted to generate ciphertext.
- **Finalization -** the tag is computed to ensure authenticity.

The decryption follows the same four steps, as shown in Figure 2.1b, with the plaintext recovered in the third phase by reversing the encryption logic.

Algorithm 1.2: Authenticated encryption and decryption procedures

Input:
key $K \in \{0,1\}^k,\ k \leq 160$
nonce $N \in \{0,1\}^{128}$
associated data $A \in \{0,1\}^*$
plaintext $P \in \{0,1\}^*$
Output:
ciphertext $C \in \{0,1\}^{|P|}$,
tag $T \in \{0,1\}^{128}$
Initialization
$S \leftarrow \mathbb{IV}_{k,r,a,b} \parallel K \parallel N$
$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$
Processing Associated Data
if $|A| > 0$ **then**
  Divide A into r-bit blocks $A_1 \dots A_s$ with padding
  for $i = 1$ **to** s **do**
    $S \leftarrow p^b((S_r \oplus A_i) \parallel S_c)$
  $S \leftarrow S \oplus (0^{319} \parallel 1)$
Processing Plaintext
Divide P into r-bit blocks $P_1 \dots P_t$ with padding
for $i = 1$ **to** $t-1$ **do**
  $S_r \leftarrow S_r \oplus P_i$
  $C_i \leftarrow S_r$
  $S \leftarrow p^b(S)$
  $S_r \leftarrow S_r \oplus P_t$
$\hat{C}_t \leftarrow (S_r)_{|P_t|} \bmod r$
Finalization
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{320-r-k}))$
$T \leftarrow (S)_{128} \oplus (K)_{128}$
return $C_1 || \dots || \hat{C}_t, T$

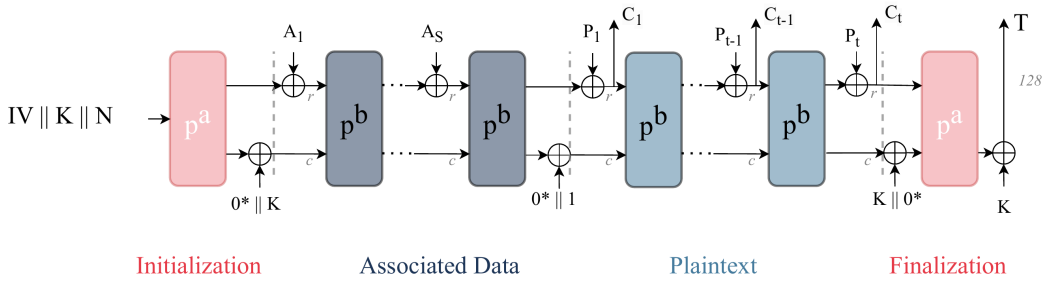
Input:
key $K \in \{0,1\}^k,\ k \leq 160$
nonce $N \in \{0,1\}^{128}$
associated data $A \in \{0,1\}^*$
ciphertext $C \in \{0,1\}^*$
tag $T \in \{0,1\}^{128}$
Output:
plaintext $P \in \{0,1\}^{|C|}$ or $\perp$
Initialization
$S \leftarrow \mathbb{IV}_{k,r,a,b} \parallel K \parallel N$
$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$
Processing Associated Data
if $|A| > 0$ **then**
  Divide A into r-bit blocks $A_1 \dots A_s$ with padding
  for $i = 1$ **to** s **do**
    $S \leftarrow p^b((S_r \oplus A_i) \parallel S_c)$
  $S \leftarrow S \oplus (0^{319} \parallel 1)$
Processing Ciphertext
Divide C into r-bit blocks $C_1 \dots \hat{C}_t$
for $i = 1$ **to** $t-1$ **do**
  $P_i \leftarrow S_r \oplus C_i$
  $S_c \leftarrow C_i$
  $S \leftarrow p^b(S)$
  $\hat{P}_t \leftarrow (S_r)_{|C_t|} \oplus \hat{C}_t$
$S_r \leftarrow S_r \oplus (P_t \parallel 1 \parallel 0^*)$
Finalization
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{320-r-k}))$
$T' \leftarrow (S)_{128} \oplus (K)_{128}$
if $T' = T$ **then**
  return $P_1 || \dots || \hat{P}_t$
else
  return $\perp$

Initialization

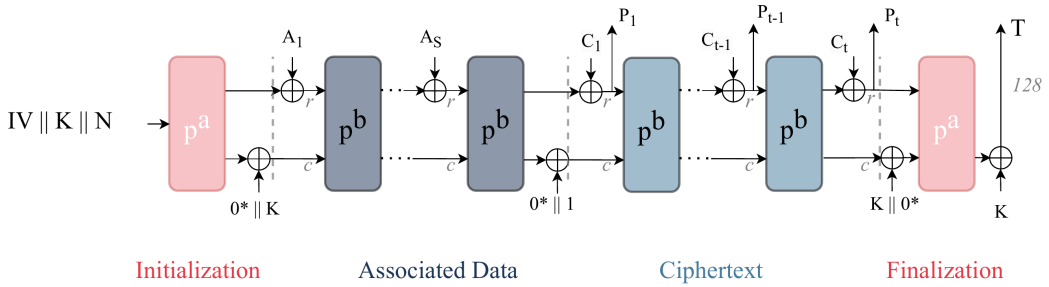
The initial 320-bit state S in ASCON is constructed by concatenating a predefined initialization vector (IV), the secret key K, and the 128-bit public nonce N:

$$S \leftarrow IV_{k,r,a,b} || K || N \tag{2.4}$$

The IV encodes key parameters of the algorithm-namely, the key length k, the rate r, the number of rounds a used in initialation and finalization phase, and the number



(a) Encryption



(b) Decryption

Figure 2.1: Authenticated encryption and decryption procedures

of intermediate rounds b. Each of these values is encoded as an 8-bit integer, and the remaining bits are padded with zeros to reach the required length. The IV values used for each variant are:

$$IV_{k,r,a,b} = \begin{cases} 80400c0600000000 & \text{for ASCON-128} \\ 80800c0800000000 & \text{for ASCON-128a} \\ a0400c06 & \text{for ASCON-80pq} \end{cases} \tag{2.5}$$

After setting the state, the permutation p^a is applied for a round:

$$S \leftarrow p^a(S) \tag{2.6}$$

Finally, the state is XORed with the secret key to complete the initialization:

$$S \leftarrow S \oplus (0^{320-k}||K) \tag{2.7}$$

This initialization ensure that the secret key is integrated into the state both before and after the initial permutation, increasing resistance against key recovery in case of partial state leakage.

Processing Associated Data ASCON processes the associated data A dividing it into s blocks of r bits each: $A_1, A_2, ..., A_s$. If the length of A is not already a multiple of r, the data is padded by appending a single '1' bit followed by the minimum number of '0' bits needed to reach the required block size. Formally, this yields:

$$A_1, ..., A_s \leftarrow \begin{cases} \text{r-bit blocks of } A||1||0^{r-1-(|A| mod\ r)} & \text{if } |A| > 0 \\ \varnothing & \text{if } |A| = 0 \end{cases} \tag{2.8}$$

Each block A_i is XORed with the outer part S_r of the internal state S, and the result is passed through the p^b permutation :

$$S \leftarrow p_b((S_r \oplus A_i)||S_c), \qquad \text{for } 1 \leq i \leq s \tag{2.9}$$

If no associated data is present ($|\mathrm{A}| = 0$), this step is skipped entirely. After all associated data blocks have been absorbed, a 1-bit domain separation constant is applied by XORing the internal state with a 320-bit vector ending in '1':

$$S \leftarrow S \oplus (0^{319}||1) \tag{2.10}$$

This domain separation ensures that the internal state reflects whether associated data was processed or not, which is crucial for the integrity guarantees provided by the cipher.

Processing Plaintext and Ciphertext

The plaintext P is processed in blocks of r bits, using the same padding rule applied to associated data. The resulting padded plaintext is then divided into t blocks $P_1, P_2, ..., P_t$:

$$P_1, ..., P_t \leftarrow \text{r-bits blocks of } P||1||0^{r-1-(|P| \ mod \ r)} \tag{2.11}$$

Encryption

For each block P_i (where $1 \leq 1 \leq t$), the encryption operation follows these steps:

1. The block is XORed with the first r bits S_r of the internal state:

$$C_i \leftarrow S_r \oplus P_i \tag{2.12}$$

2. For all blocks **except the last one**, the updated state is passed through the p^b permutation:

$$S \leftarrow p_b(C_i||S_c), \qquad \text{for } 1 \leq i < t \tag{2.13}$$

3. For the **last block**, the permutation is skipped, and the state is simply updated as:

$$S \leftarrow C_i||S_c \tag{2.14}$$

To ensure the ciphertext C has exactly the same length as the original plaintext P, the last ciphertext block C_i is truncated to match the length of the final unpadded fragment.

Decryption

Due to the inherent symmetry of the construction, decryption closely follows the encryption process. In practice, inverting the roles of ciphertext and plaintext blocks within each step is sufficient to recover the original message.

Finalization

In the finalization phase, the secret key K is XORed with the internal state S, and the result is processed through the p permutation using a round:

$$S \leftarrow p_a(S \oplus (0^r||K||0^{c-k})) \tag{2.15}$$

The authentication tag T is then computed by XORing the last 128 bits of the updated state with the last 128 bits of the key:

$$T \leftarrow S_{[319:192]} \oplus K_{[127:0]} \tag{2.16}$$

The encryption procedure outputs the final ciphertext $C_1||..||\tilde{C}_t$ along with the tag T.
During the encryption, the same operation are performed to recompute the tag.
The plaintext $P_1||...||\tilde{P}_t$ is returned only if the recomputed tag matches the received tag; otherwise, the output is rejected to preserve authenticity.

2.3 ASCON: permutation

The core of the ASCON family algorithms including ASCON-128 is built upon two 320-bit permutations: p^a and p^b. These permutations apply a variable number of rounds of a common transformation p, which is structured as a Substitution-Permutation Network (SPN). Each round consists of three sequential steps:

$$p = p_L \circ p_S \circ p_C \tag{2.17}$$

where:

- p_C is the **constant addition** layer,
- p_S is the **substitution** (non-linear) layer,
- p_L is the **linear diffusion** layer

1. Constant Addition Layer (p_C)
In this step, a round dependent constant c_r is XORed with the register word x_2. The round index r is computed differently depending on the permutation used:

- For p^a : $r = i$
- For p^b : $r = i + a - b$

The specific values of the c_r for different combinstions of a and b are reported in Table2.2. The core operation performed in this layer is:

$$x_2 \leftarrow x_2 \oplus c_r \tag{2.18}$$

$p^{a=12}$	$p^{b=8}$	$p^{b=6}$	Constant c_r
0	-	-	0xf0
1	-	-	0xe1
2	-	-	0xd2
3	-	-	0xc3
4	0	-	0xb4
5	1	-	0xa5
6	2	0	0x96
7	3	1	0x87
8	4	2	0x78
9	5	3	0x69
10	6	4	0x5a
11	7	5	0x4b

Table 2.2: The round constant c_r used in each round i

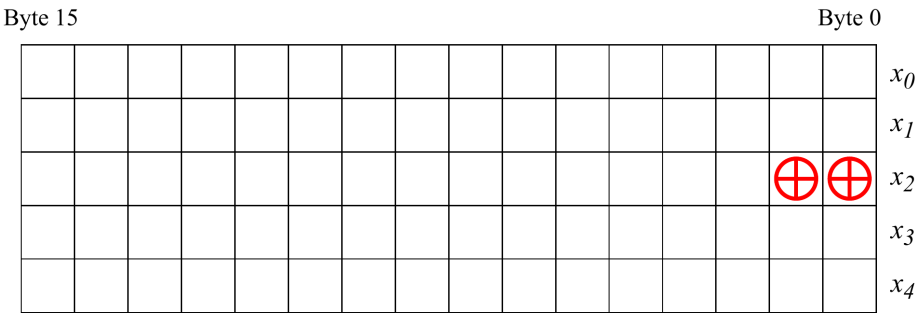


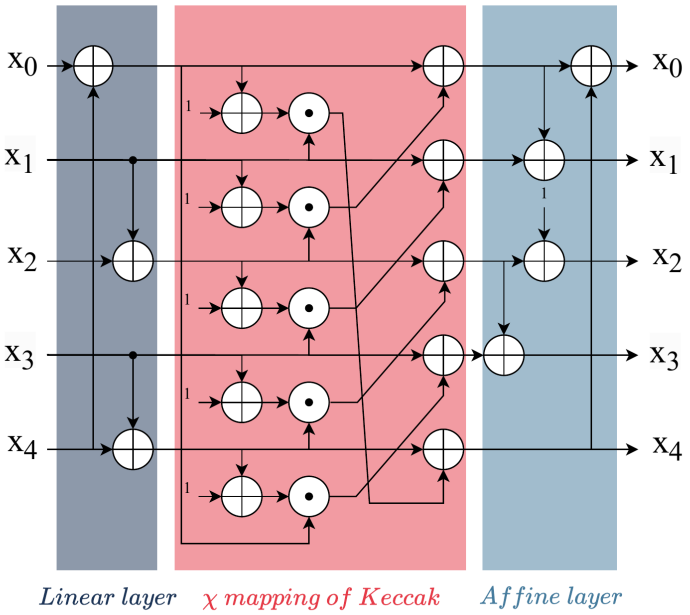
Figure 2.2: Round constant addition p_c

2. Substitution Layer p_s

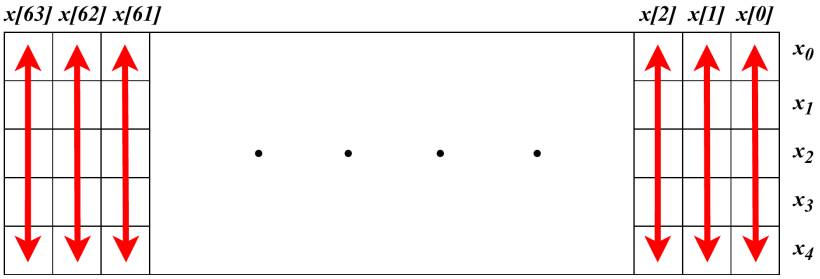
In the substitution layer, the state S is updated through 64 parallel applications of 5-bit S-box, denoted $S(x)$. The S-boxes used for ASCON are an affine transformation improves of the χ mapping of Keccak [7]. This transformation operates on each bit slice across the five 64-bit state words $x_0, x_1, x_2, x_3, x_4,$. Specifically, the S-box at position i takes as input the 5-bit vector $x_0[i], x_1[i], x_2[i], x_3[i], x_4[i]$.
The structure of the S-box is detailed in Table 2.3 and its effect is illustrated in Figure 2.3.

Table 2.3: ASCON's 5-bit S-box S as a lookup table.

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



(a) ASCON's 5-bit S-box $S(x)$



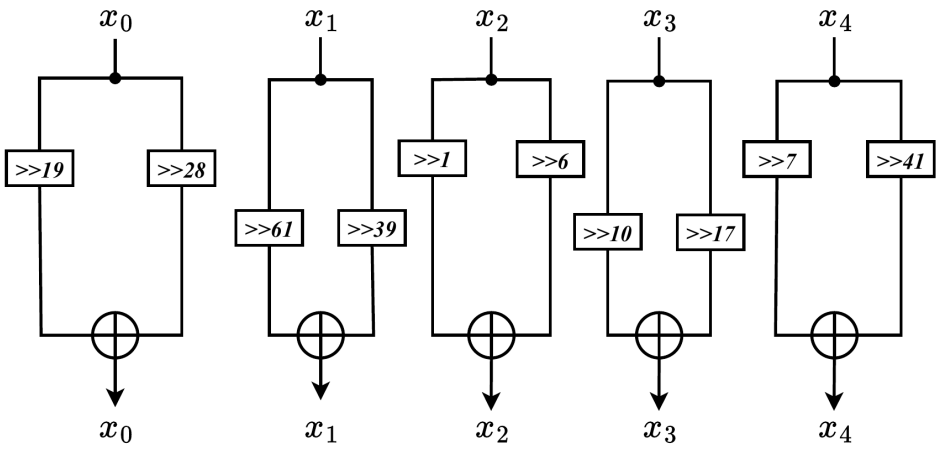
(b) Substitution layer p_s with 5-bit S-box $S(x)$

Figure 2.3: ASCON substitution layer: (a) S-box $S(x)$, (b) full layer with parallel instances

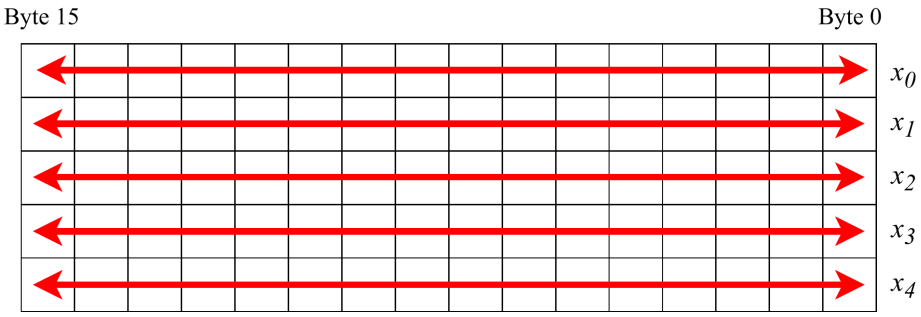
3. Linear Diffusion Layer

The linear diffusion layer p_L (Figure 2.4)provides diffusion within each register by performing two bit-wise rotations and XOR operations. The chosen rotation values are similar to the SHA-2 $\sum$ function [8], ensuring strong diffusion properties.

$$x_0 \leftarrow \sum\nolimits_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \tag{2.19}$$
$$x_1 \leftarrow \sum\nolimits_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \tag{2.20}$$
$$x_2 \leftarrow \sum\nolimits_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \tag{2.21}$$
$$x_3 \leftarrow \sum\nolimits_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \tag{2.22}$$
$$x_4 \leftarrow \sum\nolimits_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \tag{2.23}$$



(a) Linear diffusion layer



(b) Linear diffusion layer p_L application

Figure 2.4: ASCON linear diffusion layer and its application.

3 Side-Channel Attacks

Side-Channel Attacks (SCA) are a class of attacks that aim to extract sensitive information by exploiting physical or behavioral leakages unintentionally emitted by a device during the computation. Unlike theoretical attacks on cryptographic algorithms, which target mathematical weakness, SCA focus on real-world implementations, often bypassing the algorithm's theoretical security [9].
These attacks operate on the principle that every electronic device, while executing operations, produces observable signal such as power consumption, electromagnetic (EM) radiation, execution time variations, acoustic noise, or even physical movements. Since these signals can depend on the data being processed, including secret keys, attackers can analyze them to infer sensitive information and significantly reduce the complexity of breaking the system.
SCA are typically categorized along two axes (a more detailed classification is presented in [10], [11]):

- **Passive vs Active:** Passive attacks only observe the device, while active attacks deliberately interfere with it (e.g., clock/voltage glitching [12], EM injection [13], Laser/Optical [14], thermal manipulation [15]).
- **Invasive, Semi-invasive, Non-invasive:** This classification refers to the level of physical access. Today, the majority of attacks are non-invasive, requiring no hardware modifications.

As power analysis has matured, attacks have diversified into new side channels such as **electromagnetic (EM) emanations, cache access** patterns, and even **sensor readings** on mobile devices. Remote SCA techniques now leverage software only channels to compromise targets at scale—posing serious implications for mobile security, embedded systems, and cloud computing environments.
In the following chapters, we examine key power-based techniques (SPA, DPA, CPA), profiling attacks (TA), and evaluation methods (TVLA), outlining their principles, use cases, and effectiveness in real-world scenarios.

3.1 Power Measurement and Leakage Modeling

Power Measurement
Modern encryption algorithms rely on electronic devices that manipulate binary data (1s and 0s) through operations on transistor. Switching a bit from 0 to 1-or vice versa-requires applying or removing electrical current, which directly affects the

overall power consumption of the circuit. Since this consumption is correlated with the operations and data being processed, power measurements can leak information about the internal state of the system.
This forms the foundation of power **side-channel analysis**. Even subtle power fluctuations at level of individual transistor can correlate with cryptographic operations and thus reveal sensitive data, such as secret key.
To measure power , **Ohm's Law** is applied:

$$I = \frac{V}{R} \tag{3.1}$$

By placing a precise, fixed value resistor (known as *sense resistor*) in the power line of the cryptographic device, changes in current draw can be observed as voltage variations. An oscilloscope captures these fluctuactions, producing what is known as a **power trace** a time series of voltage readings during encryption or decryption.
The accuracy of this method improves with lower noise and closer proximity of the probe to the encryption device. Averaging multiple traces can further reduce random noise, thanks to the normal distribution properties of uncorrelated disturbances.

3.2 Classification of Attacks

Side-channel attacks can be classified along multiple dimensions. While physical access and attacker interaction are relevant factors (as discussed above in chapter 3), a fundamental distinction lies in whether the attacker has access to a profiling device (or " clone") during the attack setup.
We distinguish between **non-profiling attacks** and **profiling attacks**:

Classification	Description	Typical Techniques
Non-profiling	Attacker does not have access to a clone device for characterization. These attacks rely on statistical analysis of traces from the target itself.	Simple Power Analysis(SPA), Differential Power Analysis (DPA), Correlation Power Attacks (CPA)
Profiling	Attacker has access to a similar or identical device during a training phase, allowing them to build a leakage model used during the actual attack.	Template Attacks (TA), Deep Learning (DL)

Table 3.1: Classification of side-channel attacks based on attacker knowledge.

