POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design and integration of a RISC-V based accelerator for ASCON

Supervisors

Prof. Guido MASERA

Prof. Maurizio MARTINA

Ing. Alessandra DOLMETA

Candidate

Federica BADER

December 2024

Abstract

Ensuring privacy and data protection has become a paramount concern nowadays, leading to the implementation of cryptographic algorithms that have become more advanced and robust as time goes by. A complex algorithm needs a powerful device to be deployed, but the Internet of Things (IoT) is made up of all kinds of machines, many of which are constrained by computational power or memory or battery life. This is the field of lightweight cryptography, algorithms made specifically to balance security with efficiency, ensuring that all systems can safeguard data without hindering performance. The newly appointed standard is the Ascon family, which is a group of algorithms that share the same core function to perform a variety of functions, spanning from authenticated encryption with associated data (AEAD) to hashing and extendable output functions (XOF). This work is focused on improving the performance of these algorithms, through the design of a customized accelerator. This is done in a few steps; the first optimization is performed through an instruction extension of a generic RISC-V processor by means of ASIP Designer. This tool provides exceptional profiling capabilities and a fast implementation method, optimal to iteratively finding out the bottlenecks of the algorithm and solving them, while trying different designs. The main bottleneck turns out to be the core function of the algorithms, meaning that one accelerator can speed up all the family. In order to solve it, many designs were proposed, ranging from the translation in hardware of the function, to the application of optimization techniques, like unrolling. In the end, the final processor can compute the algorithm up to ten times faster than the baseline, a result that is five times faster than other implementations in literature. Meanwhile, the best compromise between occupied area and speed is characterized by a speedup of 8.6× from the reference implementation, at the expense of a marginal area increment of the processor of +20%. The acceleration performed until now has resulted in an application-specific processor, which has the fastest computation, but at the expense of having changed the nature of the processor. In order to maintain the original structure, the following step is to design a coprocessor. This is still integrated into the pipeline of the core, but it communicates through a specific interface. In the end, the obtained speedup reaches up to 5.8× faster than the baseline in the hashing algorithms, and 3.6× of the AEAD.

Keywords: Lightweight cryptography, Ascon, RISC-V, Hardware accelerator, ASIC, FPGA.

Acknowledgements

Il successo non è mai il
risultato di uno sforzo solitario:
è il prodotto dell'amore della famiglia,
della forza degli amici
e della saggezza dei maestri

Table of Contents

List of Tables

List of Figures

Acronyms

NIST

National Institute of Standards and Technology

AEAD

Authenticated Encryption with Associated Data

XOF

Extendable-Output Function

ISA

Instruction Set Architecture

ISE

Instruction Set Extension

ASIP

Application Specific Instruction Set Processor

ASIC

Application Specific Integrated Circuit

CV-X-IF

Core-V eXtension interface

Chapter 1

Introduction

In the modern world, information has become the most valuable asset. Society is based on interconnected devices that constantly exchange private information, from sensitive data to seemingly unimportant messages. For this reason, centuries ago, cryptographic protocols were developed to hide data in plain sight, yielding an unreadable message to those who try to intercept the transmission. Of course, along with cryptographic protocols, deciphering strategies have also come into existence, creating the need for more complex protocols that make decryption without a key an almost unsolvable challenge. A complex algorithm needs a powerful device to be deployed, but the Internet of Things (IoT) is made up of all kinds of machines, many of which are constrained in a handful of aspects ranging from computational power to memory and battery life. This is the field of **lightweight cryptography**, algorithms made to balance security with efficiency, ensuring that all systems can safeguard data without hindering performance.

The NIST (National Institute of Standards and Technology) has organized CAESAR, the Competition for Authenticated Encryption: Security, Applicability, and Robustness in order to find a new cryptographic standard that could not only provide security, but also flexibility and performance in terms of speed, size, and energy use. The winning algorithm was announced in 2023: **ASCON** a **family of ciphers** that has now become the standard for lightweight cryptography.

These algorithms are perfect for an *ASIP* (Application-Specific Instruction Processors) or a **coprocessor** implementation due to their simple and efficient nature.

The small state and simple bitwise operations map efficiently to specialized instruction sets, providing high throughput with low latency. These optimizations will also allow increasing performance while reducing energy consumption and maintaining flexibility, making it a perfect fit for resource-constrained environments like IoT devices and embedded systems.

1.1 Thesis objectives and organization

The main objective of this thesis is to implement an accelerator that targets the most computationally heavy functions of ASCON. This will be done by first finding the bottlenecks of the algorithm, solving them iteratively, and providing a broad range of solutions until the best result is achieved. Then, the most suitable one will be implemented and integrated into a microcontroller to test its functionality.

The thesis will be organized as follows:

- **chapter 1** is a short introduction about the motivations and the purpose of this thesis;
- **chapter 2** presents the background necessary to completely understand what will be done in the following chapters. Section 2.1 contains an explanation of cryptography in general and then specifically of the algorithms that will be the protagonists of this thesis. Section 2.2 is about Ascon and operations that characterize this family of ciphers. The following two sections, 2.3 and 2.4, are an overview of the tools that will allow the design of the accelerator;
- **chapter 3** describes the design flow that brings to the development of a collection of suitable accelerators, and ultimately to the choice of the best one in terms of speed-up and area usage. Section 3.1 is dedicated to the choice of one of the different versions of the same algorithms, showing why it is the more advantageous of the alternatives. Section 3.2 illustrates the steps made in order to find out the criticalities of the algorithm and iteratively solve them. Section 3.3 shows the result of the instruction set extension performed, discussing all the benefits and the drawbacks of this design;
- **chapter 4** is where the coprocessor is effectively implemented. In sections 4.1 and 4.2 a brief introduction of the microcontroller and its interface is given, while in section 4.3 the design of the coprocessor is thoroughly explained. At last, in section 4.4 the design is synthesized, both in FPGA and ASIC;

- **chapter 5** reports the results of the previous ISE in section 5.1, simulation in section 5.2 and synthesis in section 5.3. Then, they are compared to the data found in the literature, discussing the merits and the limitations of the implemented design. In section 5.4 some future work proposals are laid out;
- **chapter 6** is the conclusion of the thesis, it formulates the reflections on this work while summarizing all the outcomes achieved.

Chapter 2

Theoretical background

To better understand the work that will be explained later, it is useful to have a bit of theoretical background. Starting from the basis of cryptography and of the algorithms that will be the protagonists of this thesis, across the tools and programs that allow the design of the accelerator, until the processor that will bring this project into the real world.

2.1 Cryptography

> *"One must acknowledge that with cryptography no amount of violence will solve a math problem" — Jacob Appelbaum*

Using a **key**, a **cryptographic algorithm** can **encrypt** a **plaintext** from the sender into a **ciphertext**, creating an unrecognizable file. Through the use of the same or a different key, the receiver can **decrypt** the message and recover the original. Only these two people should be able to view the content of the transmission because a third party would not be able to recover the key if the algorithm is reliable enough. The process is depicted in Figure 2.1 for clarity.

Nowadays, in several areas are emerging technologies implemented in constrained devices, spanning from sensor networks to healthcare and the Internet of Things, where the data are highly sensitive. The concept of **lightweight cryptography** comes from the need to protect the messages coming from and to these types of equipment, where the performance of standard cryptographic algorithms is unacceptable. The factors that make up a good lightweight cipher are high security,

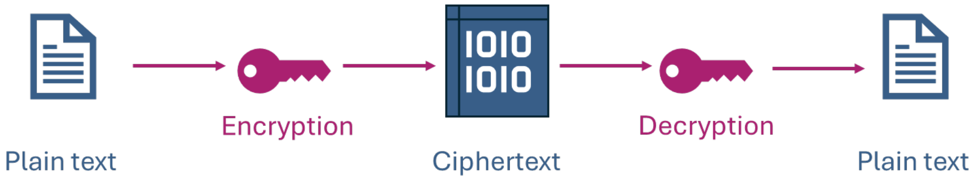


Figure 2.1: Representation of encryption and decryption

high speed, low size, low energy use, and low memory consumption. Since it is impossible to have all of these, lightweight cryptography is made of compromises. For example, they have a minimal size and they provide good security, but it makes them not safe enough to withstand attacks from quantum computers. However, this particular scenario might not be a problem, because post-quantum encryption is important for long-term secrets that would not be shared in this kind of platform.

The family of algorithms that has been chosen as a standard for lightweight cryptography is ASCON. This group is composed of methodologies for:

- AEAD, authenticated encryption with associated data. An encryption scheme that assures data confidentiality and authenticity, while including associated information.
- HASH, a transformation that converts a message of arbitrary length into one with fixed size, used for message integrity.
- XOF, extendable-output function. It is a type of hash function that allows the output to be of an adjustable size.

2.1.1 AEAD

Figure 2.2 represents the main scheme of an AEAD algorithm.

Like before, the **plain text** and the **key** are present, while the nonce, the additional data, and the tag are specific to this model. The **nonce** (number used once) is a number used to modify a ciphertext. Its goal is to avoid detection when related messages with a similar ciphertext are sent. For this reason, it must also be unique at each communication and it usually consists of a random number or a counter value that is shared between the two recipients of the message. Of course, when the number is generated randomly, it is needed also in the decryption step to recover the original message. The **tag** is what guarantees that the message received comes

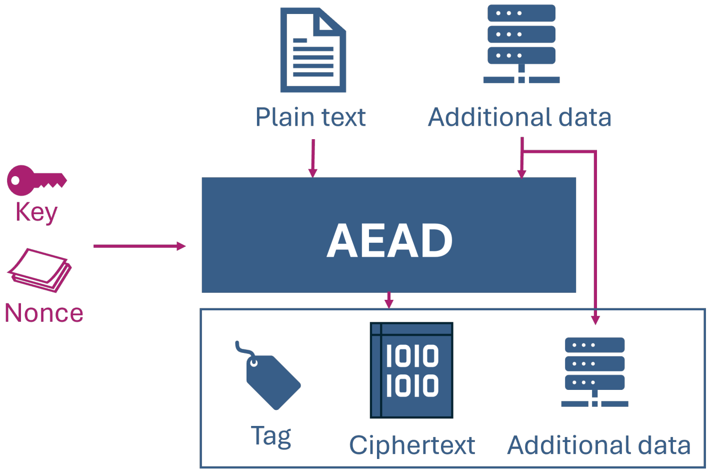


Figure 2.2: Representation of an AEAD algorithm

from the expected sender. In fact, the tag can be generated only by knowing the secret key. These are the elements that come into play in an AE (authenticated encryption), an AEAD sends also the **associated data**. This kind of information is authenticated but not encrypted, it is useful because it assures that third parties cannot repeat a message they have already intercepted in a different circumstance, by giving the context of the communication. As for the nonce case, this value takes part both in the encryption and the decryption of the ciphertext.

As said before, an AEAD scheme ensures **confidentiality** and **authenticity**. The former means that the message sent cannot be understood without the secret key, while the latter denotes that it is not forgeable. This is possible because an adversary could be able to generate a ciphertext, but not the tag [1].

2.1.2 HASH

The behavior of a hashing algorithm is represented in Figure 2.3.

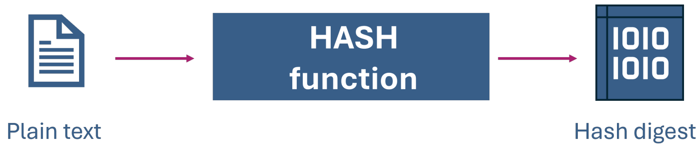


Figure 2.3: Representation of a hashing algorithm

A hash function is a mathematical transformation that takes a message of an arbitrary length and transforms it into another compressed value of fixed length, called “hash digest”. The obtained value is defined by the input, which turns into a unique digest whose starting contents cannot be recovered. Hashing can be used for key generation or number randomization [2].

2.1.3 XOF

The concept of extendable output function is related to the one of hashing. The only difference between the two is that an XOF algorithm can output a digest of arbitrary length, while the one of the hash is fixed. It can be useful in situations like deriving multiple keys instead of one at a time.

2.1.4 Sponge and Duplex Sponge Constructions

The Sponge and Duplex Sponge constructions are the basis for many cryptographic algorithms. Its working principle is represented in Figure 2.4.