This distinction reflects the amount of knowledge the attacker has prior to the actual exploitation phase.

3.3 Non-Profiling Attacks

3.3.1 Simple Power Analysis (SPA)

SPA (Simple Power Analysis) is the most direct form of power side-channel attack. It involves the visual inspection of power consumption traces collected during cryptographic operations. These traces, often captured via oscilloscope, reveal how power usage changes over time and may expose sensitive details about the internal execution of an algorithm [9]. Each trace corresponds to a sequence of power samples recorded while the device performs an encryption or decryption. Even small variations in current (order of microamperes) can reflect meaningful differences in computation. By examining these variations, an attacker can infer executed instructions, branching behavior, or even recover parts of the key [16], [17].

Because SPA relies on direct observation, it is highly effective against poorly implemented algorithms that contain data-dependent branching or lookup tables. It is typically non-invasive and requires minimal equipment: a power probe, a sense resistor, and an oscilloscope. It may also overlap with **timing attacks**, since execution time often correlates with power usage. While SPA is limited by noise and requires clearly visible operations, it remains a potent threat, especially in unprotected or legacy systems.
There are scenarios where SPA directly exposes secret information. A classical example is code whose control flow depends on secret data, such as password checks.

Algorithm 1: Simple password check function pseudocode

```
1  function passwordCheck(userPassword):
2      secretPassword ← {1, 2, 3, 4};
3      errorLedOff();
4      if length (secretPassword) ≠ length(userPassword) then
5          errorLedOn();
6          return 0;
7      for position p of the password in range
         {0, . . . , length(secretPassword) − 1} do
8          if secretPassword[p] ≠ userPassword[p] then
9              errorLedOn();
10             return 0;
11     return 1;
```

As illustrated in Fig. 3.1, power traces collected from a password check show distinct patterns depending on how many digits of the input match the secret. With a fully correct password, we observe one pattern repetition per character. With a partially correct input, only the matching prefix is visible. When the length differs, no comparison is performed, and no pattern appears. Such differences leak valuable information to the attacker.

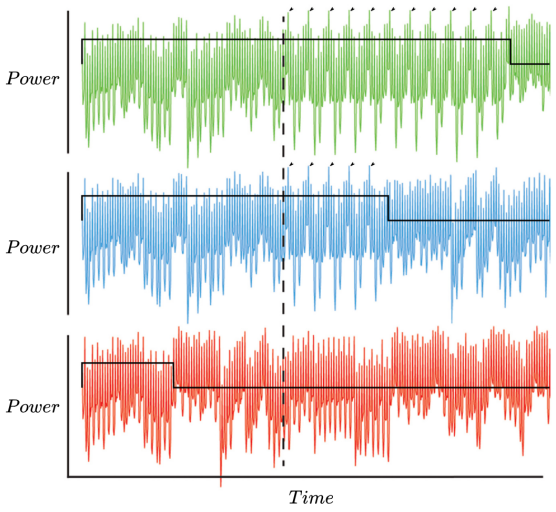


Figure 3.1: Example of power traces for correct, partially correct, and incorrect passwords.

3.3.2 Differential Power Analysis (DPA)

DPA, introduced by Kocher, Jaffe and Jun in 1999 [18], marked a turning point in side-channel research. While Simple Power Analysis (SPA), as seen, relies on visual inspection of individual power traces, DPA uses statistical methods to extract secret information across many traces, even when no obvious pattern are visible.
DPA automates the attack process by leveraging a **leakage model** to classify power traces and isolate differences linked to internal computations involving the secret key.
Attack Principle
The attacker defines a **selection function** [19], based on a key hypothesis, that predicts an intermediate value (typically a single bit or byte) of the cryptographic algorithm. This value is used to split the traces into two classes:

1. One class where the predicted value is 0.
2. One class where the predicted value is 1.

For each hypothesis:

- Traces are grouped according to the selection function.
- The **mean trace** of each group is computed.
- The **difference of means** is calculated.

If the key guess is correct, the statistical difference at specific time points will show a clear peak—revealing information about the secret key.
The **selection function** is typically constructed using common leakage models:

- **Hamming weight (HW):** assumes the power consuptions depends on the number of 1s in a data byte.
- **Hamming Distance (HD):** assumes leakage is related to number of bit flips between two successive values (e.g., register transitions).

These models allow the attacker to correlate hypotheses with measured leakage, even in the absence of full knowledge of internal device behaviour.

3.3.3 Correlation Power Analysis (CPA)

CPA, introduced by Brier, Clavier, and Olivier in 2004 [20], builds on the foundations of DPA and enhances it with more robust statistical method: correlation.
While DPA relies on comparing mean traces between groups, CPA analyzes the **correlation** between a **predicted leakage** (based on a key guess and leakage model) and the actual power traces, across time.
Attack Principle
CPA follows a model based approach using known inputs and power measurements. The attack works as follows:

1. **Known inputs** (e.g., plaintext) are collected during encryption.
2. For each **key guess:**
    - A hypothetical intermediate value is computed.
    - A **leakage model** is applied (e.g., Hamming Weight or Hamming Distance) to predict the power consuption.
3. The **Pearson correlation coefficient** is computed between the predicted leakage and the actual measured traces at each time point.
4. The key guess with highest correlation is considered correct.

3.4 Profiling Attacks

Unlike non-profiling attacks, which rely on leakage models (e.g., Hamming Weight, Hamming Distance) to partition traces into classes, Template Attacks directly exploit multivariate probability density functions to characterize the leakage distribution. In prattice, the attacker builds a "dictionary" of the expected side-channel emanations on a cloned device and then compares this reference with the measurements obtained from the target device [21].
Attack procedure. A Template Attack can be divided into four main phases:

1. **Profiling:** using a clone of the target device, the attacker collects tens of thousands of traces while varying both plaintexts and keys, in order to characterize the leakage associated with each possible subkey value.

2. **Template construction:** relevant *points of interest* (POI) are identified within the traces, and statistical parameters (typically mean vectors and covariance matrices) are computed for each subkey hypothesis. The outcome is a set of multivariate Gaussian distributions that model the device's behavior.

3. **Acquisition from the target:** a very limited number of traces are collected from the victim device.

4. **Matching and key estimation:** the victim traces are compared against the templates. For each subkey, the value that maximizes the likelihood under the statistical model is selected. This process is repeated until the entire secret key is recovered.

Key characteristics.

- Require only a very small number of traces from the victim device (sometimes even a single trace).

- The profiling phase is computationally expensive and data intensive, but needs to be performed only once.

- Achieve high attack success rates provided that the victim device closely resembles the clone used during profiling. Device-to-device variations may significantly degrade the effectiveness of the attack.

3.5 TVLA

Side-channel analysis, as seen, has traditionally focused on developing concrete attacks to recover secret information. In 2011, Goodwill et al. [22] proposed a complementary approach aimed not at breaking a device directly, but a detecting whether a cryptographic implementation exhibits exploitable leakages. This methology, known as **Test Vector Leakage Assessment (TVLA)**, has since become a de facto standard for evaluating the side-channel resistance of hardware and software implementations.

3.5.1 Theoretical Foundations

The main goal of TVLA is to determine whether countermeasures employed in a device under test (DUT) are effective in suppressing side-channel leakage. Unlike classical attacks, which target specific operations using a single selection function, TVLA applies a series of statistical tests to collected traces in order to reveal differences between carefully defined classes of inputs.

- **Non-specific leakage test:** compares traces obtained from the DUT while processing *fixed* versus *random* inputs. This test amplifies potential leakages in a generic way, even when specific attacks are not known in advance.

- **Specific leakage test:** traces are divided into classes according to known sensitive intermediates of the algorithm (e.g., S-box outputs, round states, or key related variables). The goal is to verify whether these internal computations correlate with observable side-channel information.

By distinguishing between these cases, TVLA provides both a general vulnerability assessment and a targeted check against specific leakage points.

3.5.2 Pratical Methodology

In practice, TVLA requires the collection of two datasets:

1. **Dataset 1:** generated by encrypting a large set of pseudorandom inputs with random Nonce (N_A) and with a fixed key.

2. **Dataset 2:** generated by repeatedly encrypting a fixed Nonce (N_B) under the same key.

These datasets are then combined and partitioned into two independent groups to mitigate false positives. Each group is processed using identical statistical procedures, and leakage is considered valid only if the same test point exceeds the threshold in both groups.
This methodology ensures that spurious anomalies are discarded and only consistent leakage points are reported as genuine vulnerabilities
The most widely adopted test in TVLA is the **Welch's t-test**, which evaluates the null hypothesis (H_0) that two sets of traces (e.g., Dataset 1 and Dataset 2) are drawn from the same distribution. A statistically significant difference between their means indicates the presence of leakage. Conventionally, a threshold of $|t| > 4.5$ is used to signal leakage with high confidence. The Welch's t-statistic for the first order case is computed as:

$$t = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}} \tag{3.2}$$

where μ_A and μ_B denote the sample means, σ_A and σ_B the sample standard deviations, and N_A and N_B the number of traces in groups A and B, respectively.
For higher-order designs, TVLA extends naturally by considering higher statistical moments.

4 Countermeasures: Masking

Masking is one of the most widely adopted countermeasure against Differential Power Analysis (DPA) and, more generally, side-channel attacks. It core idea is to decorrelate intermediate computations from the sensitive data being processed [23], [23]. This is achieved by splitting each sensitive variable into multiple *shares* such that observing fewer than all shares reveals no information about the original secret values. The recombination of the shares over a given field restores the original value.
Formally, a variable x is split into $s = d + 1$ shares to provide d-th order security:

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_s, \tag{4.1}$$

where $\oplus$ denotes the XOR operation over $\mathbb{F}_2$. Computations are then performed on each share independently, and the final result is obtained by recombining the partial outputs:

$$y = F(x) = F_1 \oplus F_2 \oplus \cdots \oplus F_s. \tag{4.2}$$

A fundamental security requirement is that every intermediate value must be statistically independent of all unshared sensitive variables. This notion is closely related to the d-probe model introduced by Ishai et al. [24], which defines an implementation as d-th order secure if an adversary probing up to d internal signals gains no advantage in recovering the secret. To build efficient hardware circuits under this model, different masked solutions were introduced.

4.1 Boolean Masking

In its simplest form, Boolean masking achieves first order security by splitting a sensitive variable into two shares. For instance, let $x = A_x \oplus B_x$ and $y = A_y \oplus B_y$ represent two masked inputs over $\mathbb{F}_{2^n}$. Their multiplication can be expressed as:

$$q = x \times y = (A_x + B_x)(A_y + B_2) = A_x A_y + A_x B_y + B_x A_y + B_x B_y. \tag{4.3}$$

Although each partial product is independent of the original inputs, their recombination may reintroduce dependencies. To mitigate this, a fresh random share Z_0 is typically added to one of the partial results to restore statistical independence, Fig. 4.1. However, this classical Boolean masking scheme is known to be flawed in practice, while it appears secure in the d-probe model, it is not resistant to *glitches* spurious transitions caused by differences in signal propagation times across wires or transistors. For example, if intermediate signals such as $A_x A_y$ and $A_x B_y$ arrive at

the recombination XOR gate before Z_0, the resulting value may momentarily depend on the secret input y. This timing dependent behavior can lead to first order leakage, undermining the theoretical guarantees of Boolean masking [25].
As a result, Boolean masking is generally considered inadequate as a standalone countermeasure in hardware implementations, motivating the development of more robust schemes such as Threshold Implementations (TI) and Domain-Oriented Masking (DOM), which explicitly address glitch resistance.

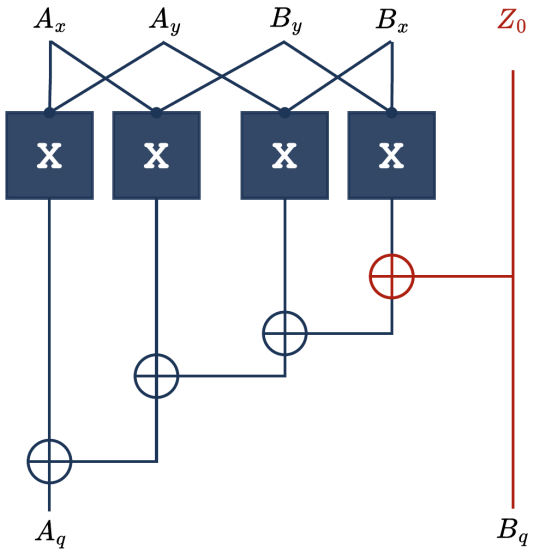


Figure 4.1: Classical masked GF (2^n) multiplier

4.2 Threshold Implementations (TI)

Threshold Implementation (TI), introduced by Nikova et al. [26], represent a masking scheme specifically designed to guarantee security even in the presence of glitches. The key idea of TI is decompose a function into *component function*, each satisfying three essential properties: **correctness**, **non-completeness**, and **uniformity**.

- **Correctness** requires that the sum of all component functions yields the same result as the original unshared function.
- **Non-completeness** ensures that no component function depends on all shares of any input variable. For first order implementations, each component must exclude at least one share of each input; more generally, for d-th order security, up to d component functions must be independent of at least one input share. This property prevents leakage caused by glitches.
- **Uniformity** demands that the output shares are uniformly distributed, regardless of the underlying unshared values. This property is often the most

difficult to achieve and typically requires either the use of additional shares, correction terms, or fresh randomness.

For linear functions over $\mathbb{F}_{2^n}$, first order security can often be obtained with only two shares. However, for non-linear functions, it is proved that at least three shares are necessary, even for first order security. In general, the number of shares required for TI follows the bound:

$$s_{\text{in}} \geq d \cdot t + 1, \tag{4.4}$$

where s_{in} is the number of input shares, d the protection order, and t the algebraic degree of the function. As an example, a first order secure TI of a GF(2) multiplier can be constructed with three shares per input and one fresh random share [27]. The outputs of the component functions are defined as:

$$F_A = B_x B_y + B_x C_y + C_x B_y + Z_0, \tag{4.5}$$
$$F_B = C_x C_y + A_x C_y + C_x A_y + A_x Z_0 + A_y Z_0, \tag{4.6}$$
$$F_C = A_x A_y + A_x B_y + B_x A_y + A_x Z_0 + A_y Z_0 + Z_0, \tag{4.7}$$

where Z_0 is a fresh random share used to ensure uniformity.

The first component function F_A is independent of all A shares, F_B is independent of all B shares, and F_C is independent of all C shares, thereby fulfilling the non-completeness property. In this way, TI achieves glitch resistance by construction. However, this security comes at a significant cost. While a classical (unmasked) GF(2) multiplier requires only four AND gates and four XOR gates, the TI variant in the above example consumes 13 AND gates, 12 XOR gates, and 3 registers. Consequently, TI offers strong first order security but incurs high hardware overhead, motivating the development of more efficient schemes such as Domain-Oriented Masking (**DOM**).

4.3 Domain-Oriented-Masking (DOM)

Domain-Oriented Masking (DOM), first introduced by Gross et al. [28], represents an efficient masking scheme tailored for hardware implementation. In contrast to Threshold Implementation (TI), which are designed at function level, DOM is based on the concept of *share domains*. Each share of a variable is associated with a distinct domain, and computations are organized such that shares from different domains are kept strictly separated. Using $d + 1$ shares per variable result in $d + 1$ domains and guarantees d-th order security in the probing model.

Moreover, DOM provides a *generic* masking paradigm with a configurable protection order, while TI constructions must be crafted for each target function (e.g., an S-box) and for each security order to satisfy correctness, non-completeness, and uniformity simultaneously; hence TI is not generic and typically requires re-derivation when the order changes.

The principle of DOM is straightforward: as long each intermediate signal is confined to a single domain, it remains independent of the unshared secret. For linear functions, this requirement is trivially satisfied since they can be evaluated independently on each share. The critical challenge arises with non-linear function, where domain

crossings are unavoidable, to prevent information leakage in these cases, DOM introduces two countermeasures:

- **resharing**, where cross domain terms are randomized using fresh shares Z.
- **register stage**, which block glitches from propagating across domain.

DOM-indep vs. DOM-dep. Two variants of DOM multipliers were proposed:

- **DOM-indep**, which assumes independently shared inputs and offers clear advantages in terms of randomness consumption and gate count.
- **DOM-dep**, which relaxes the independence requirement but incurs larger hardware overhead and was later shown to be insecure for masking orders of two or higher.

Since our work focuses on efficient and secure higher order designs, we only consider the DOM-indep construction.

4.3.1 First order DOM-indep multiplier.

A fist order secure DOM-indep GF(2^n) multiplier is illustrated in Fig. 4.2. The inputs x and y are split into two independent shares each: $x = A_x \oplus B_x$ and $y = A_y \oplus B_y$. The multiplier produces two output shares A_q and B_q, such that $q = x \cdot y = A_q \oplus B_q$.

The multiplier operates in three phases:

1. **Calculation:** Inner domain products (A_xA_y, B_xB_y) and cross domain products (A_xB_y, B_xA_y) are computed. Inner domain terms are trivially secure, while cross domain terms require protection.
2. **Resharing:** Each cross domain product is masked by adding a fresh random share Z_0, ensuring its statistical independence from the inputs. The resharing step concludes with a register stage to suppress glitches.
3. **Integration:** Reshared cross domain terms are reassigned to domains and combined with inner domain terms to produce the output shares.

Formally, the output shares of the first order DOM multiplier are:

$$A_q = A_xA_y \oplus (A_xB_y \oplus Z_0), \tag{4.8}$$
$$B_q = B_xB_y \oplus (B_xA_y \oplus Z_0). \tag{4.9}$$

Unlike classical Boolean masking, which is vulnerable to first order leakages due to glitches, the DOM-indep scheme ensures that any intermediate value depends on at most one share per variable. Thus, according to the d-probe model, an adversary must combine multiple probes across domains to extract sensitive information.

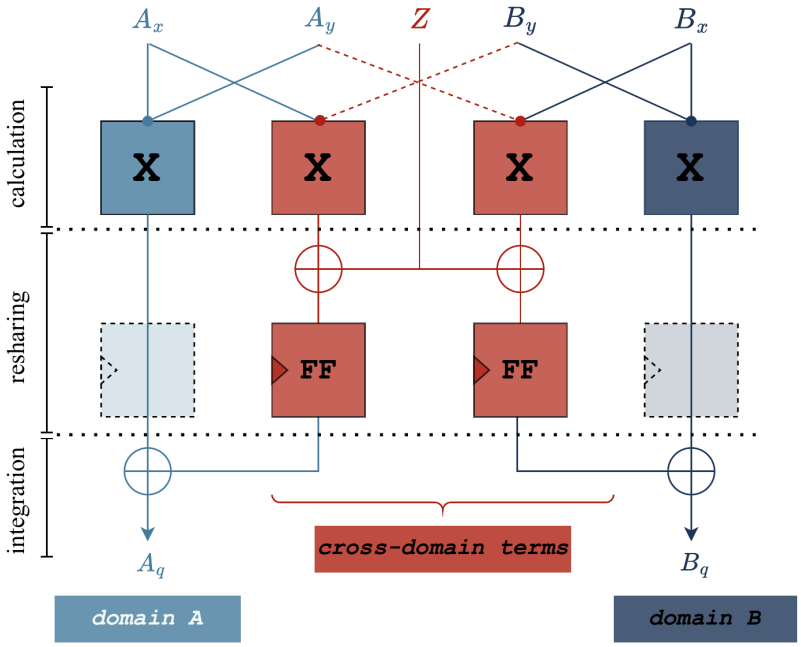


Figure 4.2: First-order DOM-indep GF (2^n) multiplier

4.3.2 High order DOM-indep multipliers.

The DOM-indep construction generalizes naturally to higher order by using $d+1$ shares per variable and distributing fresh random shares among the cross domain products. A d-th order multiplier requires exactly $d(d+1)/2$ fresh random bits and at least $d(d+1)$ registers to guarantee that all intermediate values remain independent in the probing model.
The generalized GF(2^n) multiplication can be expressed as:

$$q = x \cdot y = \left(\sum_{i=0}^{d} x_i\right) \cdot \left(\sum_{j=0}^{d} y_j\right) = \sum_{i=0}^{d}\sum_{j=0}^{d} t_{i,j}, \tag{4.10}$$

where $t_{i,j} = x_i y_j$ denotes the partial product between the i-th share of x and the j-th share of y. The diagonal terms $t_{i,i}$ correspond to inner domain products, while off-diagonal terms represent cross domain products, which are reshared using fresh random masks.

$$A_q = \mathbf{A_x A_y} \oplus (A_x B_y \oplus Z_0) \oplus (A_x C_y \oplus Z_1) \oplus (A_x D_y \oplus Z_3) \oplus (A_x E_y \oplus Z_6) \oplus \ldots \tag{4.11}$$
$$B_q = (B_x A_y \oplus Z_0) \oplus \mathbf{B_x B_y} \oplus (B_x C_y \oplus Z_2) \oplus (B_x D_y \oplus Z_4) \oplus (B_x E_y \oplus Z_7) \oplus \ldots \tag{4.12}$$
$$C_q = (C_x A_y \oplus Z_1) \oplus (C_x B_y \oplus Z_2) \oplus \mathbf{C_x C_y} \oplus (C_x D_y \oplus Z_5) \oplus (C_x E_y \oplus Z_8) \oplus \ldots \tag{4.13}$$
$$D_q = (D_x A_y \oplus Z_3) \oplus (D_x B_y \oplus Z_4) \oplus (D_x C_y \oplus Z_5) \oplus \mathbf{D_x D_y} \oplus (D_x E_y \oplus Z_9) \oplus \ldots \tag{4.14}$$
$$E_q = (E_x A_y \oplus Z_6) \oplus (E_x B_y \oplus Z_7) \oplus (E_x C_y \oplus Z_8) \oplus (E_x D_y \oplus Z_9) \oplus \mathbf{E_x E_y} \oplus \ldots \tag{4.15}$$

Each output share F_i can then be written in closed form as:

$$F_i = t_{i,i} + \sum_{j>i} \left(t_{i,j} \oplus Z_{\frac{j(j-1)}{2}+i}\right) + \sum_{j<i} \left(t_{i,j} \oplus Z_{\frac{i(i-1)}{2}+j}\right), \tag{4.16}$$

where Z_k denotes the fresh random shares assigned to cross domain products. Table 4.1 reports the hardware costs of a generic-order DOM multiplier. As can be seen, the overall hardware complexity *scales quadratically* with the protection order d.