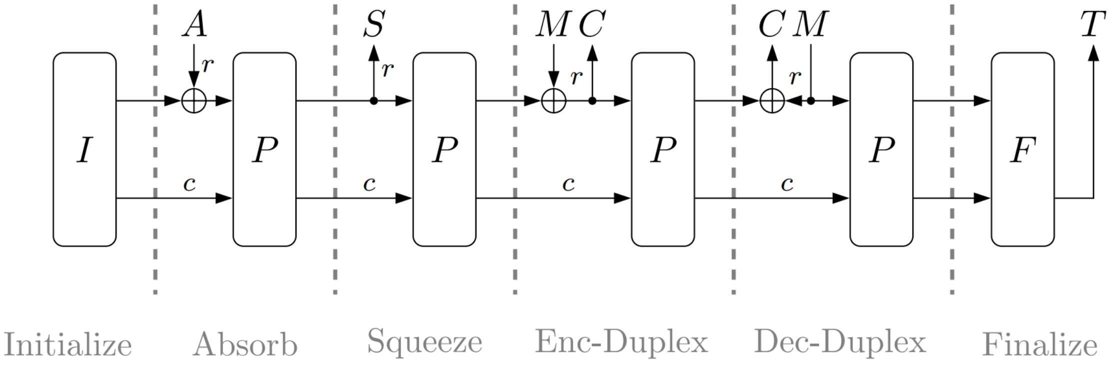


Figure 2.4: Sponge operations

The principle is based on a **permutation** P operating on a **state vector** of fixed length b. This, in turn, is made by the rate r, also known as the data block size, and the capacity c, which is the difference between b and r. Since the variable-length input will be processed by blocks of size r, padding ensures that the input data length is a multiple of the block size.

The first step of the sponge operation is the **initialization**. This is where the starting configuration of the state vector is determined, depending on the specific operation. If the algorithm involves the use of a key, then it is included in this step, like in the AEAD schemes. On the other hand, for hash functions, the initial state is usually a constant.

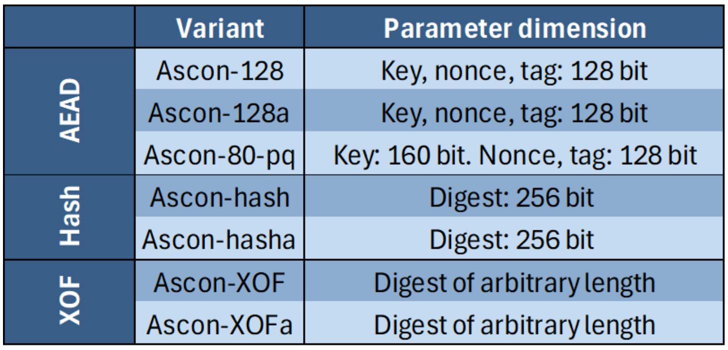
The second stage is the one of the **absorbption**. Here, the r-bit input message blocks are XORed into the first r bits of the state, while the permutation P is applied. When all the blocks are processed, the construction goes to the **squeezing** operation, where the state is returned as output blocks. The number of operations is determined by the number of bits of the outputs.

The following steps are the **duplex operations**. They differ from the sponge construction because they allow for alternating input-output phases in the same operation [3]. They can be used either for encoding or for decoding.

Just for some algorithms, like AEAD or MAC, there is also the **finalization** step, that extracts the tag from the last state of the sponge.

2.2 ASCON

Ascon is a cipher suite that provides both AEAD and hashing functionalities, the whole family with their parameters is reported in tables 2.1 and 2.2. In addition to these, there is the rate r, also known as the block size, which is 64 bits for almost all versions, with the exception for ascon128a, where it is 128.



	Variant	Parameter dimension
AEAD	Ascon-128	Key, nonce, tag: 128 bit
	Ascon-128a	Key, nonce, tag: 128 bit
	Ascon-80-pq	Key: 160 bit. Nonce, tag: 128 bit
Hash	Ascon-hash	Digest: 256 bit
	Ascon-hasha	Digest: 256 bit
XOF	Ascon-XOF	Digest of arbitrary length
	Ascon-XOFa	Digest of arbitrary length

Table 2.1: Ascon family with their parameters

The authenticated encryptions with associated data schemes are Ascon-128, Ascon-128a, and Ascon-80-pq. This last one is a new version that has increased resilience to quantum key-search [4]. The hash functions are Ascon-hash and Ascon-hasha, while the XOF are Ascon-XOF and Ascon-XOFa. Each of them provides substantially different functionality, but they all **have in common the same primitive**, which

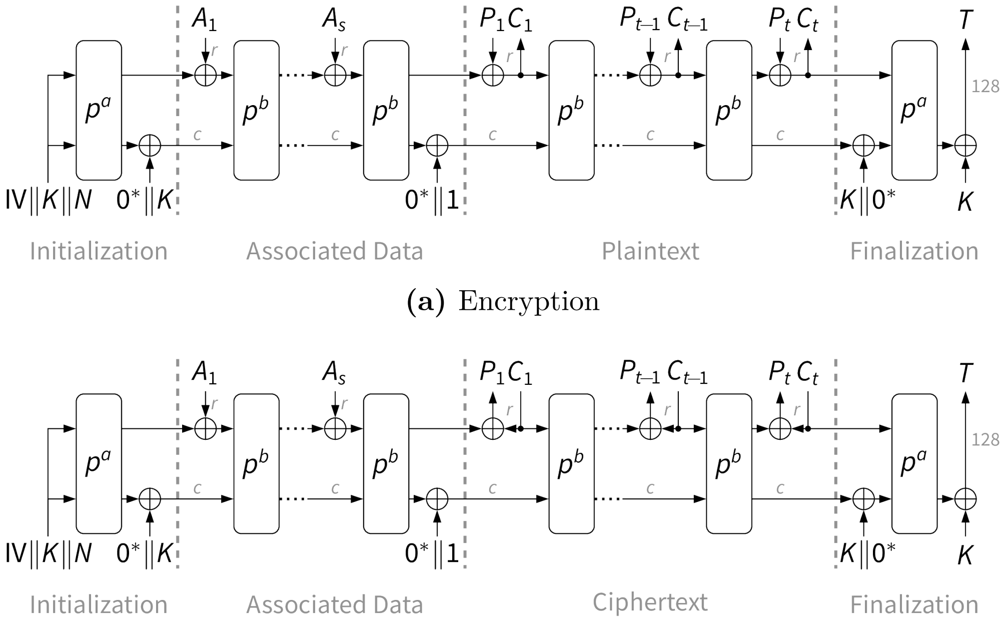
	Variant	a	b
AEAD	Ascon-128	12	6
	Ascon-128a	12	8
Hash	Ascon-hash	12	12
	Ascon-hasha	12	8
XOF	Ascon-XOF	12	12
	Ascon-XOFa	12	8

Table 2.2: Parameters that define the number of permutations performed

is a low-level algorithm, used to compose higher-level ones.

2.2.1 AEAD mode

This mode uses a duplex-sponge-based protocol that operates on a **320-bit state vector** that is, in turn, split into one outer part S_r and an inner part S_c. When the permutations are performed, this vector is represented by five registers of 64-bit each. In Figures 2.5 are reported the steps for encryption and decryption.



(a) Encryption

(b) Decryption

Figure 2.5: Encryption and decryption for the AEAD schemes

From figures 2.5, it is clear that the core parts of the algorithms are the p^a and the p^b blocks. These are the **permutations** and the numbers a and b define how many times the **round** transformation is applied. For Ascon-128, a is equal to twelve and b is six, while for the Ascon-128a algorithm, a stays the same and b is eight, as seen in table 2.2. A higher number means a stronger permutation, in fact p^a is present in the initialization and finalization phases.

As it is possible to notice, the encryption and decryption are divided into four steps:

1. **initialization**: the state vector of 320 bits is composed by the initialization vector (IV) calculated as:

   $$IV_{k,r,a,b} \leftarrow k||r||a||b||0^{160-k}$$

   where k is the key size, r is the rate, a the initialization and finalization round number and b the intermediate round number. The last bits are assigned to the zeros padding. Then the key and the nonce are added as:

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

   Then, the state goes through a permutation and the key is XORed into the c bits of the capacity:

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

2. **associated data**: the associated data A is processed in blocks of r bits. A is padded with a one, followed by as many zeros as needed to reach a multiple of r:

   $$A \leftarrow A||1||0^{r-1-(|A|\,mod\,r)}$$

   Then each A block is XORed with S_r, followed by a b permutation:

   $$S \leftarrow p^b((S_r \oplus A)||S_c)$$

   In the last step, a separation constant is XORed to S;

3. **plaintext/ciphertext**: the ciphertext/plaintext is processed in blocks of r bits. P/C goes through the same padding process as before, where a single one and as many zeros as needed are added. At each iteration, the plaintext

or the ciphertext is XORed to S_r and goes through a b permutation.
The last step for encryption consists of truncating the length of the last ciphertext block to make the ciphertext C length the same as the original plaintext P

$$\tilde{C}_t \leftarrow \lfloor C_t \rfloor_{|P|\, mod\, r}$$

For decryption the last procedure is

$$\begin{aligned}\widetilde{P}_t &\leftarrow \lfloor S_r \rfloor_\ell \oplus \widetilde{C}_t \\ S &\leftarrow \Big(S_r \oplus (\widetilde{P}_t \,\|\, 1\|0^{r-1-\ell})\Big)\|S_c\end{aligned}$$

4. **finalization**: The tag is extracted as:

$$\begin{aligned}S &\leftarrow p^a\Big(S \oplus (0^r\|K\|0^{c-k})\Big) \\ T &\leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}\end{aligned}$$

For a more in-depth explanation, refer to [4].

2.2.2 HASH and XOF modes

The operation of the hashing and XOF modes are reported in Figure 2.6.

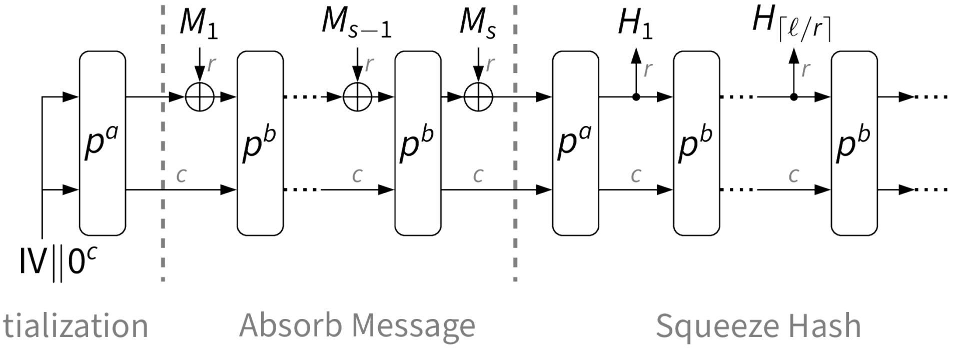


Figure 2.6: Hashing and XOF schemes

Again, the computation is split into steps:

1. **initialization**: similarly to the AEAD case, it uses the constant IV to initialize the state vector. It is calculated as:

$$IV_{h,r,a} \leftarrow 0^8||r||a||0^8||hS \leftarrow p^a(IV_{h,r,a}||0^{256})$$

This number is constant and it has a precomputed value assigned at the start of the algorithm;

2. **absorb message**: the message M is processed in blocks of r bits, padded with the same procedure used for the AEAD scheme. After this, each of the s message blocks of r bits goes through the function:

$$S \leftarrow p^a((S_r \oplus M_i)||S_c)$$

3. **squeeze hash**: the hash output is extracted from the state in blocks of r bits until the desired length is reached. The last block is truncated to

$$\tilde{H}_t \leftarrow \lfloor H_t \rfloor_{l\,mod\,r}$$

Where l is the output size.

In this case, a is always twelve, while b is twelve in the standard version (Ascon-Hash and Ascon-XOF) and eight in the "-a" variant (Ascon-Hasha and Ascon-XOFa), as shown in table 2.2.

2.2.3 Round transformation

As seen before, the **round transformation is the core of Ascon**. It is a 320-bit permutation, designed for high security and robustness with a very low area, thanks to the use of simple bitwise boolean operation.

The 320-bit state vector is divided into five registers, each of 64 bits, called x_0, x_1, x_2, x_3 and x_4. These registers go through a series of operations:

1. **addition of a round constant**: a constant of one byte is XORed to x_2:

$$x_2 \leftarrow x_2 \oplus c_r$$

2. **nonlinear substitution layer**: a 5-bit S-box is applied 64 times in parallel in a bit-sliced pattern (vertically, across words). As seen in Figure 2.7;
3. **linear diffusion layer**: XORs different rotated copies of each word horizontally, as depicted in Figure 2.8.

The constant added in the first step depends on which iteration is being performed inside the permutation. All the possible values are reported in Table 2.3. If the permutation consists of twelve rounds, then the first iteration will be used as a

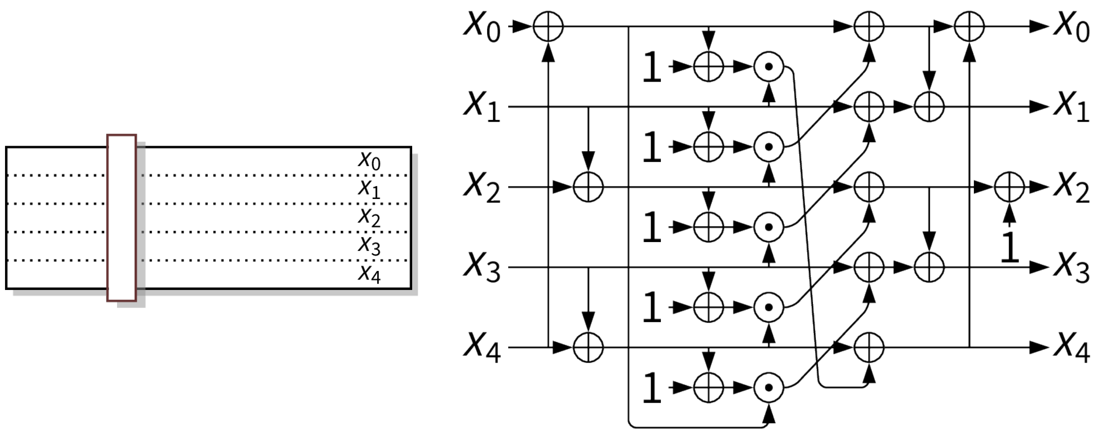
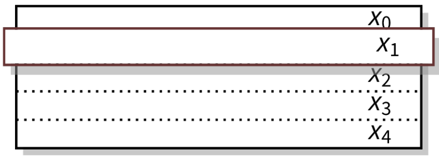


Figure 2.7: Second step of Round, nonlinear substitution layer



$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)\\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

Figure 2.8: Third step of Round, linear diffusion layer

constant 0xf0, then 0xe1, and so forth. If, instead, the permutation is composed of eight rounds, the first constant will be the one with index 2 (0xd2), followed by 0xb4 until the end. Similarly, the six rounds will start with the index 6.

Iteration	Const	Iteration	Const
0	0xf0	6	0x96
1	0xe1	7	0x97
2	0xd2	8	0x78
3	0xc3	9	0x69
4	0xb4	10	0x5a
5	0xa5	11	0x4b

Table 2.3: Table of the round constants

2.3 RISC-V

RISC-V is an open-source instruction-set architecture (ISA) that started as an educational and research instrument but is now widely used in all kinds of applications. RISC-V processors can have 32 or 64-bit parallelism (a 128-bit variant is under development), and the central register file is usually composed of 32 fields, with two read ports and one write port. The base ISA divides the instructions into four formats, depending on the register usage:

- **R-Type**: two source registers and the destination register
- **I-Type**: one of the source registers and the destination register
- **S-Type**: only the two source registers
- **U-Type**: only the destination register

A further classification can be made depending on the immediate used by the instruction, for more detail refer to [5].

The basic integer ISA can be expanded with many **extensions** that provide more functionality to the base processor, ranging from typical multiplication and floating point operations to more unusual operations like vector data and SIMD processing. Further customization of the ISA can be done with specialized instructions thanks to the open-source nature of the project. New custom instructions can be coded to carry out specific tasks or calculations, resulting in enhanced performance for certain applications. This procedure is called **Instruction Set Extension** (ISE) and allows the creation of unique processors with specialized ISA.
For the purpose of this thesis, a 32-bit version of the processor will be used, and the R-Type will be preferred for what concerns new instructions.

2.3.1 ASIP Designer

If an optimized processor is targeted to a specific field or algorithm, it is called an **application-specific instruction-set processors** (ASIP). Synopsys' ASIP Designer is a proprietary software that has been developed to ease the development of ASIP processors. The tool comes with built-in example models, including many RISC-V variants. As a starting point for this thesis, the **trv32p5x** has been chosen: this RISC-V-based processor comes with 32-bit parallelism, 5 pipeline stages, and hardware do-loop support. ASIP Designer development flow consists of a compiler-in-the-loop approach, where the architecture of the model can be seamlessly modified, compiled, and simulated both in instruction-accurate and cycle-accurate mode. The built-in debugger and disassembler allow for microcode stepping, which makes finding bugs and errors much simpler. The tool uses two main languages to describe the processor:

- **nML**: this class of files describes the memory resources, the pipeline stages, and the instruction set architecture of the model. The ISA is described using a hierarchical approach, grouping instructions associated with a specific task into a single bundle. Every bundle is then packed with the other ones until the complete ISA is formed.

- **PDG**: these files contain the description of the functional units' behavior, like ALU operations. The actions performed by the functional units can be described in a C++ style, simplifying the process of creating new functions for the processor.

The key aspects are reported in Figure 2.9.

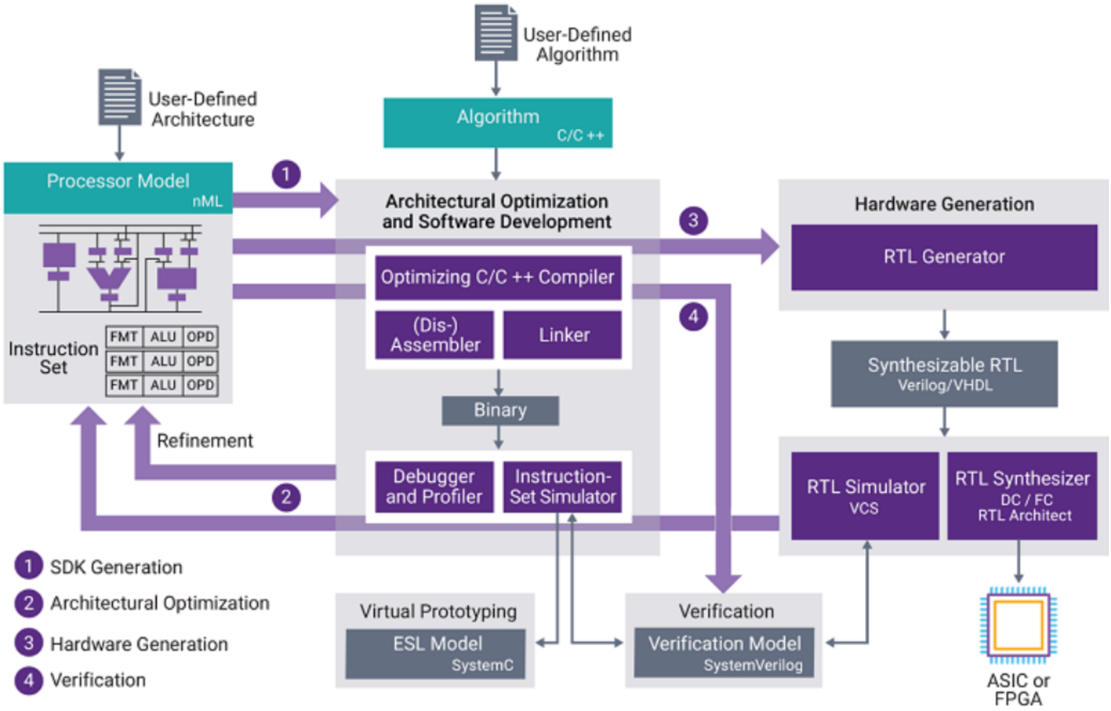


Figure 2.9: Asip Designer flow.
Image source: Synopsys' ASIP Designer™ webpage [6].

On top of that, ASIP Designer offers great profiling capabilities, including the measurements of cycle count, register accesses, instructions count, and tracing. Once the processor is completed, the tool can generate a synthesizable HDL description of the model either in Verilog or VHDL, which can be used for further development using other commercial tools.

2.4 Accelerators and coprocessors

When it comes to accelerators, two main methodologies can be adopted [7]:

- **Tightly coupled accelerators**, or TCA, are hardware components that are placed inside, or very close to, the pipeline of the processor. They require new instruction and are usually fixed-function units, highly specialized in a single task.
- **Loosely coupled accelerators**, or LCA, are modules that are usually separated from the CPU and can communicate with it in various ways, including memory mapping, I/O mapping, and interrupts.

Loosely coupled accelerators can usually be bigger and more sophisticated than their counterpart because they can work concurrently with the main processor, completing complex tasks like image processing in a few clock cycles. LCA usually don't need new instructions to operate because they are generally driven by functions already embedded into the processor, like memory operations. Tightly coupled ones, on the other hand, speed up simpler tasks, usually in a single clock cycle. They are also kept smaller to prevent long critical paths, that can lead to a loss of performance, and require ISA extension to work.

Coprocessors are TCA slightly more elaborate than usual ones, containing additional architectural components that allow for some limited autonomy relative to the primary instruction stream.

For the purpose of this thesis, a coprocessor will be developed, which will use the **CV-X-IF Interface**. This open-source interface can be used to implement standard RISC-V extensions (like the B, M, and F) as well as custom ones. The aim of this protocol is to enable the design of ASIPs with tightly coupled accelerators, without modifying the CPU of the processor.

Chapter 3

Instruction Set Extension

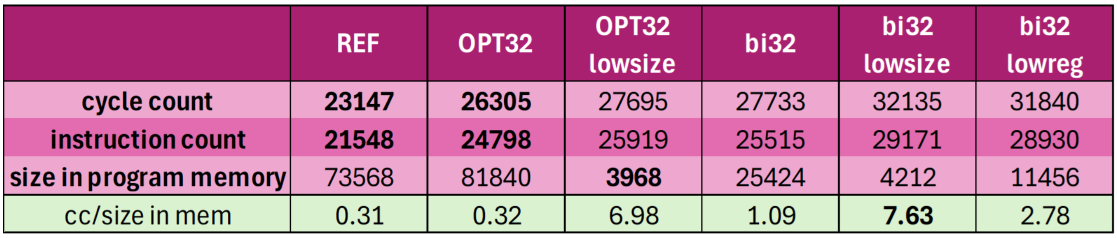
The first type of accelerator that will be implemented is through the instruction set extension of the RISC-V processor. This kind of tightly coupled acceleration allows ascertaining the full extent of the performance improvement obtained and, at the same time, provides a higher speed of analysis useful to compare different design options.

The first thing that has to be done is to choose the more appropriate implementation among the provided ones. After obtaining the starting model, the bottlenecks of the algorithm must be evaluated and solved iteratively until the desired goal is reached. In the end, the results are examined, also considering other factors that come into play when dealing with accelerators.

3.1 Choice of implementation

The official repository containing the C implementation of Ascon [8] contains many versions of it, each for a different purpose. The possible optimization criteria are the number of bits, speed, size in program memory, register usage, and presence of interleaving. The RISC-V that will be used is based on 32 bits, and both the size and the register usage are not relevant points of the final result. This is because the accelerator will be placed inside a processor that will handle many complex cryptographic algorithms, which will require larger space and more registers than Ascon would ever need. Because the other factors are not of significance, speed becomes the most prominent goal. Interleaving could hinder the performance of the system, so the final decision would be to use the 32-bit speed-optimized version, "opt32".

This whole process was verified with the profiling capabilities of Asip Designer, yielding the table 3.1.



	REF	OPT32	OPT32 lowsize	bi32	bi32 lowsize	bi32 lowreg
cycle count	**23147**	**26305**	27695	27733	32135	31840
instruction count	**21548**	**24798**	25919	25515	29171	28930
size in program memory	73568	81840	**3968**	25424	4212	11456
cc/size in mem	0.31	0.32	6.98	1.09	**7.63**	2.78

Table 3.1: Comparison of all C implementation provided

The column of the reference implementation is misleading, the cycle count seems lower than the one of opt32, suggesting that it is more efficient. In reality, the reference implementation implicitly performs unrolling and it does not deal with endianness.

Unrolling makes the computation faster, removing the overheads due to loops and function calls. There is no option to remove unrolling in this version, while in the others it was removed to have a more accurate comparison.