Table 4.1: Summary of DOM-independent multiplier variants (single-bit GF(2) multiply).

	DOM-indep	**DOM-indep + resharing**
Related input sharing	no	yes
Register stages	1	2
Fresh random shares	$\frac{d(d+1)}{2}$	$\frac{d(d+1)}{2}$
GF multipliers	$(d+1)^2$	$(d+1)^2$
XORs	$d(d+1)$	$d(d+1)+(d+1)$
Registers	$d(d+1)$	$d(d+1)+(d+1)$

In Fig. 4.3 is shown an example of a second order DOM multiplier.

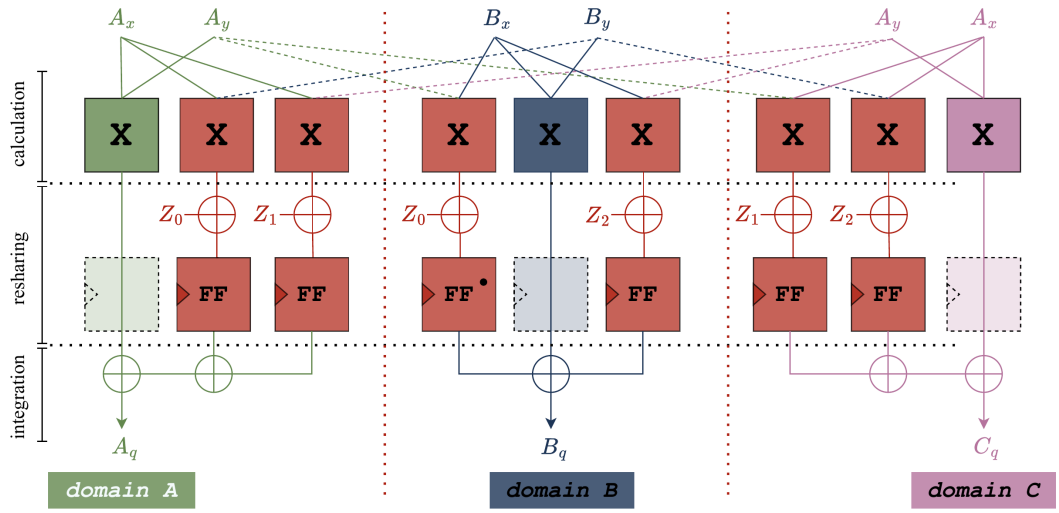


Figure 4.3: Second-order secure DOM-indep GF (2^n) multiplier

Security and cost. DOM-indep provides the same glitch resistance as Threshold Implementations but with significantly reduced hardware cost. While a TI multiplier requires 13 AND gates, 12 XOR gates, and 3 registers for first order security, the DOM-indep multiplier achieves the same level of protection with only 4 AND gates, 4 XOR gates, and 2 registers. The trade-off lies in the resharing phase, which requires additional randomness and registers that grow quadratically with the protection order. Nevertheless, DOM-indep currently represents one of the most efficient and scalable masking schemes for hardware cryptography.

4.3.3 Changing of the guards

Changing of the Guards (CotG) [29] is a technique originally introduced in the context of Threshold Implementations, with the goal of reducing or even eliminating the need for fresh randomness during computation. The method exploits the fact that shares of neighboring S-boxes are, by construction, uniformly distributed random variables. These shares can be *reused as additional randomness* to obtain a correct, uniform, and secure sharing of intermediate values.

In the context of Ascon, adopting CotG is particularly attractive because the randomness generation for re-sharing in DOM-AND gadgets typically requires dedicated hardware, contributing to area and energy overhead. By reusing neighboring shares as *Guards*, it is possible to minimize or eliminate this overhead, enabling lightweight masked implementations.

In this work we build on the result of [23], which explains how shares of neighboring S-boxes can be used as a source of randomness. Since not all bits are statistically independent from the bit currently being computed, the choice of the Guards must follow specific rules to guarantee probing security. For Ascon, this selection problem can be formalized in terms of *row* and *column offsets*, which identify the S-box position and bit used as Guard.

First-order (one Guard per DOM-AND). At **first order**, each DOM-AND requires a single *Guard*. To derive valid positions, we analyze the interaction and displacement introduced by the linear diffusion layer and the (low-latency) linear part of the S-box. For simplicity and without loss of generality of the method, we *fix the row offset to 0* and derive conditions on the *column offset* of the Guard. (Choosing a different row offset would change the admissible column offsets accordingly.)

Notation. Let R_i be the set containing the two rotation constants $\{r_{i0}, r_{i1}\}$ of row x_i (for $i \in \{0,1,2,3,4\}$). Let G be the set of admissible Guard column offsets.

Conditions to obtain Guard column offsets.

(C1) *Avoid using rotation constants directly as Guard offsets:*

$$\forall g \in G,\ \forall r_{ij} \in R_i : \quad g \neq r_{ij}.$$

(C2) *Avoid the pairwise* differences *of a row's rotation constants:*

$$\forall g \in G,\ \forall r_{ij} \in R_i : \quad g \neq (r_{i0} - r_{i1}) \bmod 64, \quad g \neq (r_{i1} - r_{i0}) \bmod 64.$$

(C3) *Avoid cross-row differences between x_0 and x_4 (due to the S-box linear layer added for low latency):*

$$\forall g \in G,\ \forall j \in \{0,1\} : \quad g \neq (r_{0j} - r_{4j}) \bmod 64, \quad g \neq (r_{4j} - r_{0j}) \bmod 64.$$

After applying **(C1)**–**(C3)** with row offset fixed to 0, we obtain 29 admissible column offsets. These are listed in Table 4.2. In practice, we further *avoid offsets that are not coprime with* 64 to prevent long-term correlations across multiple rounds (e.g.,

we discard {32}). In our implementation we pick the first coprime value, namely {11}, highlighted in the table.

Table 4.2: First-order: column offsets with row offset 0.

Column offsets for 1 guard (first order)									
{2}	{4}	{8}	{11}	{14}	{15}	{16}	{18}	{20}	{24}
{26}	{27}	{29}	{31}	{32}	{33}	{35}	{37}	{38}	{40}
{44}	{46}	{48}	{49}	{50}	{53}	{56}	{60}	{62}	

Second order (three Guards per DOM-AND). At **second order**, each DOM-AND requires *three* Guards, which significantly increases the complexity of the selection. Independence must be preserved both *among the Guards themselves* and *with respect to the masked variables*. As before, we keep the *row offset fixed to* 0 for the derivation; other row offsets lead to analogous (but different) admissible sets. Due to the number of constraints, we cast the problem as a constraint-satisfaction instance and solve it using a SAT solver (we used *Z3*), obtaining the valid triples reported in Table 4.3.

Notation. Let $G = \{g_1, g_2, g_3\}$ be the three Guard column offsets to be connected to a given S-box (position 0), and let $R_i = \{r_{i0}, r_{i1}\}$ be as above.

Generic constraints (row-offset independent).

- **(C1), (C2)** from the first-order case still apply to *each* $g \in G$.
- **(G1)** *No pair of Guards should combine (sum mod* 64*) to the target position (here,* 0*):*
$$\forall k \neq \ell \in \{1,2,3\}: \quad g_k + g_\ell \not\equiv 0 \pmod{64}.$$
- **(G2)** *Adding any rotation value must not map one Guard into another:*
$$\forall i,\ \forall j \in \{0,1\},\ \forall k \neq \ell: \quad g_k + r_{ij} \not\equiv g_\ell \pmod{64}.$$
- **(G3)** *Guard–rotation combinations must not collide across the two rotations of any row:*
$$\forall i,\ \forall k \neq \ell: \quad g_k + r_{i0} \not\equiv g_\ell + r_{i1} \pmod{64}.$$

Constraints specific to the chosen row offset (0)**.**

- **(C3)** from first order still applies (cross-row differences between x_0 and x_4).
- **(R1)** *Also avoid cross-row differences between* $\{x_1, x_2\}$ *and* $\{x_3, x_4\}$*:*
$$\forall j \in \{0,1\}: \quad g \neq (r_{1j} - r_{2j}) \bmod 64, \quad g \neq (r_{3j} - r_{4j}) \bmod 64.$$

The full (row-dependent) constraint set is more extensive; here we summarized the key ones used for encoding at row offset 0. Solving the constraints with a SAT solver yields the admissible triples in Table 4.3. As in the first-order case, additional heuristics (e.g., coprimality with 64) can be applied to reduce long-term correlations across rounds.

Table 4.3: Second-order: column-offset triples with row offset 0.

Column offsets for 3 guards (second order)				
{2,16,18}	{2,20,46}	{2,46,48}	{2,46,50}	{2,48,50}
{4,18,44}	{8,24,26}	{8,24,35}	{8,24,48}	{11,35,37}
{14,16,18}	{14,16,32}	{14,18,32}	{15,31,35}	{16,18,32}
{16,18,62}	{16,27,53}	{24,26,53}	{24,32,48}	{24,48,62}
{27,29,53}	{31,35,49}	{31,46,62}	{32,46,48}	{32,46,50}
{32,48,50}	{46,48,50}	{46,48,62}		

Shifted-domain trick (second order). In the second-order case we have three domains (A, B, C). Beyond choosing the column offsets, it is also beneficial to decide *how* to connect the domains to the random-bit locations of a DOM-AND gadget. We adopt the *shifted-domain trick*: connect Guard from domain A (G_1^A) to Z_0, domain B (G_2^B) to Z_2, and domain C (G_3^C) to Z_1 (notation as in Figure 1). This introduces symmetry, relaxes some constraints, and allows any permutation of a valid triple from Table 4.3 to be used in practice, thereby increasing the number of usable solutions.

Beyond second order. For higher orders, CotG has not been systematically extended: deriving complete, order-generic conditions that guarantee statistical independence for the bits reused as randomness becomes significantly more complex.

5 Hardware Implementation of Ascon Core

The designed presented in this work addresses the fundamental trade-off between security and efficiency in cryptographic hardware implementations. Fully masked design, while offering strong resistance against side-channel attacks (SCA), introduce significant area and latency overhead, which is particularly detrimental in resource constrained platforms. To overcome this limitation, our architecture exploits Ascon's mode level structure, selectively applying countermeasures only where strictly necessary.
In particular, we adopt a **selective masking strategy:**

- critical phases, such as **Inizialization** and **Finalization**, are executed with full masking protection;

- bulk operations, which do not require complete protection against differential power analysis, are processed without masking, thereby reducing latency and improving throughput.

This selective approach is enabled by a **modified masking gadget DOM-AND with dual functionality** (Fig. 5.3). During sensitive phases, the gadget acts as a side-channel countermeasure ensuring protection up to the desired security order. During bulk processing, the same hardware is reconfigured to support parallel execution of multiple data paths, effectively amortizing the masking overhead and achieving superior throughput-to-area ratios.

The main **design goals** of the proposed Ascon core can be summarized as follows:

1. **Side-Channel Resistance:** provide robust protection against first- and higher-order attacks through masking, applied only to security critical operations.

2. **Reconfigurability:** enable dynamic switching between masked and unmasked modes depending on the phase of computation

3. **Efficiency:** maximize throughput-to-area ratio by reusing hardware resources and instantiating only the minimum logic required for each security order.

4. **Scalability:** support configurable masking orders without without alterning the overall architecture.

5. **Reduced Randomness Requirements:** integrate the *Changing of the Guards* (CotG) technique to minimize or eliminate the consumption of the fresh randomness.

In summary, the proposed design leverages selective masking and reconfigurable hardware to achieve an optimal trade-off between security and efficiency. A high level block diagram of the architecture is presented in the next section, illustrating the structural modifications required to support dual-mode operation.

5.1 High Level Block Diagram

Building on the design objective introduced in the previous section, we now present the main characteristics of our generic architecture. The architecture is parameterizable with respect to the parallelism degree and can be therefore be instantiated for any value of parallelism. Certain configurations are of particular interest:

- **Parallelism = 1**, when minimizing area overhead is the primary goals;
- **Maximum parallelism** (see Table 5.1), when targeting the highest throughput during unmasked phases while still preserving a minimal area overhead.

These configuration offer the best trade-off in terms of **throughput-to-area ratio (Th/Area)**, which represents a key efficiency metric for our design.

Table 5.1: Maximum usable S-box parallelism for different masking degrees, enabling masking logic reuse.

Masking Degree d	**Max Parallel S-Boxes** (PAR_{MAX}) $\left\lceil \frac{64}{d+1} \right\rceil$
1	32
2	22
3	16
5	11

Note that PAR_{MAX} corresponds to the degree of parallelism that allows processing of the entire 64-bit state in a single cycle during the unmasked phases.

As previously noted, the proposed architecture guarantees **full masking** during both the initialization and finalization phases, while enabling higher throughput in the intermediate processing bulk data phase. The control of the Ascon core is handled by a synchronous **finite state machine (FSM)**, which will be described in detail in the next section (5.3).

The architecture is organized into the following main blocks:

1. **Input Network**
   This block handles the integration of the round constant and other operations

required during the algorithm, such as absorbing associated data (AD) or message (MSG) blocks, and key XORs. A schematic highlighting the responsibilities of the input network is provided in Section 5.2 in Fig. 5.4.

2. **Share Creation**
   The share creation unit generates the $(d+1)$ shares xoring the internal state with randomness provided by the RNG.

3. **S-Box Layer**
   We instantiate PAR S-Boxes in parallel. The internal DOM-AND primitive is redesigned to be both masked and capable of processing multiple bits in parallel Fig. 5.3. This dual functionality directly reflects the dual nature of the architecture.

4. **Register Layers**
   A set of $(d+1)$ registers, each of 320 bits, ensures correct separation across shares, as described in the masking Chapter 4. Additionally, several layers of flip-flops are required:

   - $5 \cdot (d+1)^2 \cdot PAR$ flip-flops within the S-Boxes, to avoid leakage caused by glitches (detailed in the DOM-AND Section 4.3);
   - $5 \cdot (d+1) \cdot PAR$ flip-flops immediately before each S-Box, to guarantee statistical independence of the S-Box inputs.

5. **Linear Diffusion Layer (LDL)**
   Applied after completing the non-linear layer, once the full 64-bit state column is available.

The modified Ascon architecture for a masking degree $d = 2$ is shown in Fig. 5.1.

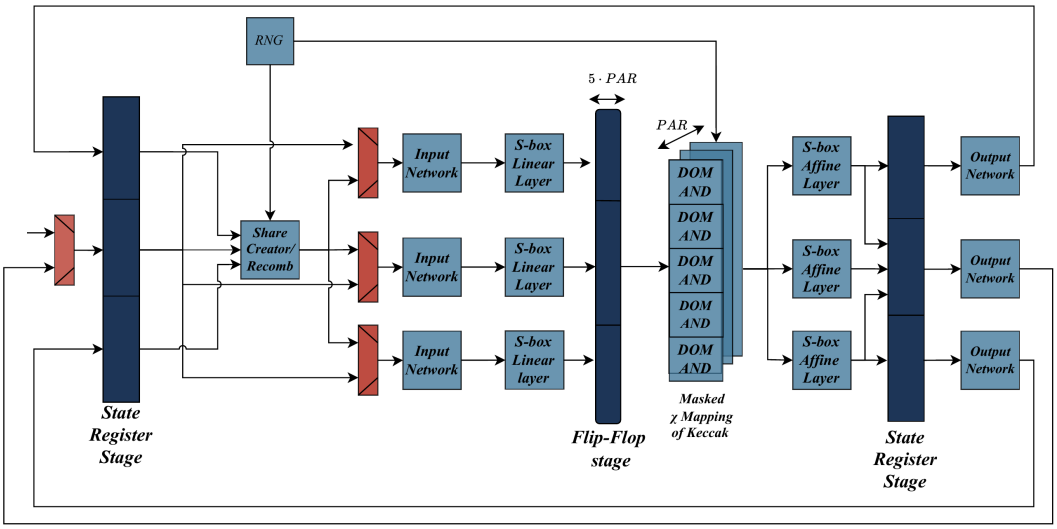


Figure 5.1: Modified Ascon architecture with 2^{nd}-order masking degree

Note: The two *State Register Stages* shown in the Fig. 5.1 actually represent the same set of registers. They have been duplicated in the illustration only for clarity, in order to present a cleaner and more readable structure.

The data processing sequence proceeds as follows:

Initialization: The IV, Key, and Nonce are first loaded. Depending on the RNG size, one or more cycles are required to generate the $(d+1)$ shares. Then the encryption starts with 12 initialization rounds (Table 2.1), processing PAR bits per cycle. The resulting share-separated state is stored in $(d+1)$ shift registers, each capable of shifting PAR bits per cycle (and, during unmasked phase , $PAR \cdot (d+1)$ bits). Special handling is required when 64 mod $PAR \neq 0$ or 64 mod $(PAR \cdot (d+1)) \neq 0$, in which case additional shift type are necessary. Before applying the Linear Diffusion Layer, the design waits until all 64 bits of a column have been processed by the Round Constant (RC) addition and S-Box stage. For this reason, dedicated counters are instantiated: one to track the current round and one to track the number of processed bits.

Associated Data (AD) and Message Processing: At the beginning of AD processing, before applying the XOR with the AD block and the round constant, the shares are recombined. This enables the hardware to process not only PAR bits, but up to $PAR \cdot (d+1)$ columns in parallel with the available resources. In this configuration, the inputs of the S-boxes are generalized: instead of receiving the $d+1$ shares of a single column, each S-box processes $d+1$ independent columns of the state. As a result, each cycle handles $PAR \cdot (d+1)$ columns, effectively increasing throughput. This requires a modification of the S-box logic, see Fig. 5.3, which must dynamically adapt to interpret its inputs either as shares (for initialization/finalization) or as independent columns (for AD/message processing) .

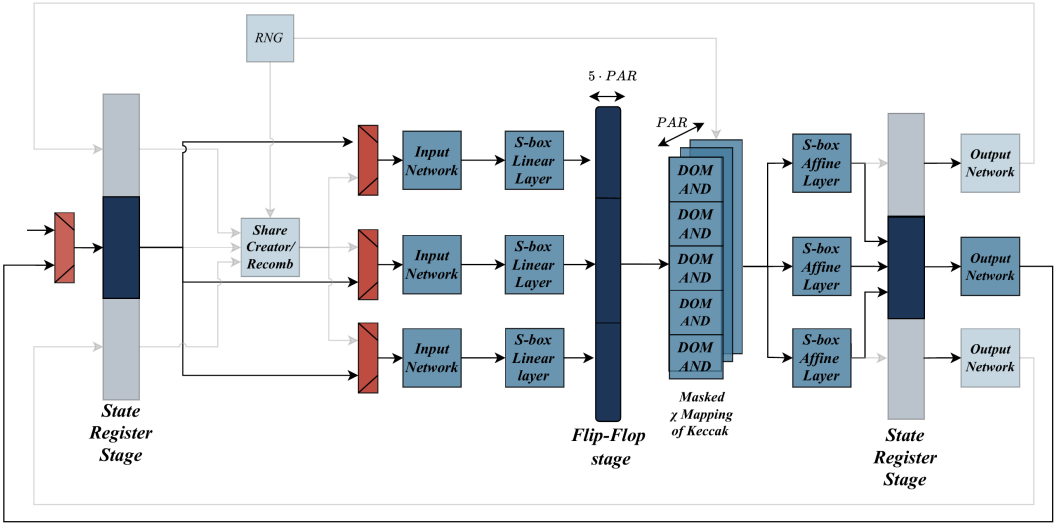


Figure 5.2: Ascon architecture at protection order $d = 2$. Only the blocks used during the unmasked (bulk) phase are highlighted.

Finalization: The finalization phase mirrors the initialization. The architecture reverts to masked mode, where the S-box layer operates on shares, and the state is processed in $(d+1)$ registers of width 320 bits each, ensuring proper domain separation.

This dual-mode operation requires the insertion of additional multiplexers to dynamically select whether the S-box operates on shares or columns, as well as $(d+1)$ parallel Round Constant Addition blocks (one per domain/column) to preserve masking consistency across all phases, which represent the principal overhead of our modified (highlighted in red in 5.1). A critical requirement during both share creation and DOM-AND computation is the use of fresh randomness at each round to ensure side-channel resistance. In our second-order masked implementation, we adopt the Changing of the Guards technique (see Section 4.3.3) to eliminate the need for fresh randomness in DOM-AND computations. However, fresh randomness remains mandatory during share creation, as the initial generation of masks cannot rely on *Guards*. Extending this technique to higher order masking remains challenging due to the rapid growth in *Guard* interactions with d.

5.2 Data Path Design

Representing the full Ascon data-path at the required level of detail would result in an excessively complex diagram. For this reason, in this section we focus on those components that have been significantly modified with respect to the original Ascon architecture, as highlighted in the high level block diagram.

5.2.1 Reconfigurable DOM-AND Gadget for Parallel Processing

The S-Box constitutes the most critical building block of the architecture, as it implements the non-linear operations and directly impacts both performance and side-channel resistance. In the original Ascon design, the DOM-AND gadget is instantiated to securely compute the masked AND between the $(d+1)$ shares of a single state bit. In our architecture, this block has been redesigned to support reconfiguration, thereby enabling parallel processing in message related phases where masking is not required.

The central idea is to **reuse the hardware resources** to perform multiple independent AND operations in parallel. Specifically, during the initialization and finalization phases, the DOM-AND operates in *masked mode*, processing the $(d+1)$ shares of a single variable according to the classical scheme. In contrast, during the message phases the gadget is *reconfigured* so that the $d+1$ columns of the state are directly mapped to the domains and treated as independent variables. This strategy enables efficient multi-column computation and significantly increases throughput.

For the first order case $(d=1)$, shown in (Fig. 5.3), the modified architecture allows two columns to be processed simultaneously, thus doubling the throughput compared

to the serialized baseline. More generally, throughput gains scale linearly with the masking degree d, since the reconfiguration enables up to $(d+1)$ columns to be processed in parallel without altering the fundamental structure of the gadget. The

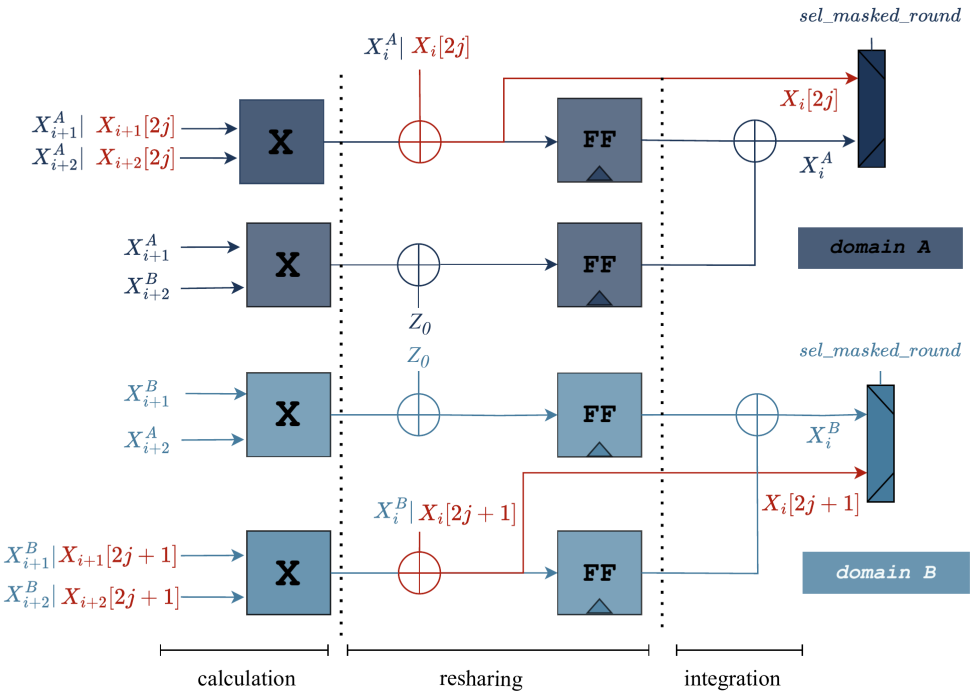


Figure 5.3: Reconfigurable DOM-AND gadget

main modifications with respect to the classical DOM-AND design (see Fig. 4.2) are highlighted in red in Fig. 5.3. The most significant change concerns the input mapping during unmasked operation: instead of receiving the $(d+1)$ shares of a single variable (A, B), the inputs to the S-Box are connected to different state columns (e.g., $2j$ and $2j+1$). This functionality is enabled by the additional multiplexers instantiated in the data block (see Fig. 5.1).
Another modification is the relocation of the affine layer XOR. In the original scheme, the affine XOR is applied sequentially after the randomness XOR performed during the resharing phase. In our implementation, these two operations are executed in parallel, which effectively saves one XOR stage in the processing pipeline [23]. Finally, the non-masked outputs are produced in $(d+1)$ parallel streams.
In principle, it would be possible to further exploit the remaining DOM-AND instances to process up to $(d+1)^2$ columns simultaneously. However, such an approach would significantly increase control complexity and routing overhead. Instead, the adopted design strikes a balance, achieving meaningful throughput improvements at minimal additional cost. This reconfigurable DOM-AND gadget therefore ensures higher order security during masked operation while delivering substantial throughput gains in unmasked phases, making it a practical and efficient solution for protected cryptographic implementations.

5.2.2 Input Network

The input network does not undergo any modification compared to the standard case. In the masked configuration, the input networks corresponding to shares $1 \dots d$ are not used, and the multiplexers always select the path where the output is equal to the input. This behavior is due to the fact that the sharing operations are applied only to Share A; otherwise, if the number of shares were even, the XOR would cancel out. In contrast, during unmasked operation, each input network correctly XORs the corresponding bits as required by the algorithm.
Fig. 5.4 shows the input network for the PAR_{MAX} case. The number of input networks are always $(d+1)$, but this configuration is presented because it allows the use of the index j—representing the share number—without requiring additional offsets. For example, in the case $d = 2$, the first input network XORs bits $0 : 21$, the second one XORs bits $22 : 43$, and the last one XORs bits $44 : 63$. In the general case, however, it is also necessary to take into account the number of processed bits. This requires the introduction of an offset expressed as $(j \cdot PAR + i \cdot PAR)+ : PAR$, where j is the share index and i is the bit counter, i.e., the number of executed shift operations. Note the key is required to perform the XOR operation before entering the finalization round Fig. 2.2.

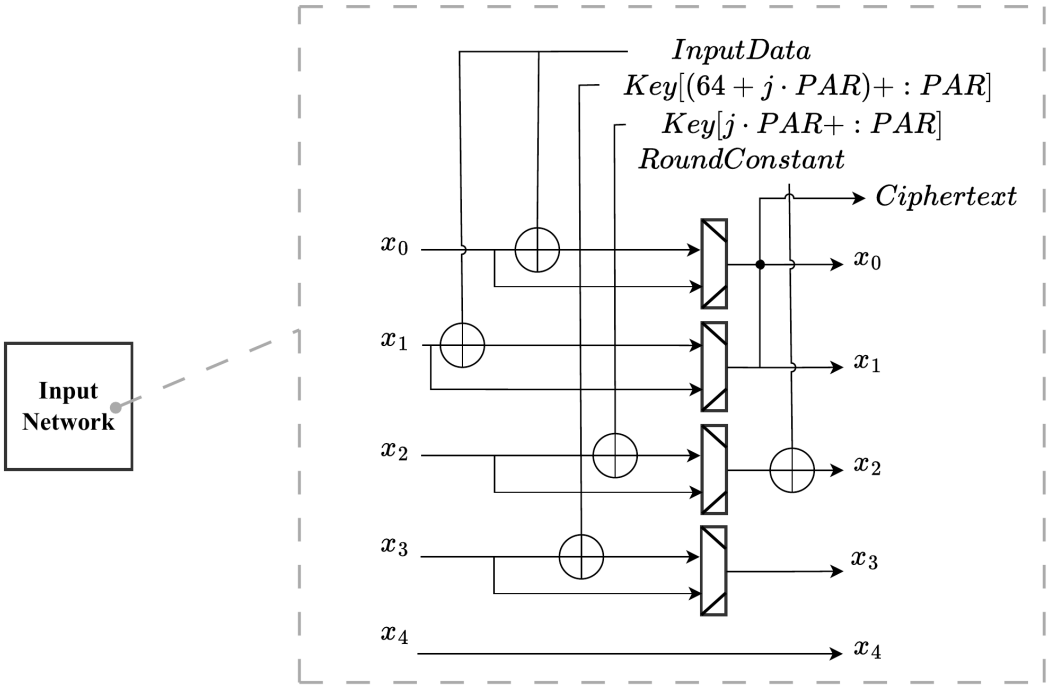


Figure 5.4: Input Network at PAR_{MAX}

5.2.3 Output Network

The output network follows the same organization as in the unmasked case. In the masked configuration, each of the $(d+1)$ output paths contains the *linear diffusion layer*, which is always applied to the state words. In addition, the output network

of *Share A* includes a dedicated XOR network that enables the key and domain additions required by the algorithm: (i) the XOR of x_3 and x_4 with the key at the end of the initialization phase, and (ii) the XOR of x_4 with $0^{63} \| 1$ at the end of the AAD processing phase. In the masked mode only Share A performs these external additions, while the remaining d output paths propagate the corresponding state words, preserving correctness and the masking invariants.
Fig. 5.5 shows the output network for the Share A; as for the linear diffusion layer, it operates on the full 320-bit state $(x_0, \ldots, x_4)$.

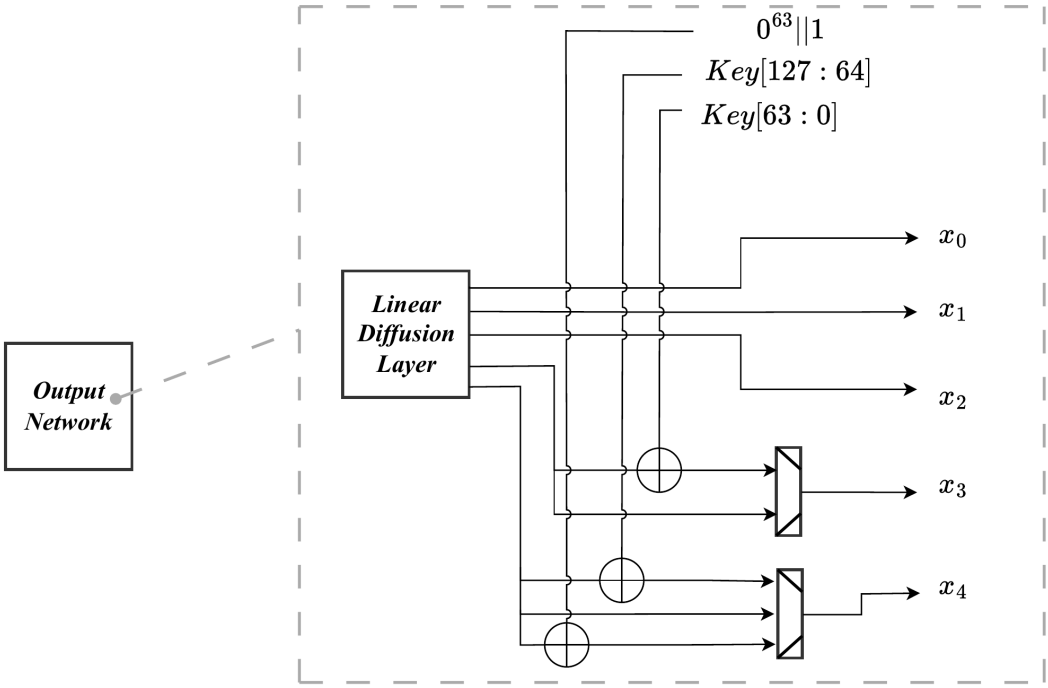


Figure 5.5: Output Network (Share A)

5.2.4 State Register

Figure 5.6 shows one of the status registers of size $d + 1$. First, we can observe the inputs: on the 320-bit interface the *State_in* is received, which represents the initialization state composed of *IV, KEY, NONCE*. In addition, the output of the output network *LDL_out* is provided, together with the input data (MSG, AAD), which are absorbed into the state through an `xor` operation (*Absorb_data*). Furthermore, inputs of PAR bits are foreseen in the masked execution, or $PAR \cdot (d+1)$ bits in the unmasked execution.
As for the outputs, the complete 320-bit state is always available, while a dedicated $(d+1) \cdot PAR$-bit output provides a reduced portion of data when needed. The selection of the correct data path is handled by the `sel_masked_round` signal, which allows the proper number of processed bits to be extracted.

At the top of the figure, the internal structure of the register is depicted. The

state is divided into five 64-bit words $(x_0, x_1, x_2, x_3, x_4)$, on which shifts may be applied. In particular, the possible shifts are:

- PAR bits,
- $PAR \cdot (d+1)$ bits,

In some cases, the following may also be required:

- 64 mod PAR,
- 64 mod $(PAR \cdot (d+1))$

It should be emphasized that the actual value of the applied shift depends on the control signals `shift_type` and `last_cycle`, and a shift is performed only if `shift_en = 1`.
A specific example is illustrated in the figure, while a particular case arises for $PAR = 3$ and $d = 2$: in this scenario, the required shifts are 3, 6, 1, and 4. Finally, it is important to highlight that all state words are always shifted by the same amount: although all possible shifts are shown in the figure for completeness, different words are never shifted by different amounts.

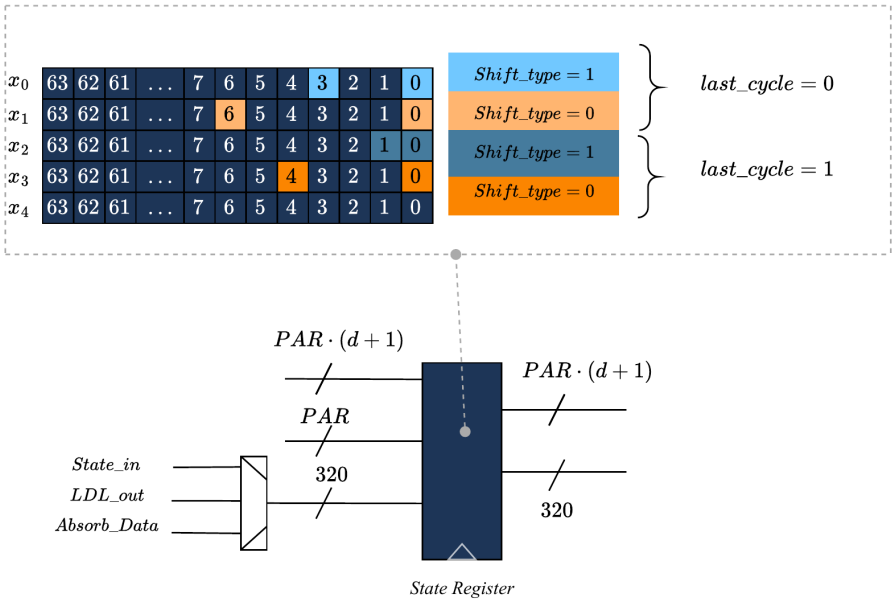


Figure 5.6: Status Register

5.3 Interface & Finite State Machine

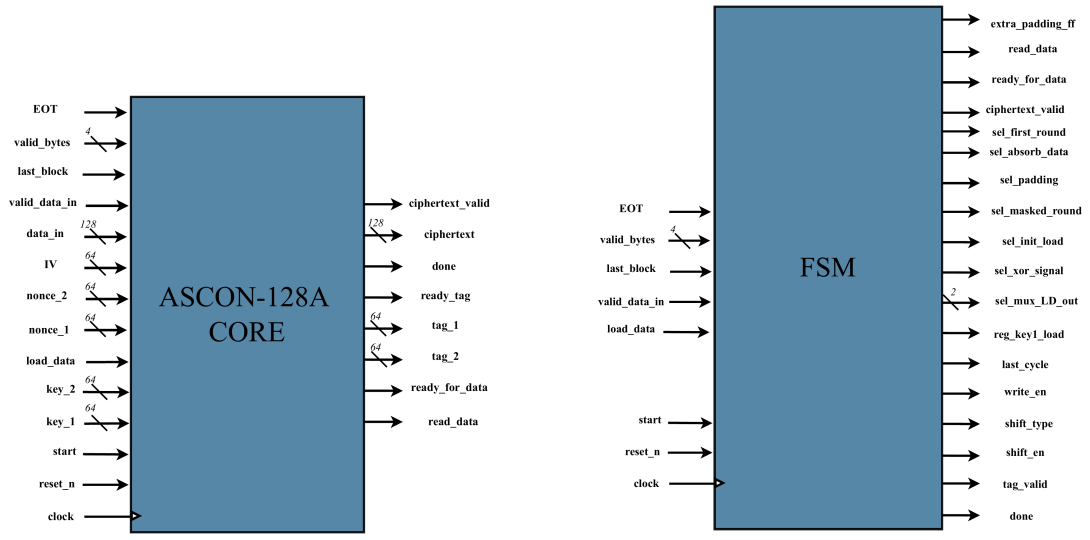
5.3.1 Core Interface

The top-level interface of the Ascon core is depicted in Fig. 5.7a.
Inputs

The data-path inputs comprise the secret key (*key_1, key_2*), the public nonce (*nonce_1, nonce_2*), and the initialization vector (*IV*), each provided as a 64-bit word. Message data are supplied through the *data_in* port (128 bits), together with the *valid_data_in* signal, which indicates the availability of either associated data (AAD) or plaintext/ciphertext blocks. The *valid_bytes* signal specifies the number of valid bytes in the final word, while the *last_block* signal identifies whether the current AAD block is the last one, after which the core transitions to plaintext/ciphertext processing. The EOT (End of Transmission) signal marks the termination of the input stream. The control interface includes the global clock (*clk*), asynchronous reset (*reset_n*), and the *start* signal, which initiates the initialization phase. A dedicated *load_data* signal is also provided to preload key and nonce values prior to starting the permutation.

Outputs

The cryptographic outputs include the ciphertext stream (*ciphertext*), accompanied by its validity flag (*ciphertext_valid*), as well as the authentication tag, split across *tag_1* and *tag_2*, with the corresponding *ready_tag* signal. The *done* signal asserts at the end of the encryption or decryption process.
To facilitate streaming integration, the core further provides handshake signals. The *ready_for_data* signal asserts when the core is ready to accept new input, while the *read_data* signal indicates that the current input block has been successfully consumed.



(a) Ascon Core Interface (b) FSM Interface

Figure 5.7: Interfaces of the Ascon core and its FSM.

5.3.2 Finite State Machine (FSM)

The control of the Ascon core is managed by a **synchronous finite state machine (FSM)** driven by the global clock (*clk*) and asynchronous reset (*reset_n*). The

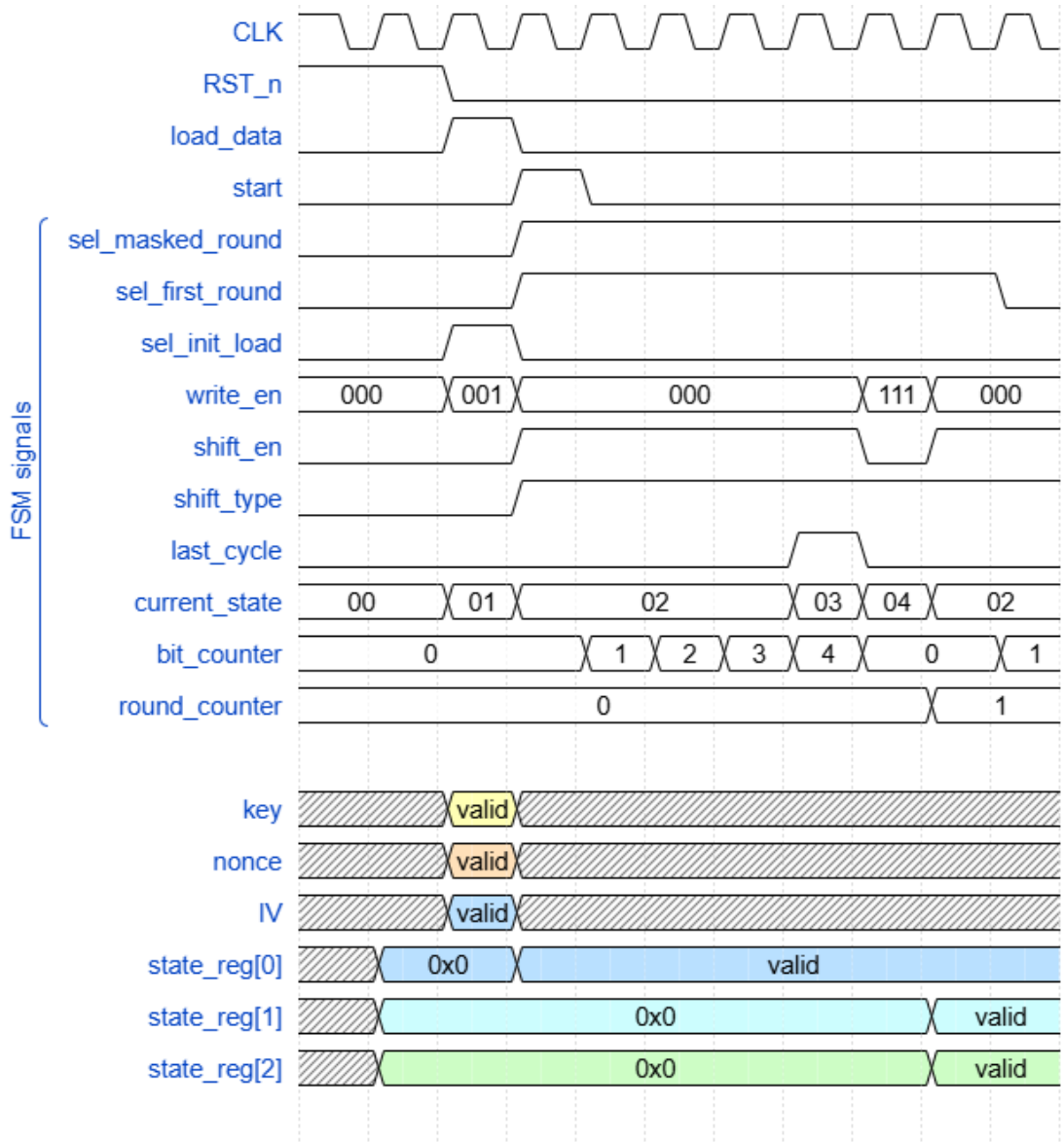


Figure 5.8: Timing Diagram Initialization phase

Note: the corresponding FSM state encoding is reported in Appendix C.