Endianness is a problem of many cryptographic algorithms: these schemes are made to deal with data in big-endian order, while RISC-V and many other processors store and read data in little-endian order. It is not a problem when the program is run on more complex machines with advanced operating systems that can automatically differentiate and handle this condition. On the other hand, when using smaller devices, there should be functions that reorder data every time it is written or read. The reference implementation is missing this whole ordeal, making the code faster and smaller, but not suitable to be put into a resource-constrained environment.

3.2 Acceleration of the most critical function

After choosing the base model, the next step is to analyze the flow of the algorithm, finding the bottlenecks that affect the performance of the system. This can be easily done through the *function profiling* of Asip Designer, which yields (the full report can be seen in appendix, B.2):

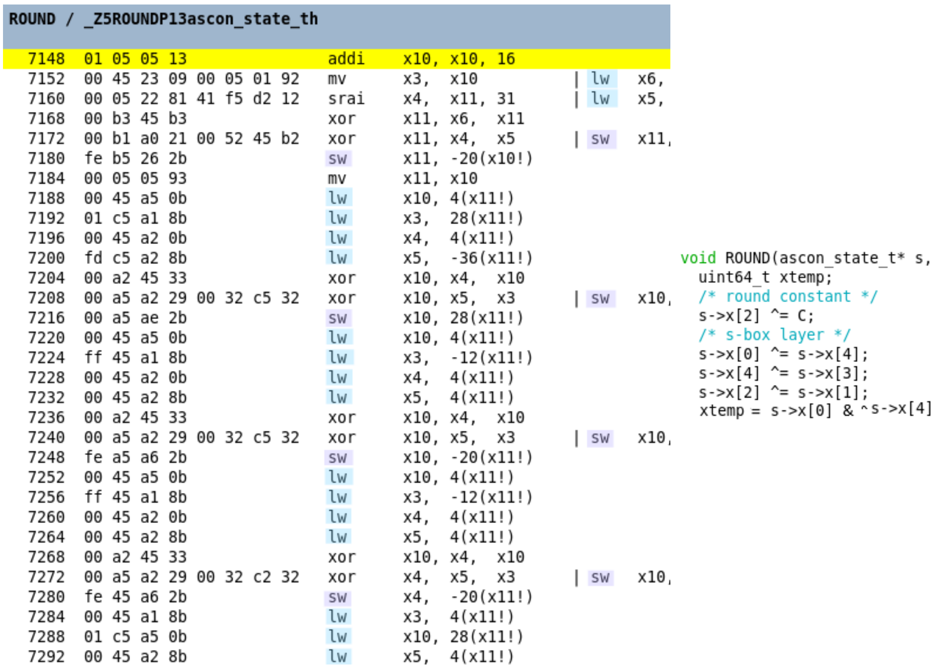
```
 Calls  Cycles tot   Cycles tot  Cycles tot   Cycles tot   Function
             (func)      (%func) (func+desc) (%func+desc)
------ ----------- ------------ ----------- ------------ -------------
   108       22464       87.28%       22464       87.28%  ROUND
    14         854        3.32%       23318       90.60%  PROUNDS
    24         720        2.80%         720        2.80%  memcpy
     2         304        1.18%        8218       31.93%  ascon_adata
   ...         ...          ...         ...          ...       ...
```

It is clear that **the bottleneck is the round function**. The second most computationally expensive function, PROUNDS, is the permutation that iteratively calls the round function. The effective cost of the two is shown by the column of total cycles of the function and its descendants. Furthermore, permutations are also called many times during one execution, which means that accelerating its performance would grant many benefits.

There are two main reasons why this function is so costly: the first is that, as explained in chapter 2, round performs many combinatorial operations that, despite their basic nature, take a long time to be performed in software. This is especially true for 32-bit processors, which have to perform twice the operations to compute the 64-bit data. The second reason can be found by performing an analysis of the storage accesses, which gives (the full report can be seen in appendix B.1):

```
Storage   Read-count  Read-count   Read-count  Function name
             (total)  (function) (% of total)
------- ----------- ------------ -----------  -------------------
DMb           29464        28080       95.30%  ROUND
                             256        0.87%  ascon_adata
                              44        0.15%  ascon_aead_decrypt
...             ...          ...          ...     ...
```

Note that the behavior is the same also for writes. A significant part of all operations made to the data memory are performed by the round function that, every time it is called, loads and stores the state vector. This pattern can be observed through the microcode that performs the C algorithm, which is reported in Figure 3.1.



```
ROUND / _Z5ROUNDP13ascon_state_th

 7148  01 05 05 13                addi    x10, x10, 16
 7152  00 45 23 09 00 05 01 92    mv      x3,  x10          | lw   x6,
 7160  00 05 22 81 41 f5 d2 12    srai    x4,  x11, 31      | lw   x5,
 7168  00 b3 45 b3                xor     x11, x6,  x11
 7172  00 b1 a0 21 00 52 45 b2    xor     x11, x4,  x5      | sw   x11,
 7180  fe b5 26 2b                sw      x11, -20(x10!)
 7184  00 05 05 93                mv      x11, x10
 7188  00 45 a5 0b                lw      x10, 4(x11!)
 7192  01 c5 a1 8b                lw      x3,  28(x11!)
 7196  00 45 a2 0b                lw      x4,  4(x11!)
 7200  fd c5 a2 8b                lw      x5,  -36(x11!)             void ROUND(ascon_state_t* s, uint8_t C) {
 7204  00 a2 45 33                xor     x10, x4,  x10                uint64_t xtemp;
 7208  00 a5 a2 29 00 32 c5 32    xor     x10, x5,  x3      | sw   x10,   /* round constant */
 7216  00 a5 ae 2b                sw      x10, 28(x11!)                s->x[2] ^= C;
 7220  00 45 a5 0b                lw      x10, 4(x11!)                 /* s-box layer */
 7224  ff 45 a1 8b                lw      x3,  -12(x11!)               s->x[0] ^= s->x[4];
 7228  00 45 a2 0b                lw      x4,  4(x11!)                 s->x[4] ^= s->x[3];
 7232  00 45 a2 8b                lw      x5,  4(x11!)                 s->x[2] ^= s->x[1];
 7236  00 a2 45 33                xor     x10, x4,  x10                xtemp = s->x[0] & ~s->x[4]
 7240  00 a5 a2 29 00 32 c5 32    xor     x10, x5,  x3      | sw   x10,
 7248  fe a5 a6 2b                sw      x10, -20(x11!)
 7252  00 45 a5 0b                lw      x10, 4(x11!)
 7256  ff 45 a1 8b                lw      x3,  -12(x11!)
 7260  00 45 a2 0b                lw      x4,  4(x11!)
 7264  00 45 a2 8b                lw      x5,  4(x11!)
 7268  00 a2 45 33                xor     x10, x4,  x10
 7272  00 a5 a2 29 00 32 c2 32    xor     x4,  x5,  x3      | sw   x10,
 7280  fe 45 a6 2b                sw      x4,  -20(x11!)
 7284  00 45 a1 8b                lw      x3,  4(x11!)
 7288  01 c5 a5 0b                lw      x10, 28(x11!)
 7292  00 45 a2 8b                lw      x5,  4(x11!)
```

Figure 3.1: Microcode that performs **a small part** of the round function

The microcode (on the left) is the translation of the small section of C code shown on the right. It is clear that many cycles are wasted to load and store words (`lw` and `sw` instructions) because these operations are performed each time a variable changes its value. Inside a single round function, there are 65 `lw` and 36 `sw`. To understand the magnitude of this expense, these numbers that have to be multiplied by how many times inside a permutation a round is called and then again by how many times a permutation is performed.

3.2.1 Hardware acceleration

The simple boolean operations, that were so expensive to do in software, can be easily realized in hardware. By doing so, the computations are done in parallel and there is no need to store intermediate results.
The round function is composed of three steps: addition of constants, Sbox, and linear diffusion layer. Each operation contributes differently to the function and

its importance must be evaluated, in case it is beneficial to accelerate only one component in order to occupy less area. This hardware also needs a dedicated register, called "reg_S", with a peculiar structure: each 64-bit segment has its own input and output port. This is necessary because the hardware needs all 320 bits from the state vector at the same time to perform the computations in a single clock cycle. The outline of the proposed architecture is represented in figure 3.2.

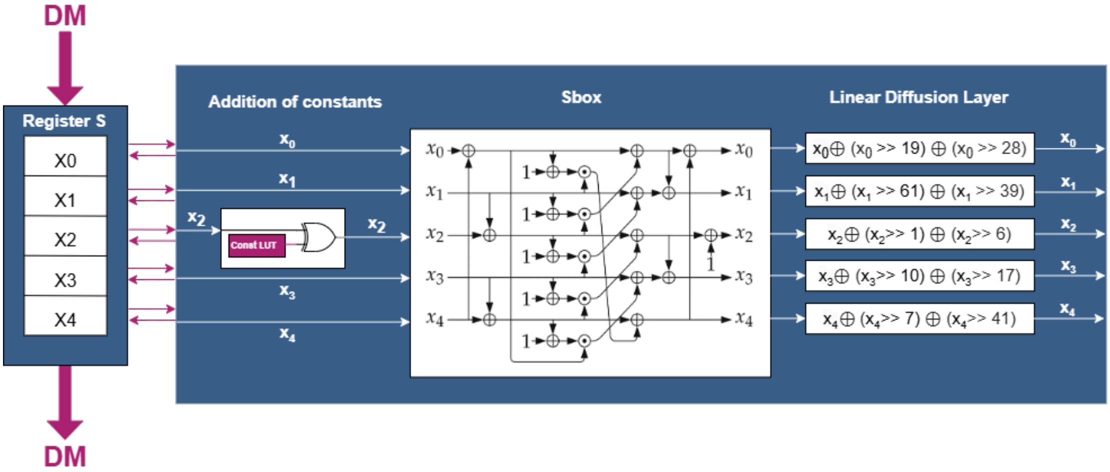

DM
Register S
X0
X1
X2
X3
X4
DM
Addition of constants
x_0
x_1
x_2
Const LUT
x_2
x_3
x_4
Sbox
x_0
x_1
x_2
x_3
x_4
1
1
1
1
1
1
Linear Diffusion Layer
$x_0 \oplus (x_0 >> 19) \oplus (x_0 >> 28)$
$x_1 \oplus (x_1 >> 61) \oplus (x_1 >> 39)$
$x_2 \oplus (x_2 >> 1) \oplus (x_2 >> 6)$
$x_3 \oplus (x_3 >> 10) \oplus (x_3 >> 17)$
$x_4 \oplus (x_4 >> 7) \oplus (x_4 >> 41)$
x_0
x_1
x_2
x_3
x_4

Figure 3.2: Block scheme of the round accelerator

There is a series of steps to perform to implement this hardware:

1. **Introduction of a new data type**. There are two kinds of data types: the ones known by the ASIP Designer compiler and the ones known by the processor (called *primitive* data types). The uint64_t format, used by the round transformation, is known just to the compiler because the processor does not have the means to handle data of 64 bits. For this reason, a new data type is implemented in the processor, called `w64`, which is associated with a new compiler data type called `ascon64_t`.

2. **Data Memory adjustments**. For the same reason as before, some modifications to the data memory are necessary since the memory can only output data with a length of 1, 2, or 4 bytes. To allow the new `w64` data to be read and written effectively, the data memory must be able to output also 8 bytes of data. The new procedures are called `ld` and `sd`.

3. **Register-S definition**. This is done in the nML language, where the register is defined as follows:

```
reg S[5] <w64,t3u> syntax(eS) read (s_r) write (s_w);
enum eS {s0 "s0", s1 "s1", s2 "s2", s3 "s3", s4 "s4"};
reg S00<w64> alias S[0] read (s00r) write (s00w);
reg S01<w64> alias S[1] read (s01r) write (s01w);
reg S02<w64> alias S[2] read (s02r) write (s02w);
reg S03<w64> alias S[3] read (s03r) write (s03w);
reg S04<w64> alias S[4] read (s04r) write (s04w);
```

The first line implements a register file with 5 locations, containing `w64` data and addressed by the `t3u` data, which is an unsigned 3-bit primitive data type. There will be one write port **s_w** and one read port **s_r** to load and store data from the data memory to the register. This is made more user-friendly by the enum `eS`, which allows to refer to a specific location as $s < location\_number >$. The last part consists of the aliases definitions that add one read and one write port for each field of the register file. Those ports are used to perform parallel computation on the data inside the register, allowing the execution of all the instructions contained originally in the loop body in a single clock cycle, in parallel. Additionally, the load and store operations are declared together with the bypass procedures. The read and write operations are the conventional ones, where one register and the immediate are used to calculate the address in the memory, and the second register contains the data to load or store. On the other hand, the bypasses are more peculiar: they are used when successive operations are performed on the register, allowing to speed up the execution. Bypassing only applies to read-after-write (RAW) data hazards, avoiding any pipeline stall when two instructions access the same register field. The compiler of ASIP Designer considers the bypasses specified in nML during scheduling, while the hardware implementation will be performed by generating multiplexers and the corresponding decoding logic. For this reason, bypassing should be done whenever possible, as the additional hardware is justified by the increased performance of the processor.