FSM orchestrates the datapath through all phases of the algorithm: initialization, absorption of associated data (AAD) and message blocks, finalization, and tag generation.
The FSM exposes a set of **control**, **status**, and **handshake signals** that connect it both to the datapath and to the external microcontroller (MCU) interface.
The FSM interface is shown in Fig. 5.7b.

Input signals

The FSM receives the following inputs:

- *clk, reset_n* – global synchronous clock and asynchronous reset.
- *load_data* – indicates that the secret key, nonce and IV are avaible at the input of the core.
- *start* – command from the MCU that triggers the transition from *LOAD DATA* phase to *INITIALIZATION SHIFT*.
- *valid_data_in* – asserts when a block of AAD or message/ciphertext is available at the input.
- *last_block* – signals that the current AAD block is the final one, after which the FSM can proceed with *message processing*. In many implementations, an alternative approach is adopted where a *message_type* signal explicitly distinguishes between AAD and message blocks, while *EOT* is consistently used to indicate the end of the transmission. The functionality is conceptually equivalent, with only the naming convention of the signals differing.
- *valid_bytes[3:0]* – specifies the number of valid bytes in the last input word, enabling padding management.
- *EOT* (End Of Transmission) – marks the end of the input stream and triggers the finalization phase.

Output and status signals

The FSM produces the following status outputs:

- *done* – asserted when the encryption or decryption process is fully completed.
- *tag_valid* – indicates that the authentication tag is available.
- *ciphertext_valid* – signals that ciphertext/plaintext output data are valid.
- *ready_for_data* – asserts when the FSM is ready to accept new input blocks.
- *read_data* – acknowledgement that the current input block has been consumed.
- *extra_padding_ff* – internal flag used to handle the case where an additional padding block must be inserted, i.e., when the end of transmission of either AAD or message data coincides exactly with a full 128-bit block.

Control signals toward the datapath

To drive the datapath, the FSM generates the following control signals:

- *shift_en, shift_type, last_cycle* – control the internal shift network. Specifically, *shift_en* enables shifting, *shift_type* selects whether to shift PAR bits (masked mode) or $PAR \times (d+1)$ bits (unmasked mode), and *last_cycle* identifies the final shift in a round, as already explained in part **Initialization** of Section 5.1, see Fig. 5.6.
- *write_en* – enables parallel load of the internal state register.
- *reg_key1_load, reg_key2_load* – control loading of the key registers during initialization.
- *sel_init_load* – selects loading of initialization data (IV, nonce, keys).
- *sel_masked_round* – selects between masked rounds (12, indices 0–11) and unmasked rounds (reduced, indices 4–11), used also to drive the multiplexer to select the S-boxes inputs and outputs.
- *sel_mux_LD_out* - selects between the raw output of the linear diffusion (LD) stage and the post-XOR path; see Fig. 5.5. When deasserted (0), both x_3 and x_4 are taken directly from the LD stage. When asserted (1), x_4 is taken from the XOR path, while x_3 is taken from the XOR path only when *sel_xor_signal* requests a key addition (init); otherwise x_3 still bypasses the XOR and uses the LD output.
- *sel_padding* - enables the padding logic when the last block is incomplete, or when an extra padding is required because the last block of the transmission is 128 bits.
- *sel_xor_signal* - selects which XOR is applied to x_4: either $x_4 \oplus K$ (key addition at the end of initialization/finalization) or $x_4 \oplus (0^{63} \| 1)$ (domain constant at the end of AAD); see Fig. 5.5.
- *sel_absorb_data* – enables absorption of AAD or message data into the state.
- *sel_first_round* enables the generation of the shares during the first round of the *INITIALIZATION* and *FINALIZATION* phases, and it controls the recombination process during the first round of the *PROCESS_AAD* phase.

MCU integration

The FSM is designed to be seamlessly integrated as a **memory mapped accelerator** within a microcontroller environment. Once started, the FSM autonomously drives the internal datapath through all phases of the Ascon algorithm, from initialization to finalization. Throughout execution, the MCU interacts only through simple handshake and status signals: *ready_for_data* and *read_data* regulate the input stream, while *ciphertext_valid*, *tag_valid*, and *done* indicate the availability of output

data and the end of processing. When the FSM reaches the *DONE* state, de-asserting *start* returns the machine to *IDLE*, ready for a new encryption or decryption session. This design minimizes software overhead, as the MCU is only responsible for providing input blocks and reading back the ciphertext and authentication tag.

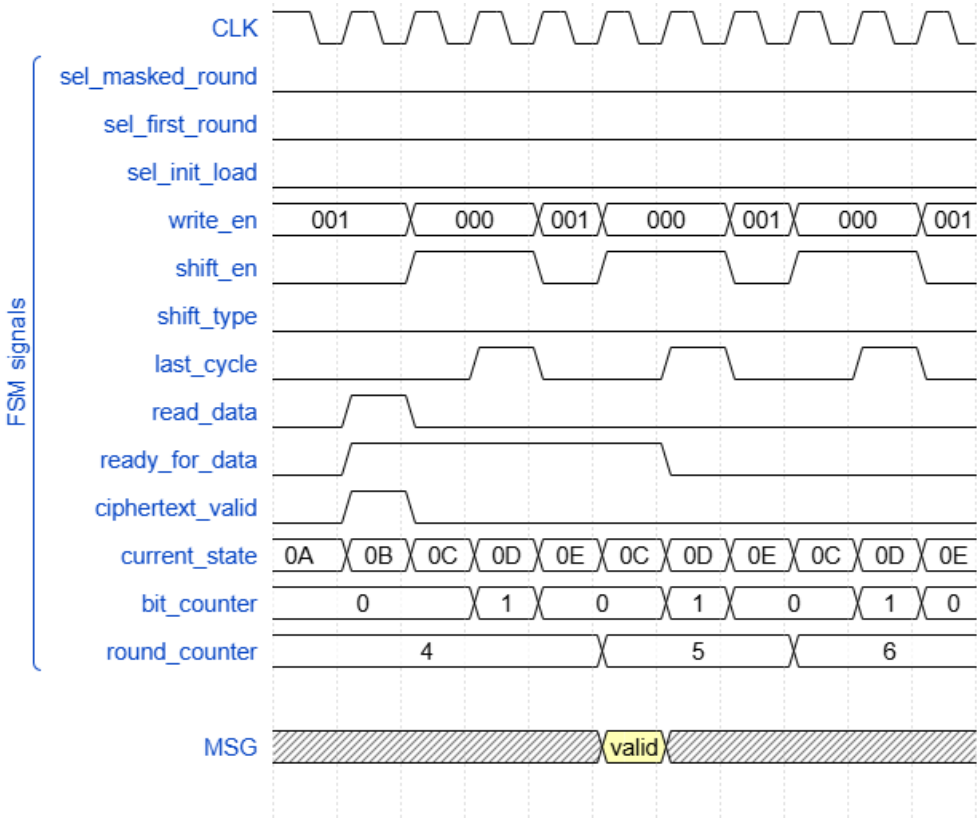


Figure 5.9: Timing Diagram Absorb MSG

Note: the corresponding FSM state encoding is reported in Appendix C.

FSM state transition diagram The overall behavior of the FSM is summarized in the **state transition diagram** shown in Fig.5.10. Each rectangle corresponds to one of the macro-states described above (initialization, absorption, message processing, finalization), while the edges represent the logical conditions and control signals that trigger the transitions.

For graphical compactness, the evolution of the FSM has been represented through macro-states:

- **IDLE**,
- **LOAD_DATA**,
- **INITIALIZATION**,
- **PROCESS_AAD**,

- **PROCESS_MSG**,
- **FINALIZATION**,
- **DONE**

However, the main phases INITIALIZATION, PROCESS_AAD, PROCESS_MSG, FINALIZATION are themselves subdivided into multiple sub-stages. In Fig. 5.10, the sub-states of the initialization phase are explicitly shown, which are analogous to those of the finalization phase: INIT(FIN)_ROUND_SHIFT, INIT(FIN)_ROUND_SHIFT_LAST, INIT(FIN)_DIFFUSE, INIT(FIN)_DIFFUSE_LAST. A similar structure applies to the process of AAD and MSG, which differ exclusively in the exit condition from their respective states (see Fig. 5.11, where the exit condition for the PROCESS_MSG replaces the highlighted condition in PROCESS_AAD, marked with an asterisk). Analyzing the state evolution in detail, in the IDLE state no

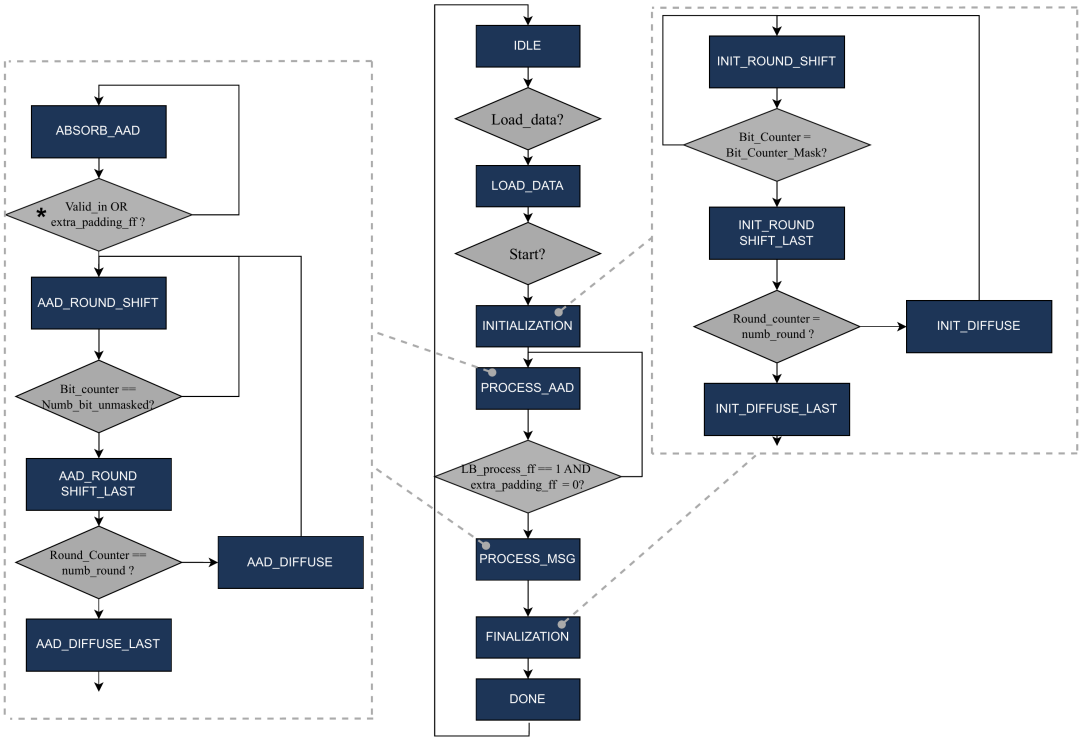


Figure 5.10: FSM state transition

operation is performed; registers are reset and the system waits for the *load_data* signal. Once asserted, the FSM transitions to the LOAD_DATA state, where the input data (IV,KEY, NONCE) are loaded. After this, the system waits for the *start* signal, which initiates the INITIALIZATION macro-states.

The INITIALIZATION phase consists of permutation A and is internally divided into the sub-states INIT_ROUND_SHIFT, INIT_ROUND_SHIFT_LAST, INIT_DIFFUSE, and INIT_DIFFUSE_LAST. Since the architecture is fully

parametric, a *bit_counter* keeps track of how many groups of PAR bits have been processed.
Once the last group of bit is reached, the FSM transitions to INIT_ROUND_SHIFT_LAST. This distinction is required because in the last cycle the shift may not be equal to PAR if 64 mod $PAR \neq 0$. Afterward, the system proceeds to INIT_DIFFUSE (if the round counter is less than 11, i.e., we are not yet at the 12th round) or to INIT_DIFFUSE_LAST otherwise. Both INIT_DIFFUSE and INIT_DIFFUSE_LAST apply the linear diffusion layer and subsequently activate the write enable signal of the $(d+1)$ registers to store the updated shares. The only difference is that, in INIT_DIFFUSE_LAST, the *rst_cnt_4* signal is also activated. This resets the round counter to 4 instead of 0, as the absorption phases of AAD and MSG must consist of 8 rounds. To compute the correct round constant (RC), the round counter must therefore start from 4.
The main differences with respect to the FINALIZATION phase are that neither the *rst_cnt_4* nor the *sel_mux_ld_out* signals are asserted, indeed during FINALIZATION, the *rst_cnt_4* signal is not required, while the *sel_mux_ld_out* signal remains inactive as well; the only control signal that is specifically asserted in this phase is *tag_valid*. This behavior is also highlighted in Fig. 5.12, where signals exclusively related to INITIALIZATION are shown in blue and marked with (I), whereas those belonging only to FINALIZATION are depicted in red and marked with (F).

After the state INIT_DIFFUSE_LAST, the FSM enters the PROCESS_AAD phase, starting from the ABSORB_AAD state. In this state, the signal *write_en[0]* is asserted, enabling only state register 0. This is because the absorption of AAD is performed in an unmasked fashion, as explained earlier in this Chapter 5, and thus the additional d registers dedicated to the shares are not used. At this stage, the control signal *sel_first_round* is also relevant, as it determines whether the shares need to be recombined immediately after the INITIALIZATION phase or not. Subsequently, the signal *sel_absorb_data* is asserted, which selects the correct data for loading into the state register, while the padding logic is enabled in case the input *valid_bytes* is less than 16. The system remains in this state until either a valid input block is received or the *extra_padding_ff* flag is asserted. The latter indicates that the last processed AAD block is the last one (so *last_block* was equal to 1) and consisted of 16 valid bytes, thus requiring the addition of an extra block of the form $1||0^*$ to be XORed with the state.
It is worth noting that the padding is handled automatically: if the input *valid_bytes* is smaller than 16, the input block is padded as $1||0^{128-b-1}$, where b is the number of valid input bits.
After this, the FSM transitions to the AAD_ROUND_SHIFT state. The behavior of this state is analogous to the SHIFT phase in INITIALIZATION, except that the signal *shift_type* is not asserted, since the shift always processes $PAR \cdot (d+1)$ bits per clock cycle. A bit counter tracks the number of processed bits, and once it reaches $num\_bit\_unmasked = \lceil 64/((d+1) \cdot PAR) \rceil$, the FSM transitions to AAD_ROUND_SHIFT_LAST. In this state, the signal *last_cycle* is asserted to handle the final shift of 64 mod $(PAR \cdot (d+1))$ bits, which may differ from $PAR \cdot (d+1)$.

The next step is the diffusion phase: the FSM transitions either to AAD_DIFFUSE (if the round counter < 11) or to AAD_DIFFUSE_LAST (if the round counter $= 11$). The main difference with respect to INITIALIZATION is that only *write_en[0]* is asserted, rather than all $(d + 1)$ registers, since the computation is performed without shares. Moreover, in the state AAD_DIFFUSE_LAST, the signal *sel_mux_ld_out_x4* is asserted if the last AAD block is being processed and no extra padding is required (*extra_padding_ff = 0*). This enables the state register to load the value XORed with $0^{319}||1$. In this case, the signal *sel_xor_signal* is not asserted, meaning that the XOR path selects $0^{127}||1$ applied to x_3 and x_4 instead of the alternative $Key[127:0]$, see Fig. 5.5.
After completing PROCESS_AAD, the FSM proceeds to the PROCESS_MSG phase. The state transitions are very similar to those of PROCESS_AAD, with only minor differences in the output signals. Specifically, in the ABSORB_MSG state, the signal *ciphertext_valid* is asserted (unless *extra_padding_ff = 1*), as highlighted in figure 5.12. In the MSG_DIFFUSE_LAST state, the signals *sel_mux_ld_out_x4* and *rst_cnt_4* are not asserted, since the FSM must proceed with all 12 rounds of the finalization phase.
The main structural difference is that the last state of the message processing phase is not MSG_DIFFUSE_LAST, but rather ABSORB_MSG, as highlighted in Fig. 5.11. Specifically, the FSM transitions to FINALIZATION either when a valid input block is received without requiring extra padding (*valid_bytes* < 16), or when the *extra_padding_ff* flag is asserted.

Finally, the system enters the FINALIZATION phase, which has already been described in detail in relation to the INITIALIZATION phase.

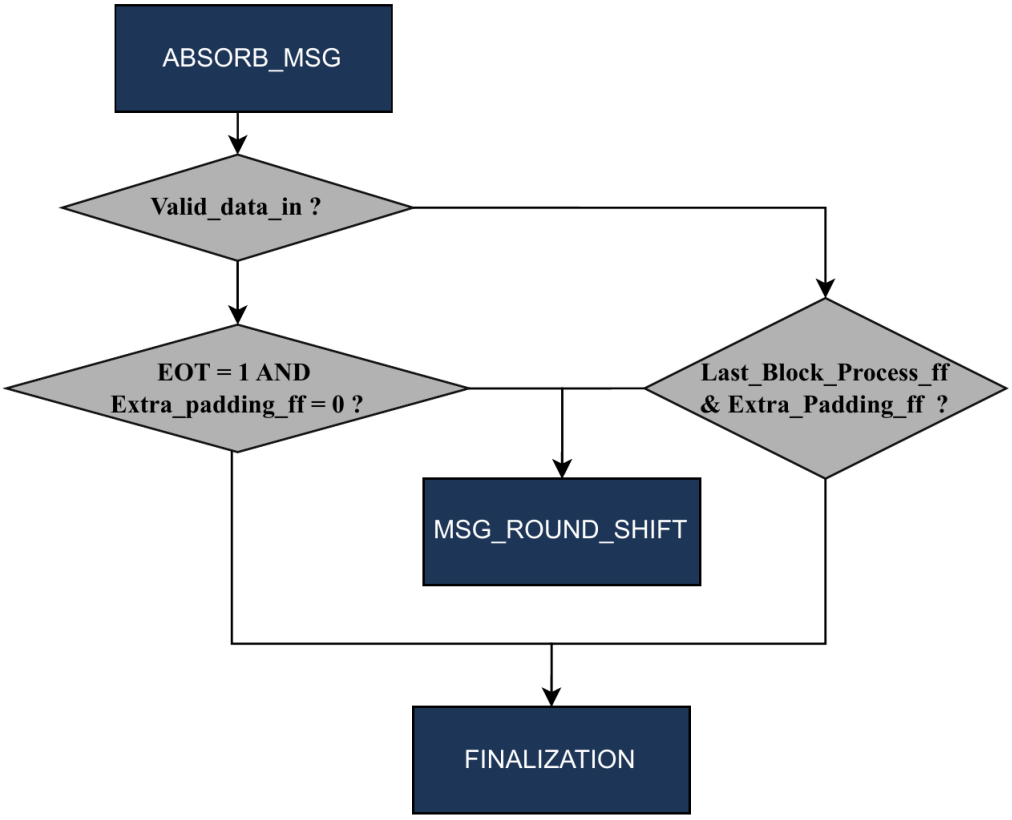


Figure 5.11: *PROCESS_MSG* exit condition

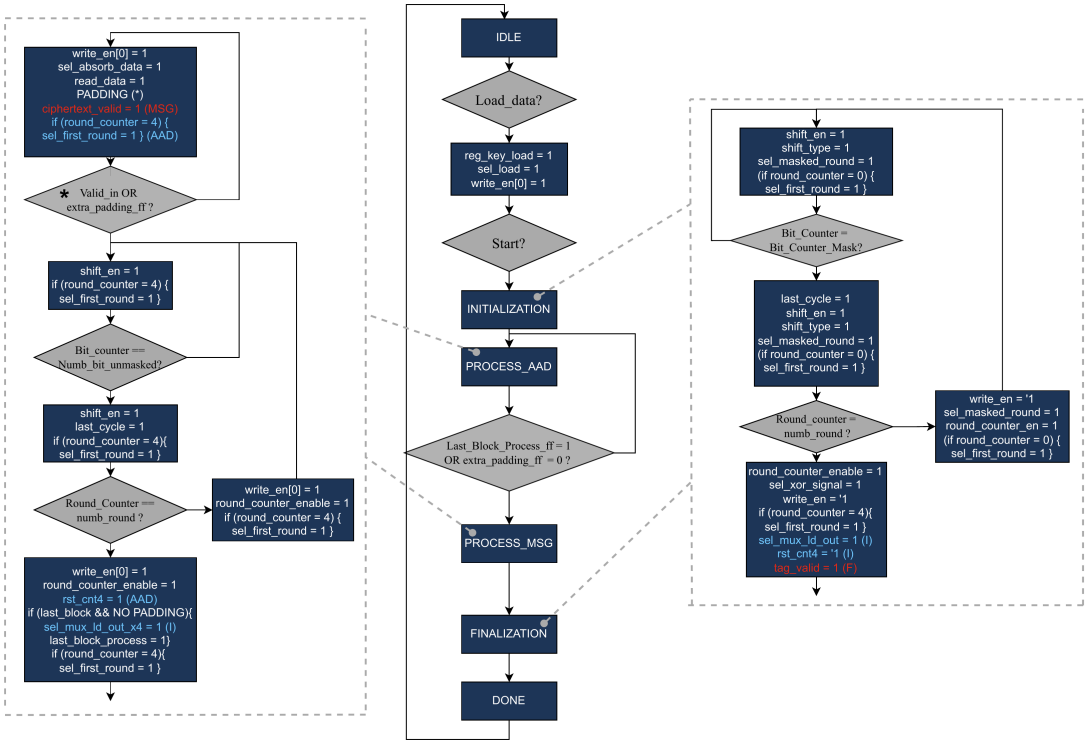


Figure 5.12: FSM output signals

5.4 Functional Verification

The functional verification of the proposed architecture was performed through a dedicated testbench implemented in C++, named `tb_auto.cpp`. The complete script will be made available in the public GitHub repository of this work (see Appendix A).
The testbench was designed to automate the entire verification flow. In particular, it performs the following tasks:

- it generates random values and lengths for both the AAD and MSG inputs;
- it feeds the generated data to the hardware core under test;
- it compares the produced output file with the results of the Python-based golden model, verifying that no mismatches occur.

This methodology ensured full functional coverage by testing the architecture under a wide range of random inputs, while guaranteeing correctness with respect to the official reference model.

6 Security Evaluation and TVLA

To assess whether a cryptographic implementation leaks sensitive information, the most widely adopted methodology is the *Test Vector Leakage Assessment* (**TVLA**), which relies on Welch's t-test. In this approach, two independent sets of side-channel traces are collected: one using a fixed nonce and the other using random nonce.

In order to perform such an evaluation, power consumption traces of the design must be collected. This requirement naturally leads to the integration of the cryptographic core on FPGA.

6.1 Integration on FPGA

We integrated the Ascon core on the CW305 Artix-7 target to enable controlled power measurements under realistic operating conditions, see Fig. 6.1 for the measuring set-up. The design adopts the ChipWhisperer USB front-end as a

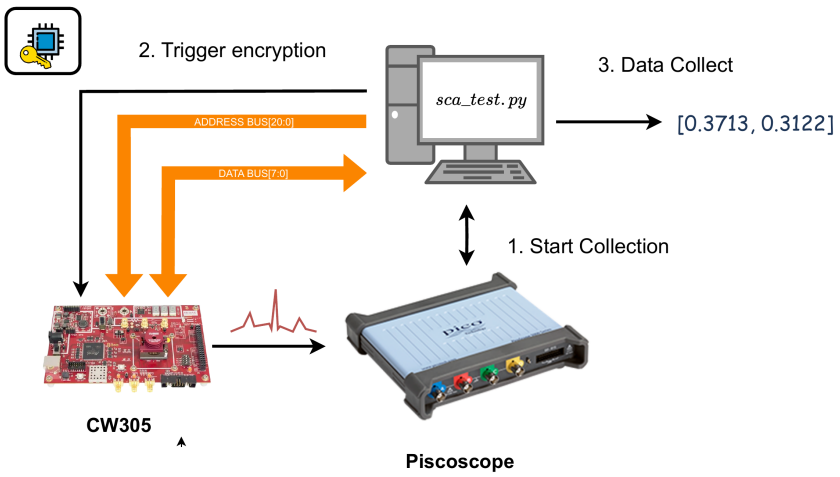


Figure 6.1: Experimental Set-up

minimal memory mapped bus, exposing a dedicated register bank. A clocking module derives the crypto clock from the on-board PLL; resets are provided both in hardware (pushbutton) and software (*rstn_sw*), see Fig. 6.2.

The **register bank**, implemented in the USB clock domain, manages the key, nonce, associated data (AD), and message (MSG) buffers, and provides the interface to the crypto core. It also generates a synchronized start pulse and latches the main status

signals (busy, ciphertext valid, tag ready, done). For completeness, the detailed functionality of each register is summarized in Table 6.1.

A **small FSM** operating in the crypto clock domain orchestrates the sequence of operations.The FSM is internally kept simple, as it is designed to support only a minimal functionality: both AAd and the MSG are assumed to be at most 16 bytes long. This design choice is motivated by the fact that, for the purpose of the TVLA evaluation, the primary interest lies in verifying that the critical phases of the algorithm, namely the **initialization** and **finalization**, do not exhibit any side-channel leakage. Consequently, the length of AAD and MSG can be arbitrary, so we opted for the simplest possible implementation.

In the IDLE state the FSM remains inactive. Once the host asserts the *load_data* signal, the FSM transitions to the LOAD_DATA state, during which the initialization data (IV, key, nonce) previously stored in the register bank are transferred into the core's internal state registers. It should be noted that the register bank must be loaded before the start of the operation since, as illustrated in Fig. 6.2, the data bus is only 8 bits wide and therefore constitutes a bottleneck.

After the LOAD_DATA state, the FSM enters the PROCESS_AD state, where the trigger signal for the oscilloscope is asserted (see Section 6.2). In this state, *msg_valid* and *msg_last* are activated, under the assumption that both AAD and MSG are exactly 128 bits. Subsequently, the FSM moves to the PROCESS_MSG state, which is analogous to the previous one, and finally transitions to the DONE state.
During the execution, the Ascon core produces the ciphertext, the authentication tag, and a 320-bit snapshot of the internal state, all of which can be retrieved over the USB interface for debugging and leakage analysis. Storing the ciphertext and tag also allows us, during the TVLA campaign, to validate that the core is operating correctly and that the cryptographic functionality is preserved while side-channel properties are being assessed.

This architecture provides a clean separation between control/IO (USB domain) and cryptographic processing (crypto domain), with explicit clock domain crossing and a stable trigger, which are essential to obtain high quality side-channel traces for TVLA.

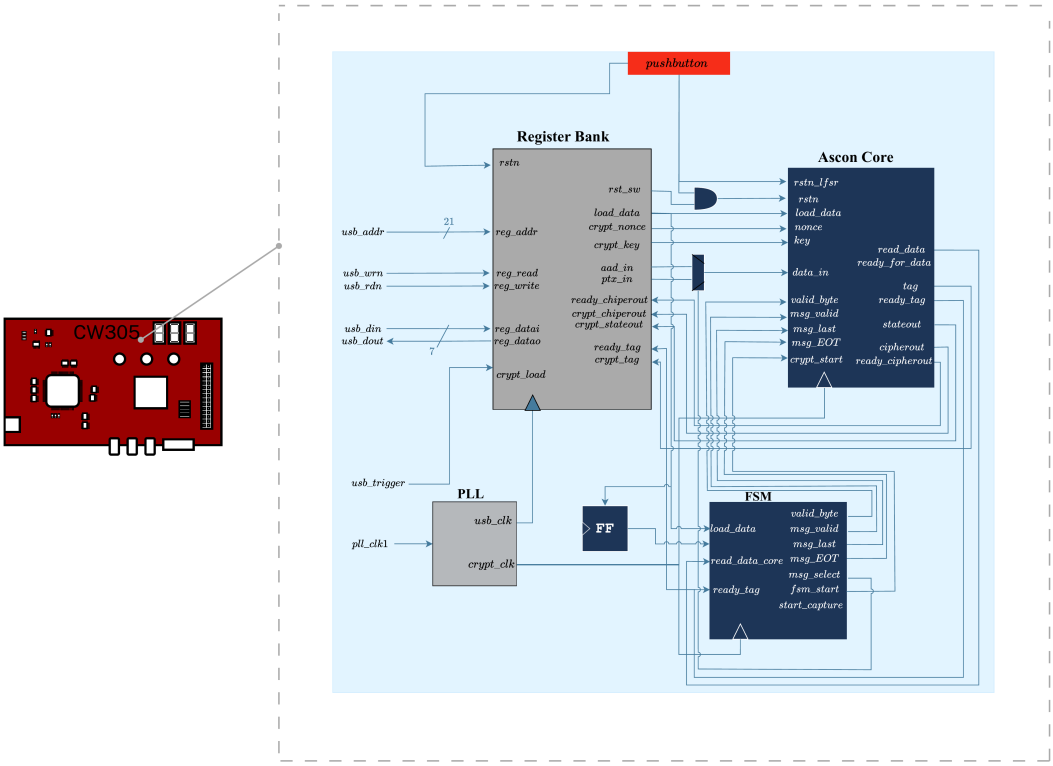


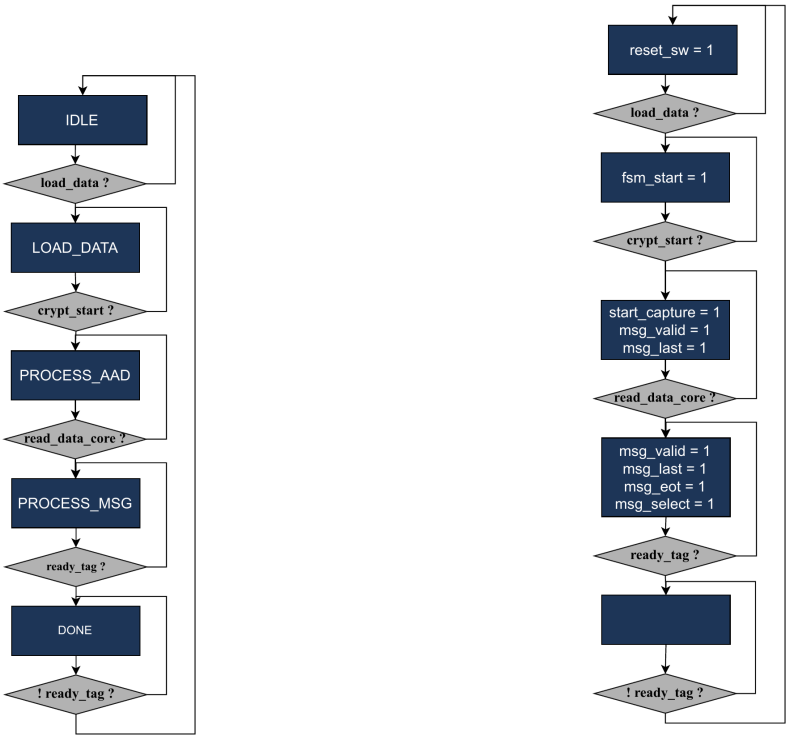
Figure 6.2: Ascon Core integrated on CW305 Artix-7 Board

6.2 Experimental Setup and Trace Acquisition

Measurements were carried out on the CW305 Artix-7 board with traces captured using a PicoScope 5000-series oscilloscope. The FPGA was clocked at 10 MHz, while the oscilloscope sampled at 100 MS/s, corresponding to 10 samples per FPGA clock cycle.

Trace acquisition was automated by means of a Python script (`sca_test.py`). This script first programs the FPGA with the correct bitstream, then loads the relevant inputs (key, nonce, AAD, and MSG) into the register bank via dedicated Python functions. Once all data have been written, bit 0 of the `REG_CRYPT_GO` register is set to 1, initiating the encryption operation. The oscilloscope trigger, as previously mentioned, is generated internally by the FPGA wrapper, thus avoiding the capture of power activity caused by register loading.

For our analysis, a total of 100,000 traces were collected: 50,000 with random nonces and 50,000 with a fixed nonce. In order to reset the core after each execution without clearing the register bank and the LFSR (used as a PRNG), an additional reset signal was introduced. This ensured that the randomness contribution differed across executions, while preserving the inputs already stored in the register bank, only the nonce is resend at each execution.
The t-scores were computed online using SCALIB [30].



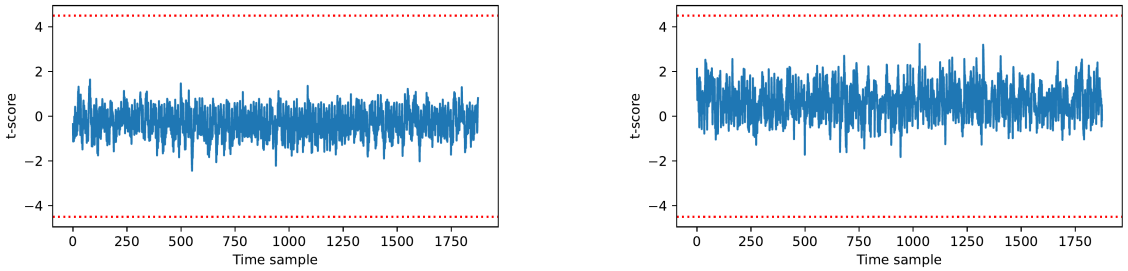
(a) FSM state transition (b) FSM output signals

Figure 6.3: FSM integrated on FPGA

6.3 TVLA Results and Interpretation

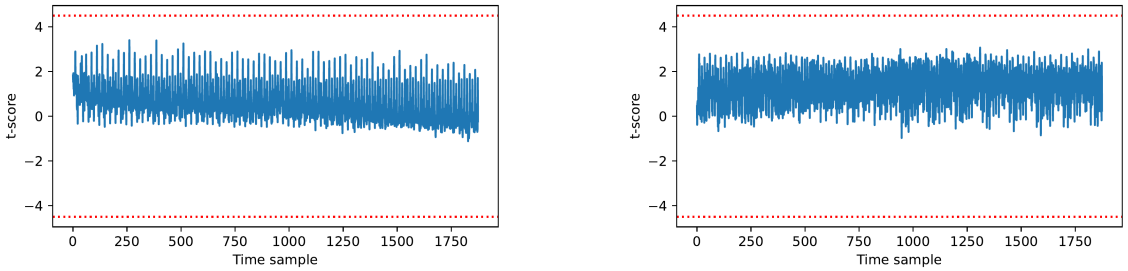
6.3.1 TVLA on S-BOX

In the first phase of development, the TVLA test was performed exclusively on the S-box and the share generation module. The results demonstrate that both the first order and second order designs are secure, with no evidence of leakage in either architecture, see Fig. 6.4, 6.5. This preliminary evaluation was carried out to ensure that the modified S-box, presented in Fig. 5.3, did not introduce side-channel leakage prior to integrating it into the complete design.



(a) 1^{st}-order t-test (b) 2^{nd}-order t-test

Figure 6.4: TVLA result of the first order masked S-box.



(a) 1^{st}-order t-test (b) 2^{nd}-order t-test

Figure 6.5: TVLA result of the second-order masked S-box.

6.3.2 TVLA on the Complete Architecture

Subsequently, the TVLA test was extended to the complete encryption architecture, both for the first-order and second order implementations. In this phase, the attack was carried out over the entire cryptographic computation, including initialization, absorption of AAD and message, and finalization.

Both architectures were instantiated with `PAR_MAX`:

- for the first order case with PAR = 32,
- for the second order case with PAR = 22.

The number of samples required during the initialization/finalization phase was estimated as:

$$n_{\text{samples\_init}} = \left(12 \cdot \left(\sup\left(\frac{64}{PAR}\right) + 3\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$

while for the absorption of AAD and MSG:

$$n_{\text{samples\_process}} = \left(8 \cdot \left(\sup\left(\frac{64}{PAR \cdot (d+1)}\right) + 2\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$

where:

- PAR denotes the parallelism factor,
- d represents the protection order,
- f_{sample} is the sampling frequency,
- f_{clock} is the design clock frequency.

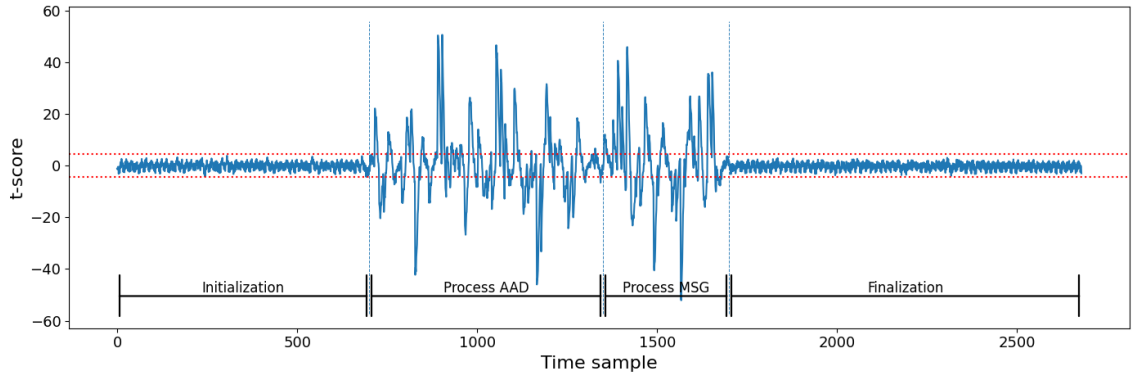
Numerical Results

Case $d = 1,\ PAR = 32$:

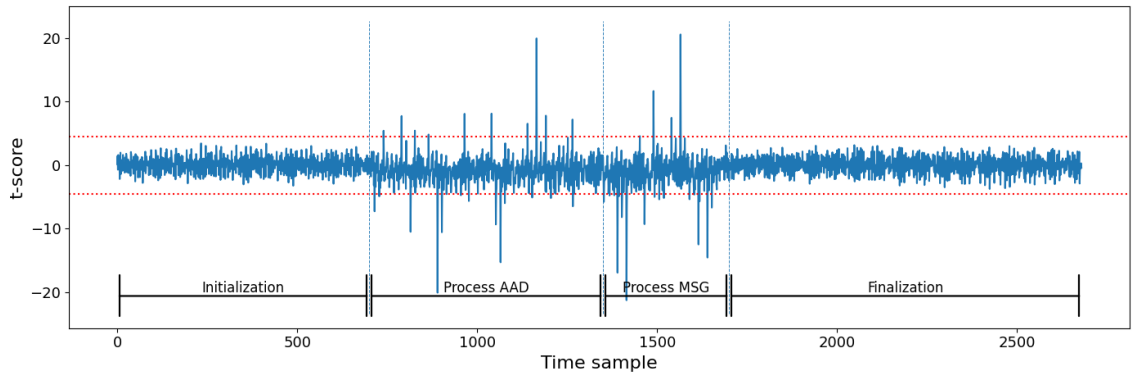
$$n_{\text{samples\_init}} = \left(12 \cdot \left(\sup\left(\frac{64}{32}\right) + 3\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}} = 61 \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$

$$n_{\text{samples\_process}} = \left(8 \cdot \left(\sup\left(\frac{64}{32 \cdot (1+1)}\right) + 2\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}} = 25 \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$

It is worth noting that, while for the initialization and finalization phases the computed number of samples matches the experimental value (i.e., the point at which the *sel_masked_round* signal goes to 0), highlighted by the first vertical line in Fig. 6.6, the duration of the PROCESS_AAD phases appears to be approximately twice as long. This behavior is nevertheless correct, since in the case where both AAD and MSG consist of 16 bytes, an additional round of `extra_padding` is required, effectively doubling the processing time, instead for the MSG is not required beacouse after the `ABSORB` of the extra_padding the p^B permutation are not required.
It is worth noting that, while for the initialization and finalization phases the computed number of samples matches the experimental value (i.e., the point at which the *sel_masked_round* signal goes to 0), highlighted by the first vertical line in Fig. 6.6, the duration of the `PROCESS_AAD` phase appears to be approximately twice as long. This behavior is nevertheless correct, since when both AAD and MSG consist of 16 bytes, an additional round of `extra_padding` is required, thereby effectively doubling the processing time. Conversely, for the `PROCESS_MSG` phase the additional round is not required, as after the absorption of the `extra_padding` the p^B permutation is not executed.



(a) 1^{st}-order t-test



(b) 2^{nd}-order t-test

Figure 6.6: TVLA result of the first-order masked implementation.

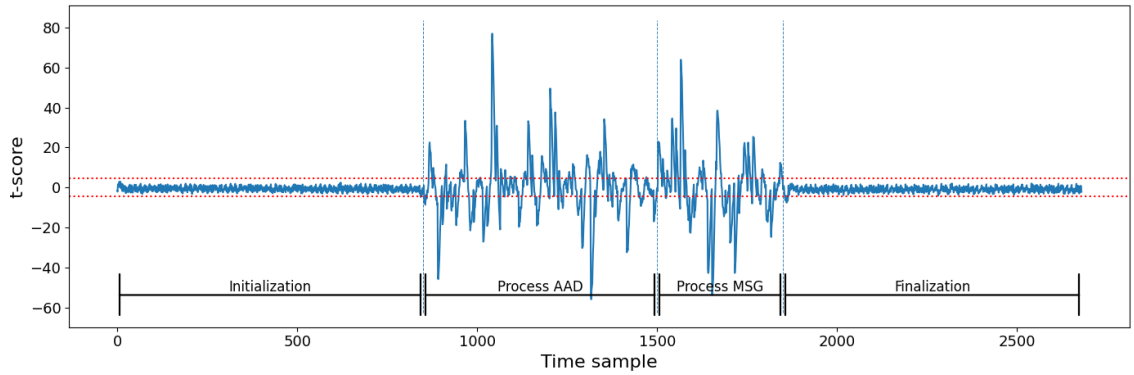
From a theoretical perspective, one would expect that the second-order TVLA applied to the first-order architecture should reveal leakage, thus indicating insecurity. However, as can be observed in both figures (Fig. 6.6), during the initialization and finalization phases the designs appear secure, since no significant leakage is detected.