4. **Description of the hardware**. This is done in the PDG language, which allows describing the primitive function in a C++ style, that will be processed and used for the generation of the hardware description. The round transformation is described as:

```
//Round const LUT
class v12_uint8_t property (vector uint8_t [12]);
const v12_uint8_t RC = {
    0xf0, 0xe1, 0xd2, 0xc3,
    0xb4, 0xa5, 0x96, 0x87,
    0x78, 0x69, 0x5a, 0x4b
};

void round( w32 i, w64 x0_i, w64 x1_i, w64 x2_i, w64 x3_i, w64 x4_i,
            w64& x0_o, w64& x1_o, w64& x2_o, w64& x3_o, w64& x4_o)
{
    //assign inputs to local variables
    uint64_t x0 = x0_i;
    uint64_t x1 = x1_i;
    uint64_t x2 = x2_i;
    uint64_t x3 = x3_i;
    uint64_t x4 = x4_i;
    uint64_t xtemp;

    uint64_t C64 = 0;
    C64[7:0] = RC[i];

    //const addition
    x2 ^= C64;
    //Sbox
    ...
    /* linear layer */
    ...

   //assign outputs
    x0_o = x0;
    x1_o = x1;
    x2_o = x2;
    x3_o = x3;
    x4_o = x4;
}
```

As it is possible to notice, the structure is almost the same as the one of the C source code. Some steps were omitted to avoid redundancy. The inputs and the outputs of the function are primitive data types, including the `w64` mentioned before, while inside the code all types known by the compiler are allowed. The most noticeable aspect here is the definition of the LUT containing all the round constants. To properly construct the array, a new PDG class is needed, namely v12_uint8_t which is a vector of twelve uint8_t elements. This, together with the const keyword, allows the implementation of a fast look-up table.

5. **Introduction of the round primitive**. Now that the hardware is fully described, it is possible to define the microinstructions that will execute the round transformation. These are written in nML as follows:

```
fu ascon;
trn asconA <w32>;
opn Ascon_Accel (op: e_ascon_rrr, rd: mX22w_EX, rs: mX21r_EX) {
    action{
        stage ID..EX:        aguA`EX` = rs;
        stage ID..WB:
        switch (op) {
            case round:
                    stage EX:      aguR = add(aguA, aguB=0) @agu;
                    stage EX..WB:  rd = aguR`EX`;
            }
        stage EX:
            asconA = aguA;
            switch (op) {
                case round:
                  round(asconA,
                  s00r=S00, s01r=S01, s02r=S02, s03r=S03, s04r=S04,
                  S00=s00w, S01=s01w, S02=s02w, S03=s03w, S04=s04w)
                        @ascon;
            }
    }
    syntax : op PADMNM " " rd "," PADOP1 rs;
    image: op[9..3] :: rs :: "00000" :: op[2..0] :: rd :: opc32.OP;
}
```

The first row is the declaration of the functional unit (keyword *fu*) ascon, followed by the 32-bit transitory (represented by the keyword *trn*) asconA, which is the nML counterpart of a VHDL signal. This will be used to represent temporarily the index of the iteration. The following block is the definition of the actual microinstructions, which are defined in a single `opn` statement. The round parentheses contain the opcode for all the instructions defined in that statement, together with their source and destination registers. Those are represented by predefined modes of the general register file of the processor (register `X`) which describe its read and write behavior. Here the *r* and *w* characters determine whether they are read or written, the numbers are representative of the multi-port structure of the internal X register file and are important for the instructions that have more than one source register. With the *action* attribute, the behavior of the instruction is described. It is divided into phases, where at each stage of the pipeline is indicated what the instruction must do. In this case, in any of the stages between ID and EX, including both ends, the value of the source register is read and assigned to asconA. Then, between the EX and WB stages, a placeholder value is inserted as a destination register. After this, to be performed just in the execution stage, there is the body of the round primitive. The line of code associates the data to its right position inside the register S and also indicates that the operation will be performed inside the ascon functional unit. The last two lines represent the syntax and image attributes. The first attribute specifies the assembler syntax for the corresponding instruction and is used for a more understandable version of the assembler instruction. The image attribute defines the binary encoding for the corresponding instruction, whose fields are common to all the instructions of a certain type which, in this case, is the arithmetic instructions with only one source and one destination register. Nothing is written in one source and the destination registers, but these locations are necessary to maintain the r-type instruction format. In the other source register, the index of the iteration to perform is passed to the accelerator. This is made to avoid sending at each round the constant value, in such a way that the hardware is more self-sufficient with its look-up table.

6. **Declare and add to the ISA all the custom instructions** of the processor. This is done in two instances:

```
opn ascon_instr(
        Ascon_Accel //round primitive
      | load_instr_regS
      | store_instr_regS
);

opn A (
        alu_instrs
      | mpy_instrs
      | div_instrs
      | ctrl_instrs
      | zol_instrs
      | swbrk_instr
      | ascon_instrs //custom instr.
);
```

The opn keyword is used to identify an instruction that the processor can execute, grouping instructions of the same kind. In this case, first, all instructions added until now are under the *ascon_instr* operation, which in the left column is included in the arithmetic instructions of the processor.

In Figure 3.3, under the arithmetic operations, the newly added instructions are reported.

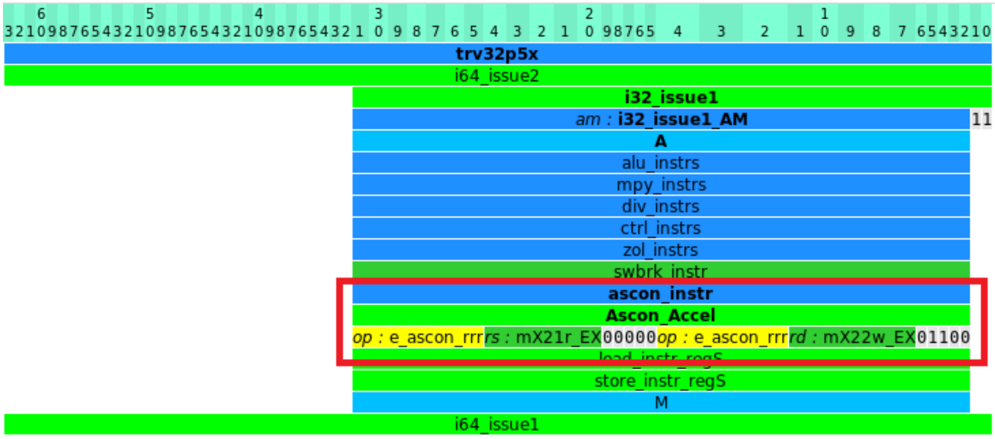
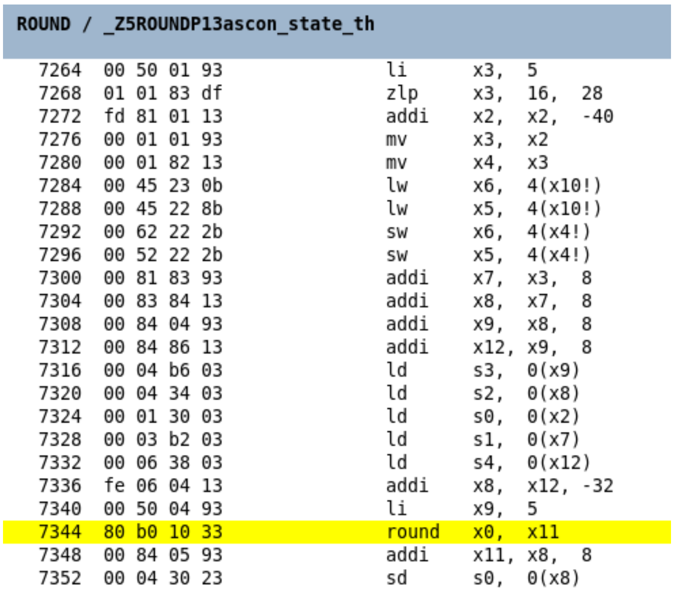


Figure 3.3: Addition of the primitive to the ISA

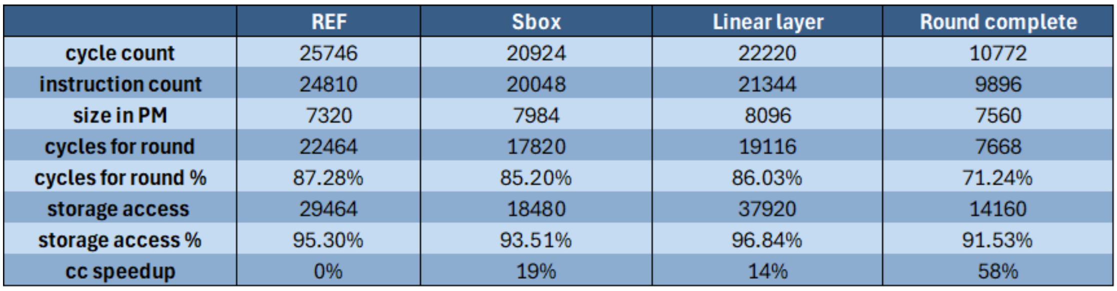
In Figure 3.4 there is the microcode just modified. Now there are just the new primitive function round and five loads and stores (partially shown) for the five 64-bit entries of the state vector.



```
ROUND / _Z5ROUNDP13ascon_state_th

 7264  00 50 01 93              li     x3,  5
 7268  01 01 83 df              zlp    x3,  16,  28
 7272  fd 81 01 13              addi   x2,  x2,  -40
 7276  00 01 01 93              mv     x3,  x2
 7280  00 01 82 13              mv     x4,  x3
 7284  00 45 23 0b              lw     x6,  4(x10!)
 7288  00 45 22 8b              lw     x5,  4(x10!)
 7292  00 62 22 2b              sw     x6,  4(x4!)
 7296  00 52 22 2b              sw     x5,  4(x4!)
 7300  00 81 83 93              addi   x7,  x3,  8
 7304  00 83 84 13              addi   x8,  x7,  8
 7308  00 84 04 93              addi   x9,  x8,  8
 7312  00 84 86 13              addi   x12, x9,  8
 7316  00 04 b6 03              ld     s3,  0(x9)
 7320  00 04 34 03              ld     s2,  0(x8)
 7324  00 01 30 03              ld     s0,  0(x2)
 7328  00 03 b2 03              ld     s1,  0(x7)
 7332  00 06 38 03              ld     s4,  0(x12)
 7336  fe 06 04 13              addi   x8,  x12, -32
 7340  00 50 04 93              li     x9,  5
 7344  80 b0 10 33              round  x0,  x11
 7348  00 84 05 93              addi   x11, x8,  8
 7352  00 04 30 23              sd     s0,  0(x8)
```

Figure 3.4: Microcode view of the new primitive

In Table 3.2 are represented the combinations: “REF” is the reference implementation with no dedicated hardware, "Sbox” implements both the addition of constants and Sbox, “Linear layer” handles just the linear diffusion layer, and “Round complete” implements the whole function.