On the other hand, during the `PROCESS_AAD` and `PROCESS_MSG` phases, the TVLA values exceed the threshold. This result is nevertheless consistent with the theoretical model: these phases do not induce sensitive leakage related to the secret key. Indeed, the TVLA test is a non-specific statistical method aimed at detecting differences in power traces rather than demonstrating the feasibility of an attack. In this case, the observed differences are explained by the fact that the processed data depends on the `NONCE`, which is fixed in one half of the trace and variable in the other. Since the shares have been recombined and are no longer refreshed with randomness, it is expected that the TVLA highlights such variations as a form of apparent "leakage," even though it does not compromise key security.

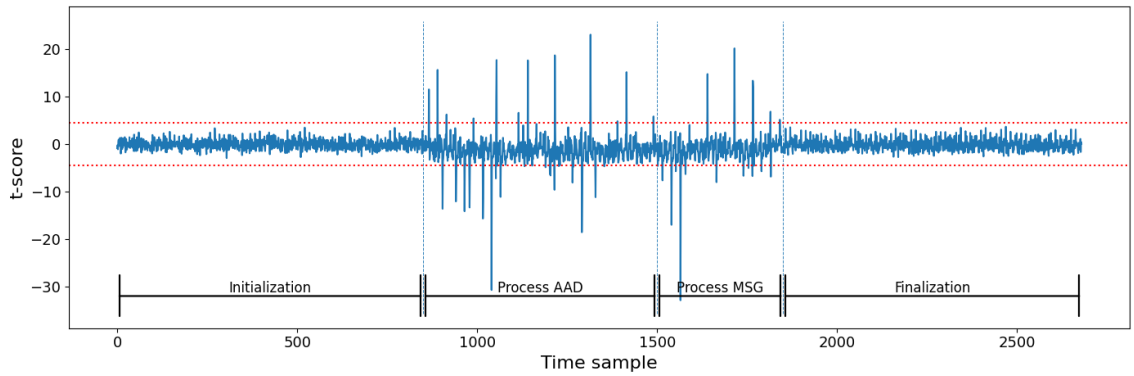
Case $d = 2,\ PAR = 22$**:**

$$n_{\text{samples\_init}} = \left(12 \cdot \left(\sup\left(\frac{64}{22}\right) + 3\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}} = 73 \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$

$$n_{\text{samples\_process}} = \left(8 \cdot \left(\sup\left(\frac{64}{22 \cdot (2+1)}\right) + 2\right) + 1\right) \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}} = 25 \cdot \frac{f_{\text{sample}}}{f_{\text{clock}}}$$



(a) 1^{st}-order t-test



(b) 2^{nd}-order t-test

Figure 6.7: TVLA result of the first order masked implementation.

Address (hex)	Register	Purpose / Notes
0x00	REG_CLKSETTINGS	Not modified from the original CW305 wrapper.
0x01	REG_USER_LED	Not modified from the original CW305 wrapper.
0x02	REG_CRYPT_TYPE	Not modified from the original CW305 wrapper (design identifier).
0x03	REG_CRYPT_REV	Not modified from the original CW305 wrapper (design revision).
0x04	REG_IDENTIFY	Not modified from the original CW305 wrapper (board identifier).
0x05	REG_CRYPT_GO	Used to assert the load_data signal, triggering the start of a new operation in the Ascon core.
0x06	REG_CRYPT_TEXTIN	Register used to load the Associated Data (AD) to the core.
0x07	REG_CRYPT_CIPHERIN	Register used to load the ciphertext when operating in decryption mode.
0x08	REG_CRYPT_TEXTOUT	Register used to retrieve the plaintext produced by the decryption.
0x09	REG_CRYPT_CIPHEROUT	Register used to retrieve the ciphertext produced during encryption.
0x0A	REG_CRYPT_KEY	Register used to store the secret key.
0x0B	REG_BUILDTIME	Not modified from the original CW305 wrapper.
0x0C	REG_CRYPT_TAGOUT	Register used to store the authentication tag produced by the Ascon, readable via the USB interface.
0x0D	REG_CRYPT_NONCEIN	Register used to store the nonce supplied by the host.
0x0E	REG_CRYPT_STATEOUT	Register used to expose the internal state of the Ascon.
0x0F	REG_VALID_BYTES_AD	Register used to indicate the number of valid bytes in the AD.
0x10	REG_VALID_BYTES_MSG	Register used to indicate the number of valid bytes in the MSG.
0x11	REG_CRYPT_TEXTIN_BUFFER_MSG	Register used to store the MSG.
0x12	REG_CRYPT_STATUS	Register exposing status flags: used both for debugging (e.g., during development) and for detecting when ready_tag is asserted, signaling that the result can be read.

Table 6.1: Register bank map for the CW305 Ascon integration.

7 Results and Analysis

Modern lightweight cryptography must deliver not only provable side–channel robustness but also competitive silicon efficiency. To quantify the silicon cost of our masked Ascon designs, we synthesized the RTL with *Synopsys Design Compiler* (targeting the `tcbn65lplvt` 65 nm standard–cell library at the `[TT/SS/FF]` corner, `[Vdd = 1.2 V]` and `[T=25 °C]`. All figures are *core-only* (I/O pads and memories excluded). Gate equivalents (GE) are normalized to the two–input NAND cell.[1] Synthesis used a single clock domain and a flat hierarchy; unless otherwise stated, the same constraints and clock–gating options were applied to all variants to ensure a fair comparison.

7.1 ASIC Area Results

Tables 7.1 and 7.2 report the DC area breakdown for the first– and second–order implementations at two parallelism points: PAR = 1 (minimal parallelism) and PAR_{MAX} (32 for d=1, 22 for d=2) in Gate Equivalent. The categories follow DC's *report_area* convention: combinational logic, buffer/inverter cells, and non–combinational (sequential) cells.

Group	kGE (PAR=1)	kGE (PAR=32)
Combinational	9.41	15.26
Buffers / Inverters	0.76	1.22
Non–combinational	4.90	9.29
Total	**14.31**	**24.55**

Table 7.1: GE breakdown for the first–order masked core ($d = 1$) at PAR=1 and $\text{PAR}_{\text{MAX}} = 32$.

Group	GE (PAR=1)	GE (PAR=22)
Combinational	11.29	17.54
Buffers / Inverters	1.03	1.48
Non–combinational	6.80	12.71
Total	**18.09**	**30.25**

Table 7.2: GE breakdown for the second–order masked core ($d = 2$) at PAR=1 and $\text{PAR}_{\text{MAX}} = 22$.

[1] Extracted from `tcbn65lplvttc.db.alib`, cell `ND2D0LVT`.

Finally, we provide a compact comparison across masking orders, using the maximum parallelism we deploy per order $(32, 22, 16, 11)$ for $(d = 1, 2, 3, 5)$, respectively). Fig. 7.1 illustrates how the area grows with the masking order d when configured for maximum throughput (one round per cycle). Notably, the incremental overhead of moving from one masking order to the next remains moderate.

Implementation	d	PAR_{MAX}	**kGE**
First–order masked core	1	32	18.09
Second–order masked core	2	22	30.25
Third–order masked core	3	16	33.76
Fifth–order masked core	5	11	47.26

Table 7.3: Gate–equivalent (GE) counts at PAR_{MAX}. Absolute area figures are omitted due to foundry library NDA.

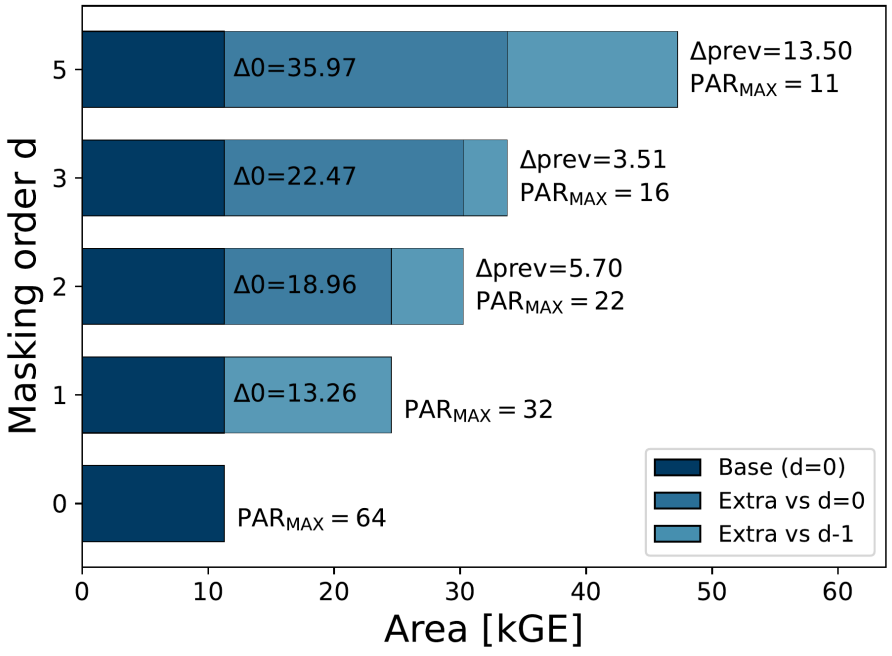


Figure 7.1: Area vs. masking order at PAR_MAX.

7.1.1 Comparison Against the State of the Art

Table 7.4 benchmarks our gate-equivalent (kGE) figures against recent state-of-the-art (SoA) implementations across masking orders $d \in \{1, 2, 5\}$. A clear trend emerges: *as the masking order increases, our* ***relative*** *area savings over the SoA become progressively larger*, see Fig. 7.2. This behavior is intuitive for our micro-architecture, which deliberately reduces the effective parallelism (PAR) at higher orders and thus instantiates fewer parallel S-boxes; the dominant nonlinear cost is therefore amortized more aggressively than in highly parallel SoA designs that replicate the S-box network

per share. While this confirms the expected area efficiency, the more insightful metric for hardware practicality is the *throughput-per-area* ratio (Mbps/kGE), which we analyze in the next subsection.

Order d	Our Design	[31]	[32]	[23]	[33]
1	24.5	30.42	42.8	26.1	50.4
2	30.2	–	90.9	52.6	102.4
5	47.3	–	339.8	–	3557.7

Table 7.4: Gate equivalents (kGE) by masking order d (rows) and work (columns). '–' denotes not reported.

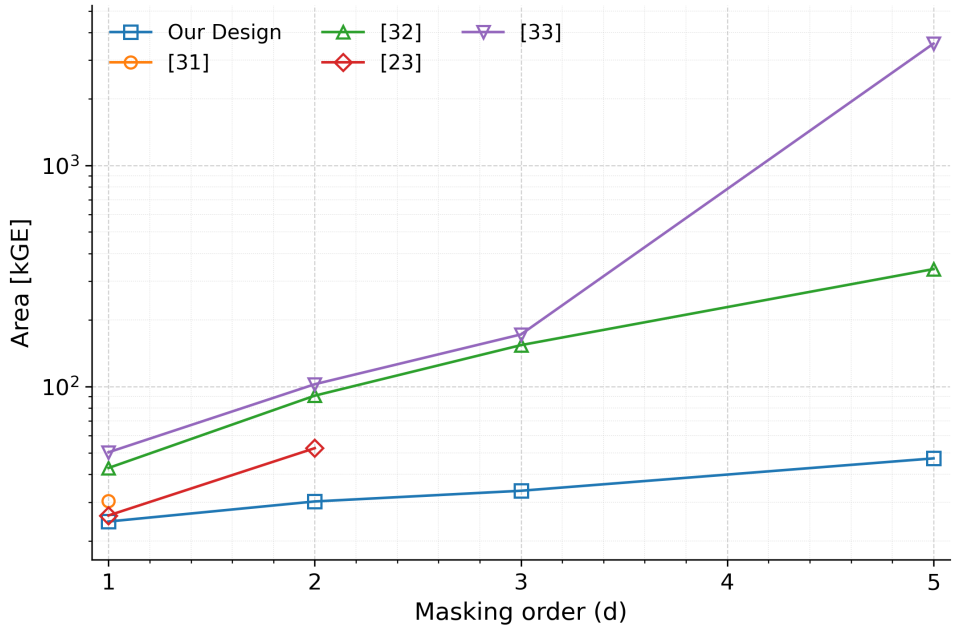


Figure 7.2: Area result (kGE) vs masking order d

7.2 ASIC Throughput Results

We report throughput for our ASIC as

$$\mathrm{TH} = \frac{\text{processed bits}}{N_{\text{cycles}} \cdot t_{\text{cp}}} = f_{\max} \cdot \frac{\text{processed bits}}{N_{\text{cycles}}}, \tag{7.1}$$

where N_{cycles} is the total number of clock cycles to complete an AEAD operation under a given MSG and AAD length, t_{cp} is the critical path (clock period), and $f_{\max} = 1/t_{\text{cp}}$. Unless otherwise stated, "processed bits" refers to the bits of payload (MSG) and associated data (AAD) actually absorbed by the AEAD datapath (authentication tag bits are excluded from the numerator but their generation/verification contributes to N_{cycles}). For a sponge/duplex AEAD such as ASCON, N_{cycles} naturally decomposes as

$$N_{\text{cycles}} = N_{\text{init}} + N_{\text{aad}}(|A|) + N_{\text{msg}}(|P|) + N_{\text{final}}, \tag{7.2}$$

with $|A|$ and $|P|$ the AAD and plaintext lengths. The initialization and finalization terms are fixed per operation, whereas N_{aad} and N_{msg} scale with the number of absorbed rate blocks.[2] As a consequence, for short packets the fixed $N_{\text{init}} + N_{\text{final}}$ dominates and depresses the effective throughput, while for long packets the amortized cost of these phases vanishes and the throughput approaches the architecture's steady state limit $f_{\max} \cdot$ (bits per cycle in the processing phase).

Application profiles and lengths. We evaluate three representative profiles that cover typical AEAD deployment ranges:

- **IoT–small (telemetry)**: very short payloads with compact headers. As a concrete representative we consider $MSG \approx 16\,\text{B}$ ($\approx 128\,\text{bits}$) and AAD in the order of a small MAC/transport/application header (e.g., LoRaWAN MAC fields) ≈ 13–$16\,B$. This reflects widely used low-power telemetry where messages are only tens of bytes and the LoRaWAN MAC adds a $\approx$13 B overhead when no options are present [33].

- **Wireless–medium (Wi-Fi/802.11 data)**: moderate payloads with MAC header authenticated as AAD. We adopt $AAD \approx 22$–$30\,\text{B}$ (portions of the 802.11 MAC header treated as AAD in AEAD modes) and a MSG in the ≈ 250–$300\,B$ range, representative of small application datagrams that are common on WLANs [34].

- **Ethernet-like (L2/L3/L4 data)**: full-MTU payload with lower-layer headers authenticated. A practical composition is Ethernet II (14 B) + IPv4 (20 B min) + UDP (8 B) $\Rightarrow AAD \approx 42 - 128\,\text{B}$ and $MSG \approx 1500\,\text{B}$ (classic Ethernet MTU payload) [34].

Why our reconfigurable core benefits from larger AAD/MSG. Our core can reconfigure to sustain the maximum absorption/processing rate during both the AAD and MSG phases. In all three profiles this means that, once initialization completes, the datapath operates at its highest utilization for the bulk of the work. The fixed costs of *Initialization* and *Finalization* are therefore amortized over more rate blocks whenever $|A|$ and $|P|$ are larger, yielding (i) higher effective throughput and (ii) smaller throughput variance across inputs. This effect is most visible in the Ethernet-like profile and, to a lesser extent, in the wireless-medium profile; it is least pronounced in the IoT-small profile where $N_{\text{init}} + N_{\text{final}}$ constitutes a larger share of N_{cycles}.

For throughput calculation we adopted the theoretical expression in Equation 7.1 and verified its consistency with simulation results by measuring the number of clock cycles N_{cycles} between the rising edge of the *start* signal and the assertion of *tag_valid*.
The cycle count was modeled as follows.

[2] For ASCON-128 and ASCON-128a the data rate r is 64 and 128 bits, respectively, hence N_{aad} and N_{msg} grow with $\lceil |A|/r \rceil$ and $\lceil |P|/r \rceil$.

Initialization and Finalization. These phases require the full execution of the 12 permutation rounds, with a cycle count given by Eq. 7.3.

$$N_{\text{init/final}} = 12 \cdot \left(\frac{64}{\text{PAR}} + \text{LATENCY} \right). \tag{7.3}$$

Here, PAR denotes the datapath parallelism (i.e., the number of bits processed per cycle), while LATENCY accounts for the pipeline depth. As illustrated in Fig. 5.1, and considering the internal flip-flop stage present within the DOM-AND block during this phase, the LATENCY is fixed to 3.

AAD and MSG absorption. During the absorption phase, each r-bit block (with $r = 64$ or 128, depending on the variant) is processed over eight rounds. In this case, the flip-flop stage inside the DOM-AND is bypassed, which reduces the effective pipeline depth. Consequently, the cycle count per block is

$$N_{\text{absorb}} = 8 \cdot \left(\frac{64}{\text{PAR} \cdot (d+1)} + \text{LATENCY} - 1 \right) + 1, \tag{7.4}$$

where PAR denotes the datapath parallelism, d the masking degree, and LATENCY the pipeline depth. The additional +1 term, compared to the *Init/Final* case, accounts for the explicit absorption of the message block.
The overall contribution of AAD and MSG phases then scales linearly with the number of absorbed rate blocks:

$$N_{\text{aad/msg}} = N_{\text{absorb}} \cdot \left\lceil \frac{|A|}{r} \right\rceil + N_{\text{absorb}} \cdot \left\lceil \frac{|P|}{r} \right\rceil. \tag{7.5}$$

At this point, the only remaining parameter required to compute the throughput is the critical path delay t_{cp}. This value was obtained by issuing the `report_timing` command in Synopsys Design Compiler. The resulting t_{cp} values, together with the corresponding throughput figures, are summarized in Tab. 7.5.

APPLICATION		**IOT**		**WIRELESS**		**ETHERNET**	
masking order	t_{cp} [ns]	N_{cycle}	TH [Gb/s]	N_{cycle}	TH [Gb/s]	N_{cycle}	TH [Gb/s]
$d = 1$	0.70	195	0.94	647	5.32	2547	6.74
$d = 2$	0.78	219	0.75	660	4.60	2569	5.99
$d = 3$	0.80	243	0.66	693	4.33	2593	5.78
$d = 5$	0.86	291	0.51	741	3.77	2641	5.28

Table 7.5: Throughput (TH) as a function of masking degree and application (IoT, Wireless, and Ethernet) at PAR_{MAX}.

The results were obtained assuming data blocks of 16 B and 16 B for the IoT case, 30 B and 300 B for the Wireless case, and 42 B and 1500 B for the Ethernet case.

As shown in Fig. 7.3, the throughput (TH) trend as a function of parallelism for different masking degrees is reported for **ETHERNET** applications. It can be

observed that the curves terminate at the maximum supported parallelism, for the reasons already discussed in Section 5.1. It is also noticeable that, for certain values of PAR, the curve becomes nearly flat. This behavior is due to the fact that increasing the parallelism from one value to the next does not reduce the number of required shifts. For instance, in the case of $d = 2$, moving from $PAR = 16$ to $PAR = 20$ still requires four shifts during the initialization/finalization phase and two shifts during the message processing phase. We can observe in the IoT case (Fig. 7.4)

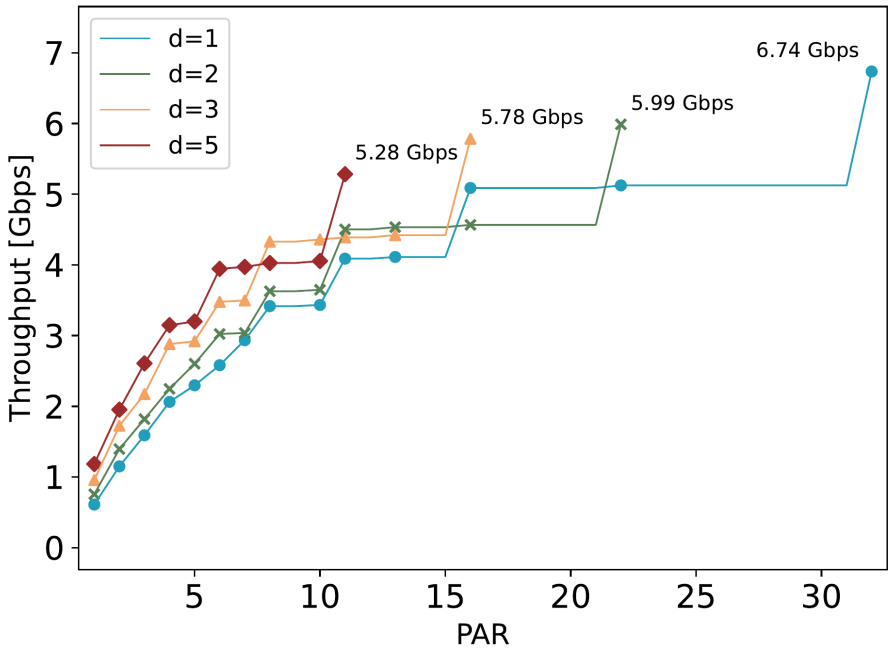


Figure 7.3: Throughput (TH) varying PAR and d for Ethernet application

that initialization and finalization phases dominate the overall execution time. This effect arises because, with only two blocks of AAD and MSG to process, the fixed initialization cost is amortized over a very small number of blocks. Consequently, the dominant factor becomes the parallelism itself, which directly determines the number of cycles required in these phases, while the contribution of the term $\text{PAR} \cdot (d + 1)$ in the message and AAD processing is comparatively less significant.

For conciseness, detailed throughput–parallelism plots for the **Wireless** profile is omitted from the main text and reported in Appendix B.