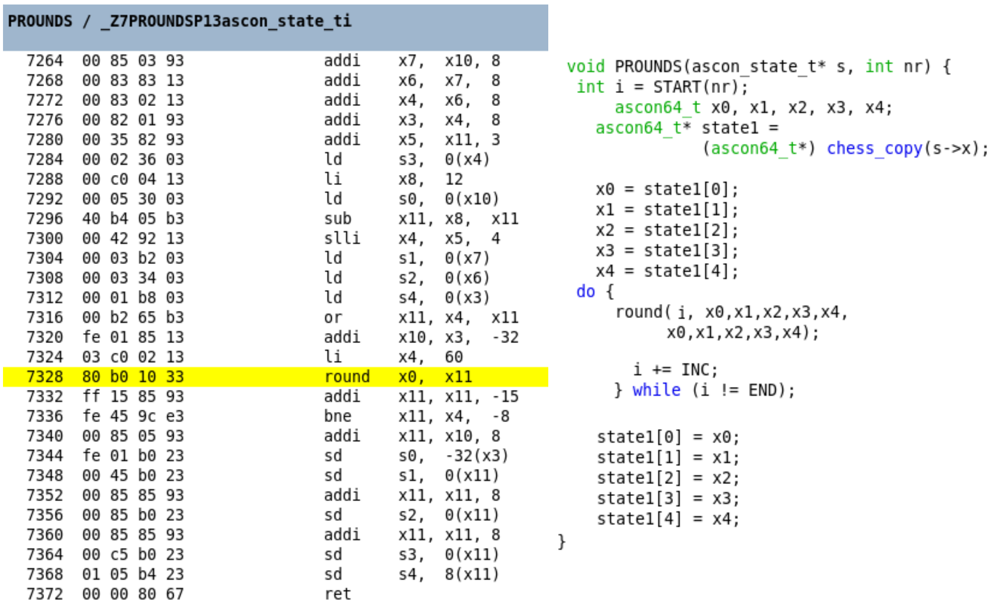
	REF	Sbox	Linear layer	Round complete
cycle count	25746	20924	22220	10772
instruction count	24810	20048	21344	9896
size in PM	7320	7984	8096	7560
cycles for round	22464	17820	19116	7668
cycles for round %	87.28%	85.20%	86.03%	71.24%
storage access	29464	18480	37920	14160
storage access %	95.30%	93.51%	96.84%	91.53%
cc speedup	0%	19%	14%	58%

Table 3.2: Comparison of proposed partial hardware accelerators

The results indicate that, to have the biggest speed-up, the whole function needs to be implemented in hardware. If one were to be in a really constrained environment, he could think to accelerate only the Sbox, but chances are the result will not be satisfactory enough. A 58% speed-up in clock cycles is a wonderful result, along with the reduction of 52% of the storage accesses.

3.2.2 Dedicated memory improvement

The reduction of 52% on the storage accesses is a good result, but the round function alone is still responsible for 91.53% of the total reads (always referring to table 3.2). This happens because each time the function is called, the state vector is loaded. For example, if a permutation with *a* equal to 12 is carried out, it means that the state vector will be loaded through five cycles twelve times, which will waste 60 cycles. If the register were used in a more advanced way, one could perform only the first five cycles to load the data, **storing the partial results of the permutation** inside this memory area. This is done in software because the register is already predisposed to support this behavior. The starting situation is the one of code A.2, from there the round function is deleted and the round primitive is used directly in the PROUND function. In this way, avoiding the function call and the constant conversions between C and processor datatypes, the inputs and outputs of the S register can be directly connected between one cycle of the loop and the following one. The new full code is in the appendix, A.3 and the microcode that performs a **a whole permutation** is in figure 3.5.



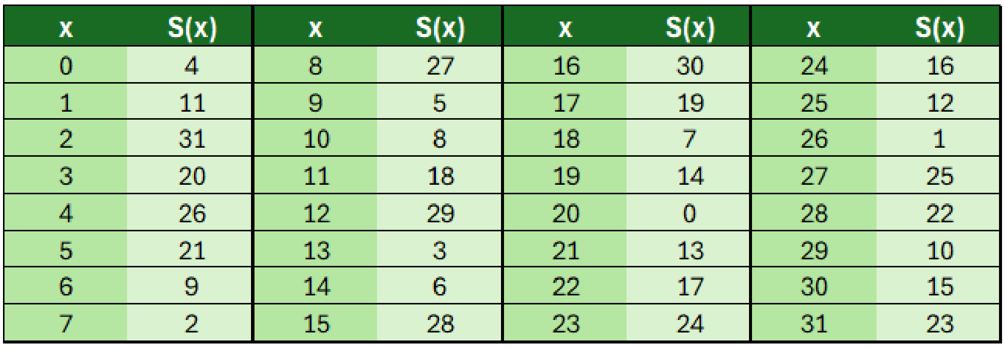
```
PROUNDS / _Z7PROUNDSP13ascon_state_ti

 7264  00 85 03 93              addi   x7,  x10, 8
 7268  00 83 83 13              addi   x6,  x7,  8
 7272  00 83 02 13              addi   x4,  x6,  8
 7276  00 82 01 93              addi   x3,  x4,  8
 7280  00 35 82 93              addi   x5,  x11, 3
 7284  00 02 36 03              ld     s3,  0(x4)
 7288  00 c0 04 13              li     x8,  12
 7292  00 05 30 03              ld     s0,  0(x10)
 7296  40 b4 05 b3              sub    x11, x8,  x11
 7300  00 42 92 13              slli   x4,  x5,  4
 7304  00 03 b2 03              ld     s1,  0(x7)
 7308  00 03 34 03              ld     s2,  0(x6)
 7312  00 01 b8 03              ld     s4,  0(x3)
 7316  00 b2 65 b3              or     x11, x4,  x11
 7320  fe 01 85 13              addi   x10, x3,  -32
 7324  03 c0 02 13              li     x4,  60
 7328  80 b0 10 33              round  x0,  x11
 7332  ff 15 85 93              addi   x11, x11, -15
 7336  fe 45 9c e3              bne    x11, x4,  -8
 7340  00 85 05 93              addi   x11, x10, 8
 7344  fe 01 b0 23              sd     s0,  -32(x3)
 7348  00 45 b0 23              sd     s1,  0(x11)
 7352  00 85 85 93              addi   x11, x11, 8
 7356  00 85 b0 23              sd     s2,  0(x11)
 7360  00 85 85 93              addi   x11, x11, 8
 7364  00 c5 b0 23              sd     s3,  0(x11)
 7368  01 05 b4 23              sd     s4,  8(x11)
 7372  00 00 80 67              ret
```

```
void PROUNDS(ascon_state_t* s, int nr) {
 int i = START(nr);
     ascon64_t x0, x1, x2, x3, x4;
   ascon64_t* state1 =
              (ascon64_t*) chess_copy(s->x);

   x0 = state1[0];
   x1 = state1[1];
   x2 = state1[2];
   x3 = state1[3];
   x4 = state1[4];
 do {
     round( i, x0,x1,x2,x3,x4,
           x0,x1,x2,x3,x4);

       i += INC;
    } while (i != END);

   state1[0] = x0;
   state1[1] = x1;
   state1[2] = x2;
   state1[3] = x3;
   state1[4] = x4;
}
```

Figure 3.5: Microcode view of a whole permutation

It is not possible anymore to use the term instruction set extension because, due to the improvement in the register usage, one of the laws on ISE was violated. Each instruction should store its result inside the data memory each time is performed, not into a local register. The optimization that is being performed is a **tightly coupled acceleration**.

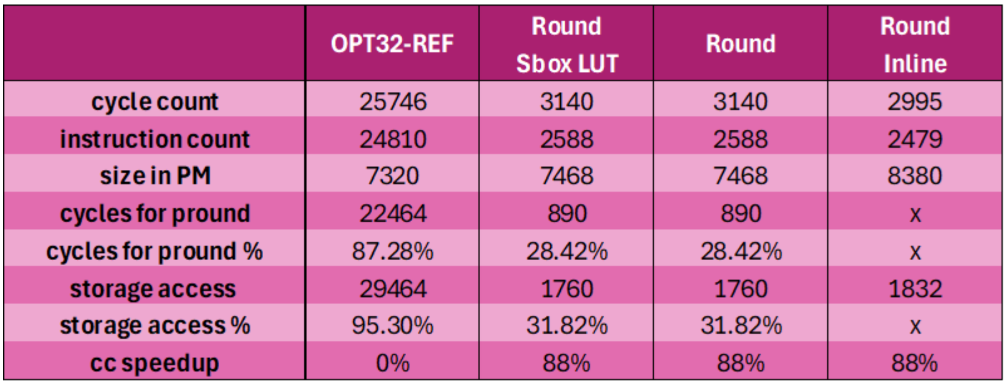
Another interesting design choice involves the replacement of the logic in the substitution layer with a LUT. This, theoretically, should make the computation faster and easier because the calculations have to be performed only once and then stored inside the memory. On the other hand, it increases the susceptibility to side-channel attacks as the computation of the state vectors becomes more predictable. The LUT that performs the Sbox is represented in Table 3.3.



x	S(x)	x	S(x)	x	S(x)	x	S(x)
0	4	8	27	16	30	24	16
1	11	9	5	17	19	25	12
2	31	10	8	18	7	26	1
3	20	11	18	19	14	27	25
4	26	12	29	20	0	28	22
5	21	13	3	21	13	29	10
6	9	14	6	22	17	30	15
7	2	15	28	23	24	31	23

Table 3.3: LUT of the Sbox

The results coming from all the implementations discussed are reported in Table 3.4.



	OPT32-REF	Round Sbox LUT	Round	Round Inline
cycle count	25746	3140	3140	2995
instruction count	24810	2588	2588	2479
size in PM	7320	7468	7468	8380
cycles for pround	22464	890	890	x
cycles for pround %	87.28%	28.42%	28.42%	x
storage access	29464	1760	1760	1832
storage access %	95.30%	31.82%	31.82%	x
cc speedup	0%	88%	88%	88%

Table 3.4: Results of the full round accelerator

With the improvement of the dedicated memory, the number of storage accesses has decreased significantly, consequently affecting the cycle count that dropped by 88%, meaning an **acceleration of ×8.6**. The last column shows the effect of using round as an inline function. The number of cycles is reduced, as the overhead for the function calls and returns disappear, but as a consequence, the size of the program memory becomes bigger. The difference in the two parameters is minimal and both are valid. On the other hand, it can be observed that using a LUT as a Sbox has no effect in terms of cycle count because in both cases they perform the computation in one clock cycle.

In the appendix, codes A.1 and A.3 are the source files that implement the round function before and after the optimization. It is possible to note how lighter the code becomes when dedicated instructions are used in these situations, where the functions are complicated and used often.

The function and storage reports now are (the full reports are in appendix B.3 and B.4):

```
Calls  Cycles tot   Cycles tot  Cycles tot   Cycles tot     Function
           (func)      (%func) (func+desc) (%func+desc)
----- ----------- ------------ ----------- ------------ --------------
   14         890       28.42%         890       28.42%  PROUNDS
   24         720       22.99%         720       22.99%  memcpy
    1         664       21.20%        1469       46.90%  aead_decrypt
  ...         ...          ...         ...          ...       ...
```

```
Storage   Read-count  Read-count   Read-count  Function name
             (total)  (function) (% of total)
------- ----------- ----------- ------------   --------------------
DMb            1760         560       31.82%   PROUNDS
                            492       27.95%   ascon_aead_decrypt
S               610         610      100.00%   PROUNDS
...             ...         ...          ...        ...
```

3.2.3 Unrolling

Despite the great improvements, PROUND is still the most computationally expensive function in terms of cycles. As it includes a loop, it is natural to try to **apply unrolling**. This technique consists of repeating the body of the loop multiple times, changing the termination code and getting rid of redundant branch instructions, increasing instruction-level parallelism [9]. This also means that more area will be occupied, but it will be possible to perform more than one round function in one clock cycle.

The first decision to be made concerns how many times to unroll the function, which corresponds to how many round iterations will be performed in the same clock cycle. The block that performs the unrolling will be called “Px”, where x stands for how many rounds are executed inside. It must be remembered that the goal of this thesis is to implement an accelerator that works on the whole Ascon family, whose number of rounds performed in a permutation differs significantly. A first approach to unrolling will involve the greatest common divisor between the parameters. Referring to Table 2.2, it is possible to see that the potential combinations are: 6, 8, and 12, whose shared common factor is 2. The rough picture of P2 is represented in Figure 3.6, to demonstrate that the hardware gets duplicated.

The other possible choice is to implement an unrolling for all kinds of permutation. This means that the data will pass through a series of blocks where they are deviated based on the number of permutations to be performed. A representation of this behavior is in Figure 3.7.

This structure allows reusing the blocks to occupy as much area as possible. For example, if one had to perform all twelve rounds, the data would pass through the P4, P2, and P6 blocks, while for six it would pass directly to the P6 blocks.

The results coming from the simulations of these approaches are summarized in Table 3.5.

It is possible to observe that also the case of P6 was explored. This is noteworthy for ascon128 because this algorithm performs permutations of six and twelve rounds that would benefit from this decision.

The results confirm that unrolling is beneficial for the acceleration of the algorithm in terms of the cycle count. This is due to two factors: the first is that a permutation can be performed in one clock cycle instead of the six, eight, or twelve more, and

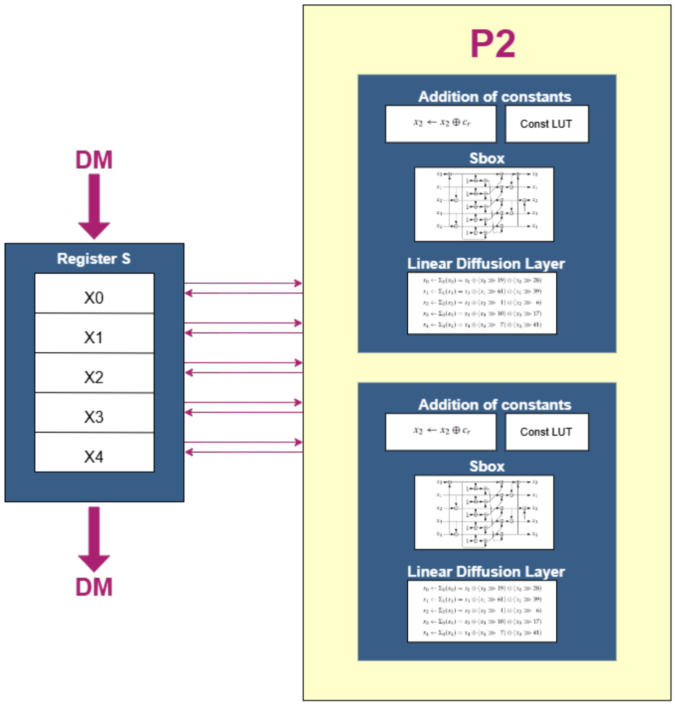


Figure 3.6: Outline of P2

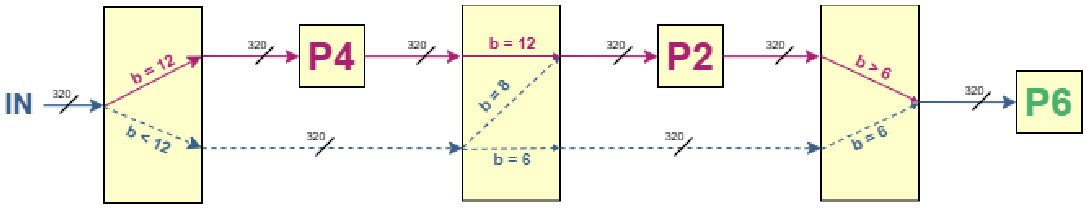
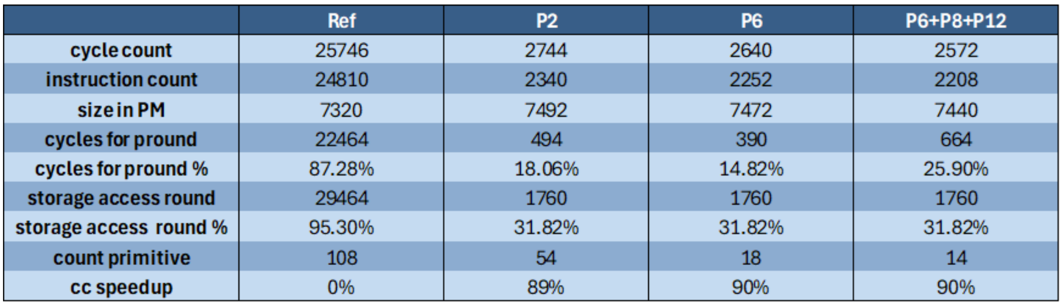


Figure 3.7: Outline of all permutations unrolled

the second concerns the parameter *count primitive*. This value reports how many times the new instruction is called. In the base implementation, the round function is called 108 times, while adding P2 halves this number, which decreases with higher unrolling factors. Having fewer function calls means having less overhead both for the calls and returns, reducing the cycles needed to perform the algorithm. Returning to the P6 case, it is clear that it brings great benefits to the algorithm because the major part of permutations (that are p^b of six rounds, as was discussed in chapter 2) are completely performed in one cycle, while in the few cases where p^a is needed, it is heavily accelerated.



	Ref	P2	P6	P6+P8+P12
cycle count	25746	2744	2640	2572
instruction count	24810	2340	2252	2208
size in PM	7320	7492	7472	7440
cycles for pround	22464	494	390	664
cycles for pround %	87.28%	18.06%	14.82%	25.90%
storage access round	29464	1760	1760	1760
storage access round %	95.30%	31.82%	31.82%	31.82%
count primitive	108	54	18	14
cc speedup	0%	89%	90%	90%

Table 3.5: Results of the approaches to unrolling

With unrolling, as reported in the following function report (also present in appendix B.5), the round function is not the most critical anymore.

```
Calls  Cycles tot   Cycles tot  Cycles tot   Cycles tot    Function
           (func)      (%func) (func+desc) (%func+desc)
------ ----------- ------------ ----------- ------------ --------------
    24         720       26.32%         720       26.32%   memcpy
     1         664       24.27%        1271       46.45%   aead_decrypt
     1         633       23.14%        1240       45.32%   aead_encrypt
    14         494       18.06%         494       18.06%   PROUNDS
   ...         ...          ...         ...          ...      ...
```

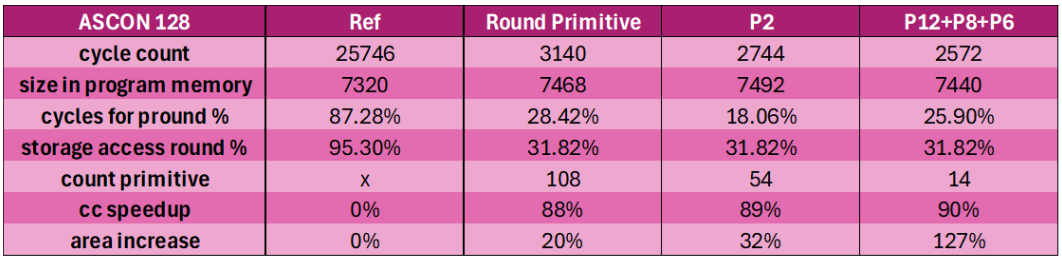
The new most critical function is `memcpy`, which has the functionality of transferring data from the memory. Of course, this instance cannot be substituted, so the iterative optimization has now ended.

3.3 Results

A summary of all implemented designs is reported in Table 3.6.

The speedup can also be visually represented through Figure 3.8, where the whole circle is the total cycle count of the reference implementation, and each slice represents how many cycles are reduced with that design.

One parameter that was greatly improved by the accelerators is the storage accesses in memory, due to the introduction of the dedicated memory. It is possible to observe that this number stays constant through all the designs proposed because



ASCON 128	Ref	Round Primitive	P2	P12+P8+P6
cycle count	25746	3140	2744	2572
size in program memory	7320	7468	7492	7440
cycles for pround %	87.28%	28.42%	18.06%	25.90%
storage access round %	95.30%	31.82%	31.82%	31.82%
count primitive	x	108	54	14
cc speedup	0%	88%	89%	90%
area increase	0%	20%	32%	127%

Table 3.6: Results of the designs discussed on Ascon128a

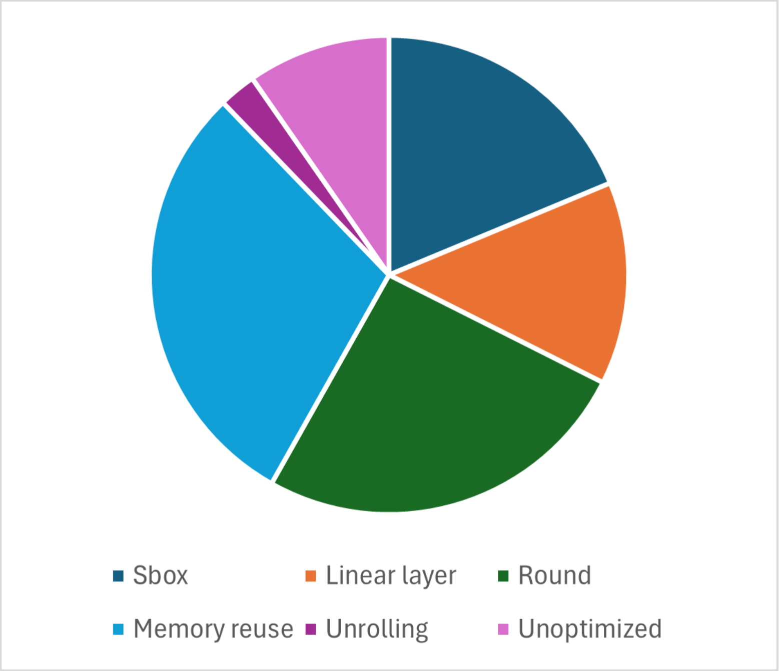


Figure 3.8: Diagram of the cycle count reduction

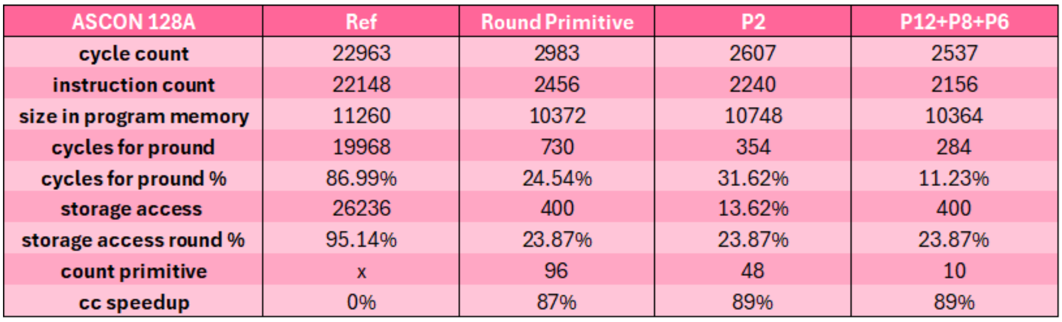
unrolling does not affect this. One possible solution that would reduce this number would be to directly use the register file as a dedicated memory, reserving some spaces for the state vector and introducing multiplexers that alternate the storage in the reserved spaces between the round function execution and the normal usage; as proposed in [10]. The downside of this solution is that the core of the processor was changed, greatly impacting the flexibility of the RISC-V.

3.3.1 Impact on area

Table 3.6 also adds another parameter to the picture: the **area increment** due to the introduced hardware. The reported value is calculated for the whole processor. Unrolling all possibilities deals with the biggest speedup but at the cost of too much area. A more reasonable solution, considering that the objective is to accelerate a function for lightweight cryptography, would be to choose either the standard acceleration or the P2. The difference in occupied area and performance is not too substantial, based on the kind of application they are both valid. In this case, the simple round is more advantageous because adding 10% more area just to achieve a 1% more speedup is not a good enough deal, remembering that the resources are constrained. For what concerns the idea of substituting the Sbox with the LUT, the synthesis highlights a negligible difference between the implementations. In fact, the LUT modification uses less than 100 fewer cells, meaning that this implementation is not worth reducing the resistance to side-channel attacks.

3.3.2 Effect on the rest of the family

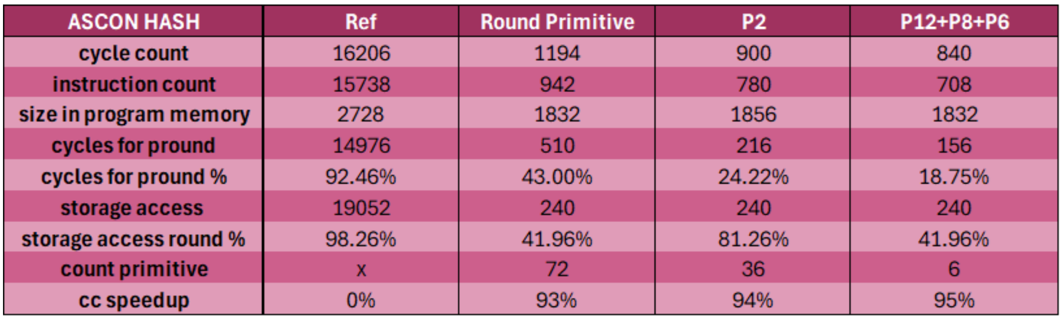
Tables 3.7, 3.8 and 3.9 report the effects of the designed accelerators in the Ascon family.