7.3 ASIC TH/Area Results

Having obtained the throughput (TH) and area values, we also report their ratio Tab. 7.6, $\text{TH/A} = \text{TH}/A$ (in Mb/s per kGE), as a size-normalized figure of merit. TH/A quantifies how effectively the architecture converts silicon area into delivered throughput: the higher the value, the more throughput is achieved per unit area (equivalently, the smaller the area required per Mb/s). This normalization enables a fair comparison across masking degrees and parallelism settings, where area may

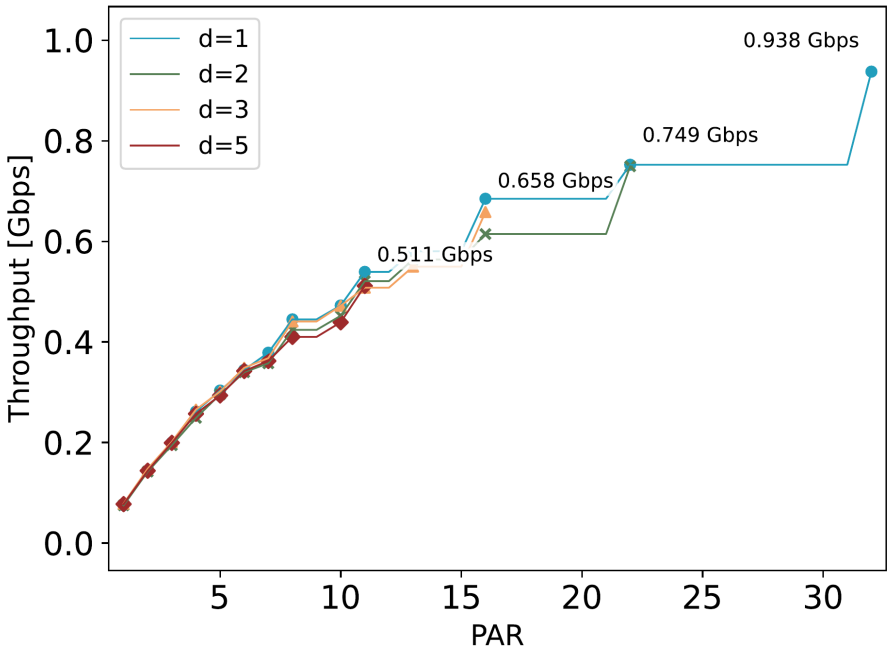


Figure 7.4: Throughput (TH) varying PAR and d for IoT application

grow faster than throughput, and will be used in the next section to position our results against prior masked ASCON implementations.

Table 7.6: Area, throughput (TH) and throughput-per-area (TH/A) vs. masking degree at $PAR_{\max}$.

		TH [Gb/s]			TH/A [Mb/s per kGE]		
Mask degree	**Area [kGE]**	**IoT**	**Wireless**	**Ethernet**	**IoT**	**Wireless**	**Ethernet**
$d = 1$	24.5	0.94	5.32	6.74	38.4	217.6	275.5
$d = 2$	30.2	0.75	4.60	5.99	24.8	152.3	198.3
$d = 3$	33.8	0.66	4.33	5.78	19.5	128.1	171.0
$d = 5$	47.3	0.51	3.77	5.28	10.8	79.7	111.7

TP/A is computed as TH/Area with Area in kGE; values rounded to one decimal.

7.3.1 Comparison Against the State of the Art

As shown in Tab. 7.7, the table reports the maximum throughput-to-area ratios (TH/A) of our architecture compared with state of the art masked Ascon implementations. While the results may appear very favorable, and in many respects they are, several important remarks must be made.
First, the table is constructed to highlight the *maximum achievable throughput*, i.e., assuming a continuous stream of data blocks and neglecting the fixed overhead of initialization and finalization. In other words, the reported values represent the *asymptotic throughput* that is reached for sufficiently long messages. This convention

is widely adopted in the hardware cryptography literature. The throughput values are obtained using the following expression:

$$\mathrm{TP} = \frac{f_{ck} \cdot size(P_i)}{\mathrm{Latency}_{round}} \tag{7.6}$$

where $size(P_i)$ is the size of the plaintext block processed per cycle, f_{ck} is the maximum operating frequency, and $\mathrm{Latency}_{round}$ is the number of cycles per round. Experimental results demonstrate that our design consistently achieves superior TH/A across all masking orders. This advantage comes from the re-use of the masking hardware during message processing and from the deeper pipelining of combinational logic, which shortens the critical path and allows higher maximum clock frequencies.
It must be emphasized, however, that the initialization/finalization overhead is not captured by this formula. While this overhead is present in all implementations, it is more pronounced in our design. This explains why the values in the table should be interpreted as asymptotic throughput, whereas short-message scenarios (e.g., IoT profiles) exhibit lower effective throughput.

The most important aspect to highlight, however, is the trend of the throughput to area ratio (TH/A) as the masking order increases. In competing designs, moving from first- to fifth-order masking typically results in a reduction of TH/A by a factor of about 7–8. In contrast, our architecture shows a much milder degradation, with TH/A decreasing only by a factor of roughly 2.5. This clearly underlines the efficiency and scalability of our approach, especially when higher-order protection is required.

Finally, we note that in the Tab. 7.7 the reported latency for our design is written as

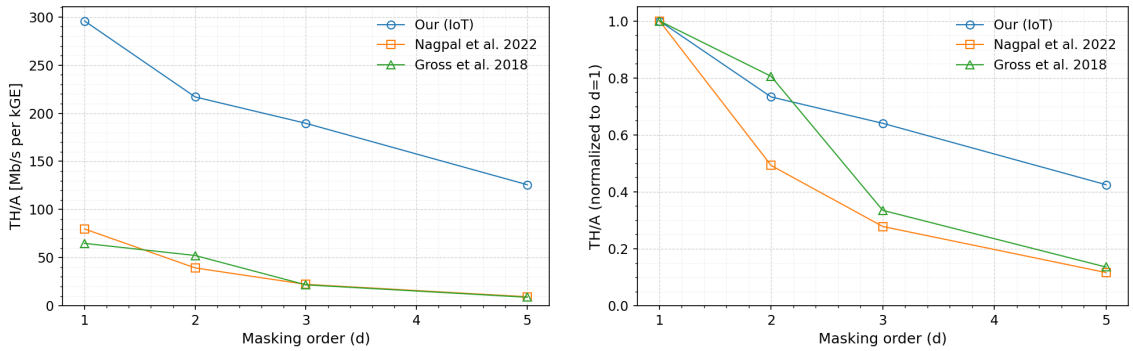
Table 7.7: Comparison with State-of-the-Art Ascon masked implementations.

Work	**Protection Order**	**Area** [kGE]	**Randomness** [bit/cycle]	**Latency** [cycles/round]	**TH** [Gbps]	**TH/A** [Mbps/kGE]
Our Design	1-order	24.5	0*	2+1	7.31	298.4
	2-order	30.2	0*	2+1	6.56	217.2
	5-order	47.3	4800	2+1	5.95	125.8
[31]	1-order	30.42	4	2	3.77	124
[33]	1-order	50.4	320	1	4.35	79.8
	2-order	102.4	960	1	4.02	39.3
	5-order	3557.7	4800	1	3.34	9.3
[32]	1-order	42.8	2048	1	2.77	64.8
	2-order	90.9	4608	1	3.34	52.2
	5-order	339.8	18432	1	2.99	8.8
[23]	1-order	26.1	0*	2	n.d.	n.d.
	2-order	52.6	0*	2	n.d.	n.d.

* Obtained with *changing of the guards* technique

"2+1". This notation indicates that during the message processing phase only two

sequential stages (flip-flops/state registers) are traversed, while in the initialization and finalization phases one additional register stage must be accounted for, corresponding to the internal register of the DOM-AND gate.
The same trends are visualized in Fig. 7.5, which presents two complementary views. The Fig. 7.5a a reports the absolute throughput-per-area (TH/A) across masking orders for our design and prior masked ASCON implementations. Fig. 7.5b normalizes each curve to its own first–order value, so as to emphasize the relative degradation with increasing masking order rather than absolute magnitude. The normalized view makes the trend immediately comparable across architectures and corroborates the discussion above: state-of-the-art designs typically lose a larger fraction of TH/A when moving to higher orders, whereas our architecture exhibits a markedly milder decline.



(a) Absolute TH/A vs. masking order.

(b) TH/A normalized to the d=1 value of each architecture.

Figure 7.5: Comparison of throughput-per-area (TH/A) across masking orders: absolute values (a) and normalized trends (b).

8 Conclusion and Future Work

8.1 Conclusions

This thesis presented an efficient hardware implementation of ASCON-128/ASCON-128a that is fully parameterizable in the datapath parallelism (PAR) and in the masking degree (d). The proposed architecture reuses masking hardware during both AAD and MSG phases and employs deeper pipelining on the non-linear layer, which shortens the critical path and enables higher $f_{\max}$. An analytical cycle model was developed and validated against simulation to derive throughput across representative IoT, Wireless, and Ethernet profiles.

When compared with prior masked ASCON designs, our core attains competitive and often superior throughput-per-area (TH/A), particularly for larger payloads where initialization/finalization costs are amortized.

8.2 Future Work

(1) Exploit all $(d{+}1)^2$ masked ANDs to increase effective parallelism. In the current core the masked non-linear layer exploits $(d{+}1)$ parallel DOM-ANDs per cycle. An alternative is to utilize *all* $(d{+}1)^2$ pairwise ANDs available in DOM, thereby reducing the number of S-box instances required per 64-bit rate from $\lceil 64/(d{+}1) \rceil$ to $\lceil 64/(d{+}1)^2 \rceil$. This could further increase TH/A at higher orders. The approach, however, requires a non-trivial refactoring of the core.

(2) One-cycle reduction per round via register placement and scheduling. A second direction is to reorganize the round pipeline so that the masked DOM-AND is applied immediately after the state registers. By doing so, the flip-flop layer currently used to guarantee input independence for the DOM-AND could be removed, reducing the per-round latency from (L) to $(L{-}1)$ clock cycles (in our prototype, from 3 to 2). The expected benefit is higher throughput at the same PAR and d. This optimization must be weighed against a likely increase of the critical path; it also demands a new Python golden model to trace intermediate masked states and a refreshed verification flow (functional, formal where applicable, and side-channel leakage assessment).

A GitHub Directory

NicoPaninforni/myAscon128-A

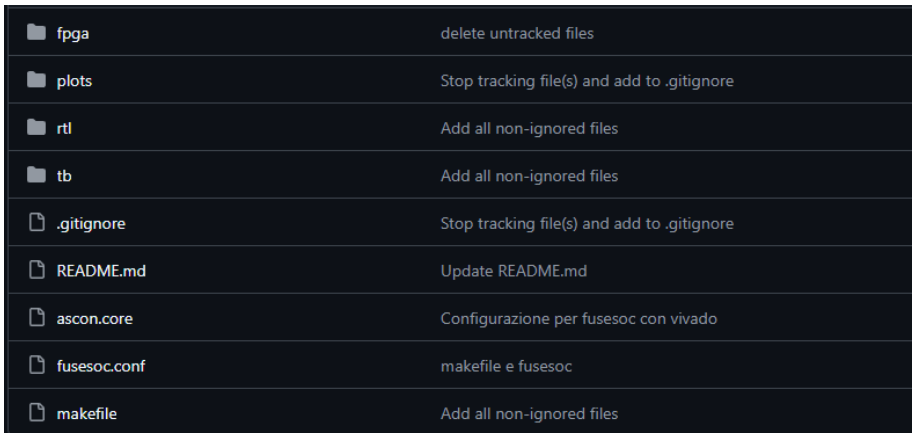


Figure A.1: Repository structure of the ASCON-128 RTL project on GitHub

`rtl/` Register-transfer-level source code (Verilog/SystemVerilog) for the ASCON-128 core. Includes design parameters (e.g., `ascon_params.sv`) used by simulation and synthesis.

`tb/` Testbench sources for RTL simulation with Verilator (e.g., `tb_auto.cpp`). The testbench generates randomized inputs, drives the DUT, and checks outputs against the Python golden model.

`fpga/` FPGA-oriented wrapper and integration files, with a dedicated testbench in `fpga/tb/`. The current wrapper targets up to 128-bit AAD and message lengths and produces waveform dumps for inspection.

`plots/` Staging area for figures and analysis artifacts (e.g., plots or summaries produced during verification or synthesis evaluation).

`.gitignore` Ignore rules for build artifacts, generated files (e.g., VCD/FST traces, netlists), and other temporary outputs.

`README.md` High-level usage notes and workflow overview for simulation, synthesis, and verification.

`ascon.core` FuseSoC core description: declares file sets, targets (simulation/synthesis), and tool backends, enabling reproducible builds of the ASCON core.

`fusesoc.conf` Local FuseSoC configuration (e.g., core library paths and backend options) used by the automated flow.

`makefile` Entry points for the automated flow (simulation, ASIC synthesis, and post-synthesis simulation), orchestrating FuseSoC/Verilator and Design Compiler runs.

B Throughput Graphs

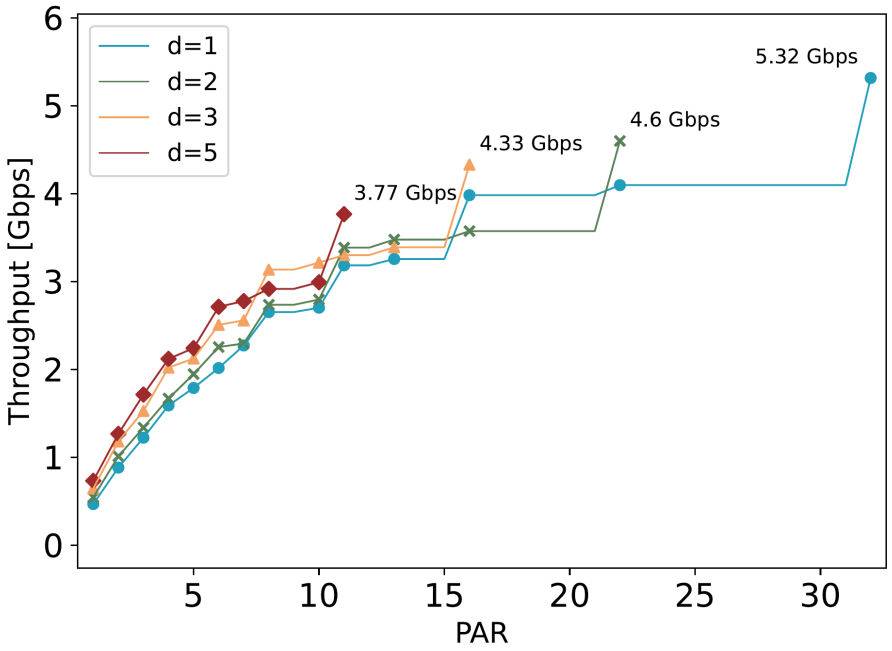


Figure B.1: Throughput (TH) varying PAR and d for Wireless application

C FSM state encoding

Code (hex)	State
0x00	IDLE
0x01	INIT_LOAD
0x02	INIT_ROUND_SHIFT
0x03	INIT_ROUND_SHIFT_LAST
0x04	INIT_ROUND_DIFFUSE
0x05	INIT_ROUND_DIFFUSE_LAST
0x06	ABSORB_AD_DATA
0x07	PROCESS_AD_SHIFT
0x08	PROCESS_AD_SHIFT_LAST
0x09	PROCESS_AD_DIFFUSE
0x0A	PROCESS_AD_DIFFUSE_LAST
0x0B	ABSORB_MSG_DATA
0x0C	PROCESS_MSG_SHIFT
0x0D	PROCESS_MSG_SHIFT_LAST
0x0E	PROCESS_MSG_DIFFUSE
0x0F	PROCESS_MSG_DIFFUSE_LAST
0x10	FINALIZATION_SHIFT
0x11	FINALIZATION_SHIFT_LAST
0x12	FINALIZATION_DIFFUSE
0x13	SQUEEZE_TAG
0x14	DONE

Table C.1: FSM state encoding for Ascon core.

Bibliography

[1] National Institute of Standards and Technology. *Lightweight Cryptography*. Accessed: 2025-08-22. 2023. URL: `https://csrc.nist.gov/projects/lightweight-cryptography`.

[2] Mattia Mirigaldi et al. "The Quest for Efficient ASCON Implementations: A Comprehensive Review of Implementation Strategies and Challenges". In: *Chips* 4.2 (2025), p. 15.

[3] Hannes Gross et al. "Ascon hardware implementations and side-channel evaluation". In: *Microprocessors and Microsystems* 52 (2017), pp. 470–479.

[4] Christoph Dobraunig et al. "Ascon v1. 2: Lightweight authenticated encryption and hashing". In: *Journal of Cryptology* 34.3 (2021), p. 33.

[5] H Dobbertin, A Bosselaers, and B Preneel. "International Workshop on Fast Software Encryption". In: (1996).

[6] Guido Bertoni et al. "Duplexing the sponge: single-pass authenticated encryption and other applications". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2011, pp. 320–337.

[7] Guido Bertoni et al. "Keccak specifications". In: *Submission to nist (round 2)* 3.30 (2009), pp. 320–337.

[8] Secure Hash Standard. "Secure hash standard". In: *FIPS PUB* (1995), pp. 180–1.

[9] François-Xavier Standaert. "Introduction to side-channel attacks". In: *Secure integrated circuits and systems*. Springer, 2009, pp. 27–42.

[10] Raphael Spreitzer et al. "Systematic classification of side-channel attacks: A case study for mobile devices". In: *IEEE communications surveys & tutorials* 20.1 (2017), pp. 465–488.

[11] Mark Randolph and William Diehl. "Power side-channel attack analysis: A review of 20 years of study for the layman". In: *Cryptography* 4.2 (2020), p. 15.

[12] Colin O'Flynn. "Fault injection using crowbars on embedded systems". In: *Cryptology ePrint Archive* (2016).

[13] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. "Electromagnetic fault injection: the curse of flip-flops". In: *Journal of Cryptographic Engineering* 7.3 (2017), pp. 183–197.

[14] Jasper GJ Van Woudenberg, Marc F Witteman, and Federico Menarini. "Practical optical fault injection on secure microcontrollers". In: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2011, pp. 91–99.

[15] Michael Hutter and Jörn-Marc Schmidt. "The temperature side channel and heating fault attacks". In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2013, pp. 219–235.

[16] Data Encryption Standard et al. "Data encryption standard". In: *Federal Information Processing Standards Publication* 112.3 (1999).

[17] Morris J Dworkin et al. "Advanced encryption standard (AES)". In: (2001).

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Annual international cryptology conference*. Springer. 1999, pp. 388–397.

[19] Carolyn Whitnall, Elisabeth Oswald, and François-Xavier Standaert. "The myth of generic DPA... and the magic of learning". In: *Cryptographers' Track at the RSA Conference*. Springer. 2014, pp. 183–205.

[20] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation power analysis with a leakage model". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2004, pp. 16–29.

[21] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. "Template attacks". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2002, pp. 13–28.

[22] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. "A testing methodology for side-channel resistance validation". In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.

[23] Srinidhi Hari Prasad et al. "Efficient low-latency masking of ascon without fresh randomness". In: *Cryptology ePrint Archive* (2023).

[24] Yuval Ishai, Amit Sahai, and David Wagner. "Private circuits: Securing hardware against probing attacks". In: *Annual International Cryptology Conference*. Springer. 2003, pp. 463–481.

[25] Stefan Mangard and Kai Schramm. "Pinpointing the side-channel leakage of masked AES hardware implementations". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2006, pp. 76–90.

[26] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold implementations against side-channel attacks and glitches". In: *International conference on information and communications security*. Springer. 2006, pp. 529–545.

[27] Begül Bilgin et al. "Threshold implementations of small S-boxes". In: *Cryptography and communications* 7.1 (2015), pp. 3–33.

[28] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order". In: *Cryptology ePrint Archive* (2016).

[29] Joan Daemen. “Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing”. In: *International conference on cryptographic hardware and embedded systems*. Springer. 2017, pp. 137–153.

[30] Gaëtan Cassiers and Olivier Bronchain. “Scalib: A side-channel analysis library”. In: *Journal of Open Source Software* 8.86 (2023), p. 5196.

[31] Hannes Groß et al. “Suit up!–made-to-measure hardware implementations of Ascon”. In: *2015 Euromicro Conference on Digital System Design*. IEEE. 2015, pp. 645–652.

[32] Hannes Groß, Rinat Iusupov, and Roderick Bloem. “Generic low-latency masking in hardware”. In: *IACR transactions on cryptographic hardware and embedded systems* (2018), pp. 1–21.

[33] Rishub Nagpal et al. “Riding the waves towards generic single-cycle masking in hardware”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), pp. 693–717.

[34] Luan Cardoso dos Santos, Johann Großschädl, and Alex Biryukov. “FELICS-AEAD: benchmarking of lightweight authenticated encryption algorithms”. In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2019, pp. 216–233.

Acknowledgements

Al Professore Guido Masera, relatore di questa tesi, per la disponibilità, la guida costante e i preziosi consigli che hanno reso possibile la realizzazione di questo lavoro.
A Mattia Mirigaldi per il supporto tecnico e scientifico, e per gli stimoli ricevuti durante tutto il percorso. Desidero in particolare esprimere la mia sincera gratitudine al relatore, al Professore Maurizio Martina e a Mattia per la pazienza, i consigli e l'aiuto che mi hanno guidato e sostenuto anche nella scelta del mio percorso futuro.