ASCON 128A	Ref	Round Primitive	P2	P12+P8+P6
cycle count	22963	2983	2607	2537
instruction count	22148	2456	2240	2156
size in program memory	11260	10372	10748	10364
cycles for pround	19968	730	354	284
cycles for pround %	86.99%	24.54%	31.62%	11.23%
storage access	26236	400	13.62%	400
storage access round %	95.14%	23.87%	23.87%	23.87%
count primitive	x	96	48	10
cc speedup	0%	87%	89%	89%

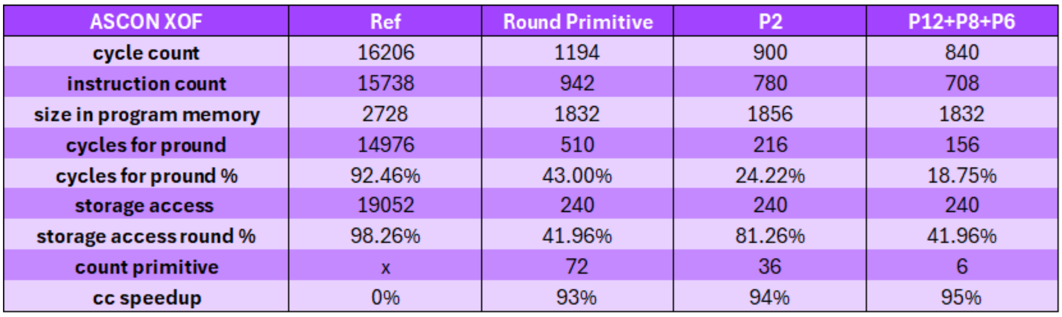
Table 3.7: Results of the designs discussed on Ascon128a

Notice that Ascon-Hash and Ascon-XOF have the same performances because the XOF outputs a message of the same length of the hash. It is possible to see that the same hardware affects differently the four algorithms, based on how many times the round function is called. The hash and XOF functions are the ones that benefit the most from the accelerator because they perform twelve rounds for each permutation. Again, just optimizing one round at a time deals a great performance increment alone, reaching up to 93% speedup.



ASCON HASH	Ref	Round Primitive	P2	P12+P8+P6
cycle count	16206	1194	900	840
instruction count	15738	942	780	708
size in program memory	2728	1832	1856	1832
cycles for pround	14976	510	216	156
cycles for pround %	92.46%	43.00%	24.22%	18.75%
storage access	19052	240	240	240
storage access round %	98.26%	41.96%	81.26%	41.96%
count primitive	x	72	36	6
cc speedup	0%	93%	94%	95%

Table 3.8: Results of the designs discussed on Ascon-Hash



ASCON XOF	Ref	Round Primitive	P2	P12+P8+P6
cycle count	16206	1194	900	840
instruction count	15738	942	780	708
size in program memory	2728	1832	1856	1832
cycles for pround	14976	510	216	156
cycles for pround %	92.46%	43.00%	24.22%	18.75%
storage access	19052	240	240	240
storage access round %	98.26%	41.96%	81.26%	41.96%
count primitive	x	72	36	6
cc speedup	0%	93%	94%	95%

Table 3.9: Results of the designs discussed on Ascon-XOF

Chapter 4

Coprocessor

The designed accelerator achieves a great performance, but its integration requires significant changes to the core's architecture, meaning that it is not flexible when it is used in other applications. If, instead, one wants to maintain flexibility while still being integrated into the pipeline of the core, the right choice is to adopt a **coprocessor**. The only modification that is needed in this case is to add the interface, so that whichever core with the required connection can use this and other accelerators, that all communicate with the same protocol.

4.1 CV-X-IF

The eXtension interface provides tightly coupled and low-latency access to the CPU. Each opcode that is not used by the processor can be assigned to the interface, through the use of a **handshake protocol**.

The most important signals that characterize the interface are:

- **issue elements**: `issue_valid` indicates that the dispatcher wants to pass on an instruction, `issue_req` is the instruction opcode from the ID stage of the core, `issue_ready` signals if the instruction is effectively one to be performed by the accelerators and `issue_resp` indicates if the received opcode refers to one of the coprocessors attached to the interface;
- **commit elements**: signals if the offloaded instruction has a valid commit or kill information;

- **results elements**: `results_ready signals` if the results can be accepted by the core, `results_valid` indicates that the coprocessor has a valid result and `results` is a data packet containing all the results.

The direction and timing of the aforementioned signals are reported in Figure 4.1.

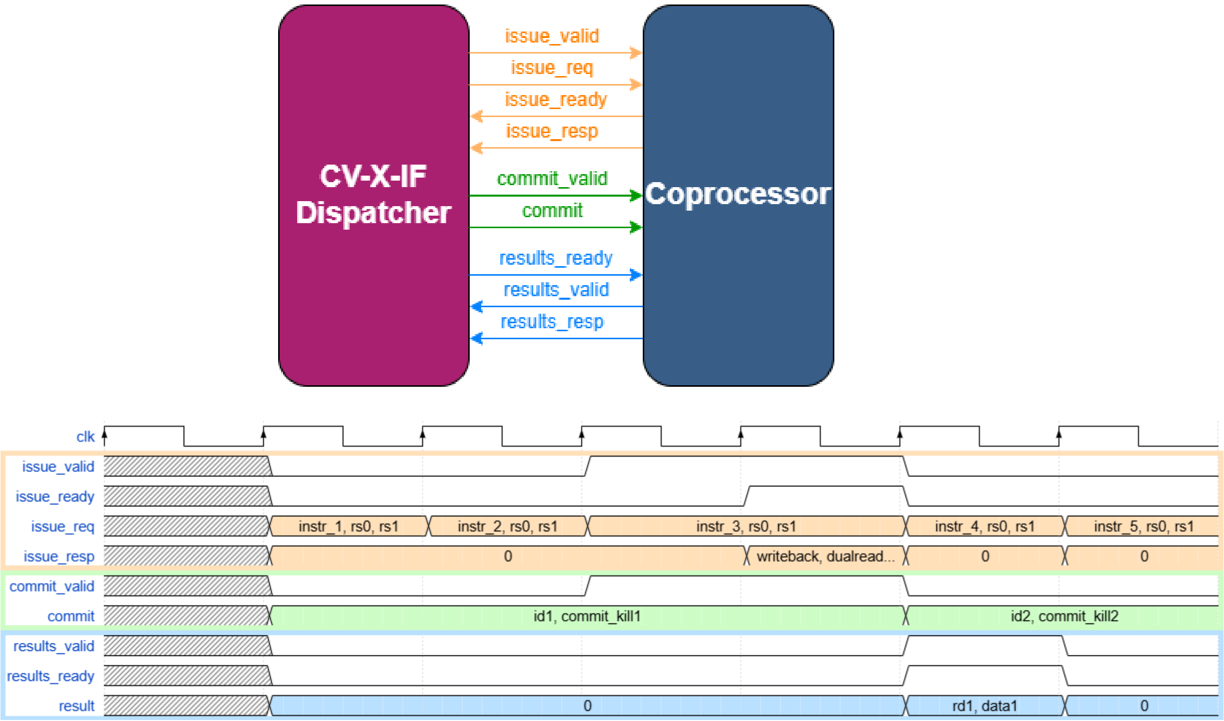


Figure 4.1: Signals that characterize the CV-X-IF interface

4.2 X-HEEP microcontroller

The accelerator will be placed in the microcontroller X-HEEP (eXtendable Heterogeneous Energy-Efficient Platform). This device is a 32-bit RISC-V-based microprocessor characterized by power domains targeted for ultra-low-power edge-computing applications. The processor architecture is reported in figure 4.2, where it is possible to distinguish between the CPU subsystem domain, memory banks domain, peripheral subsystem domain, and always-on peripheral subsystem domain.

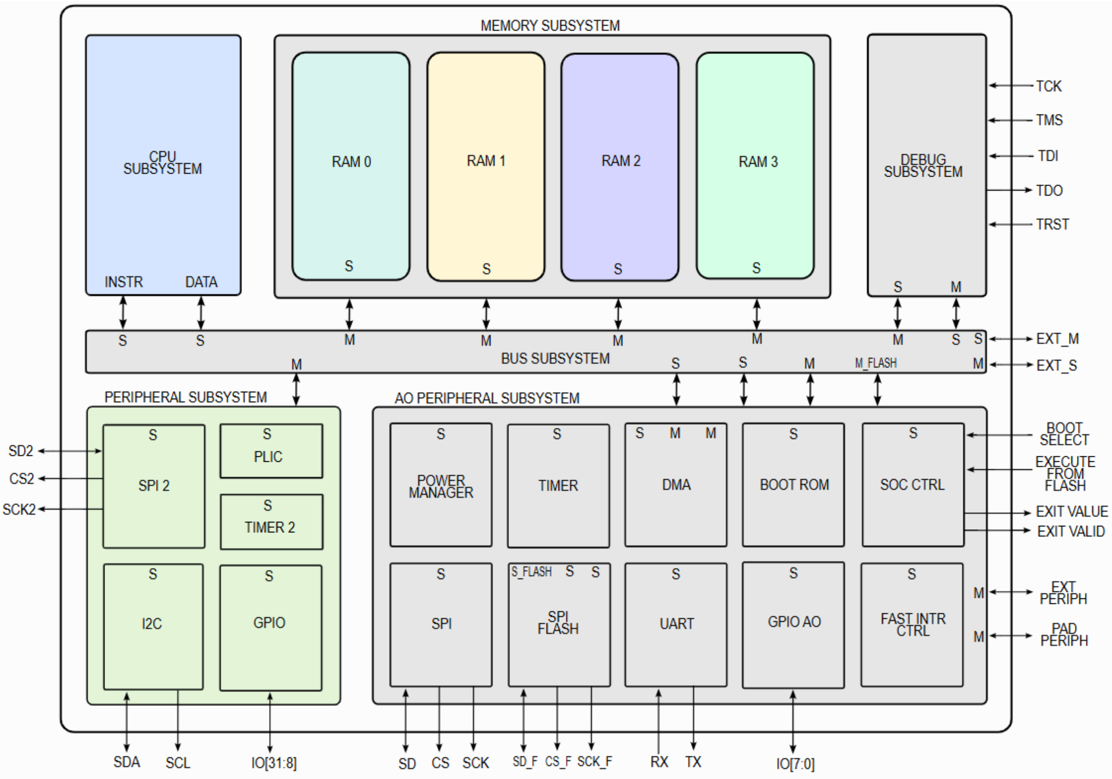


Figure 4.2: X-HEEP architecture

This processor supports the CV-X-IF interface. The architecture of the CPU is modified as reported in Figure 4.3.

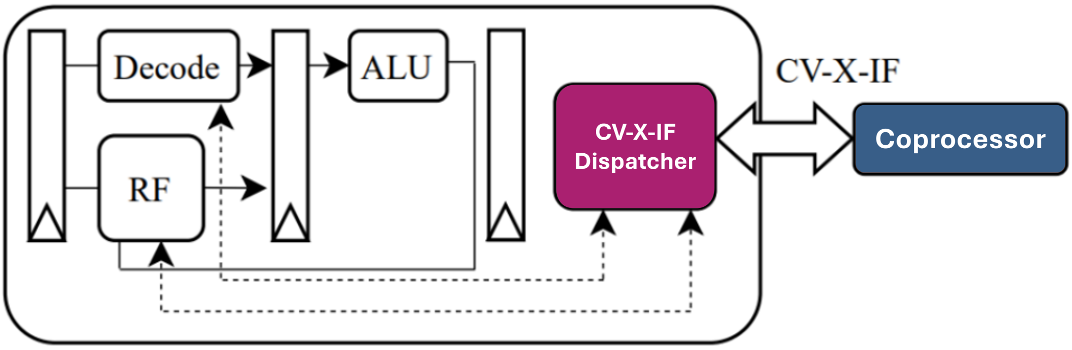


Figure 4.3: Block scheme of the processor with the added CV-X-IF interface

From this figure is clear what has been said before: the accelerator is still tightly coupled inside the pipeline of the processor, but the CPU retains most of its original architecture. Of course, the obtained performance will not be the same as in the ISE case, because, at each interaction, there is a handshake protocol to respect instead of directly executing the instruction.

4.3 Design of the coprocessor

The coprocessor has been designed with a modular approach, each block has been tested and validated before being integrated into the structure.

The top structure is the wrapper, represented in figure 4.4.

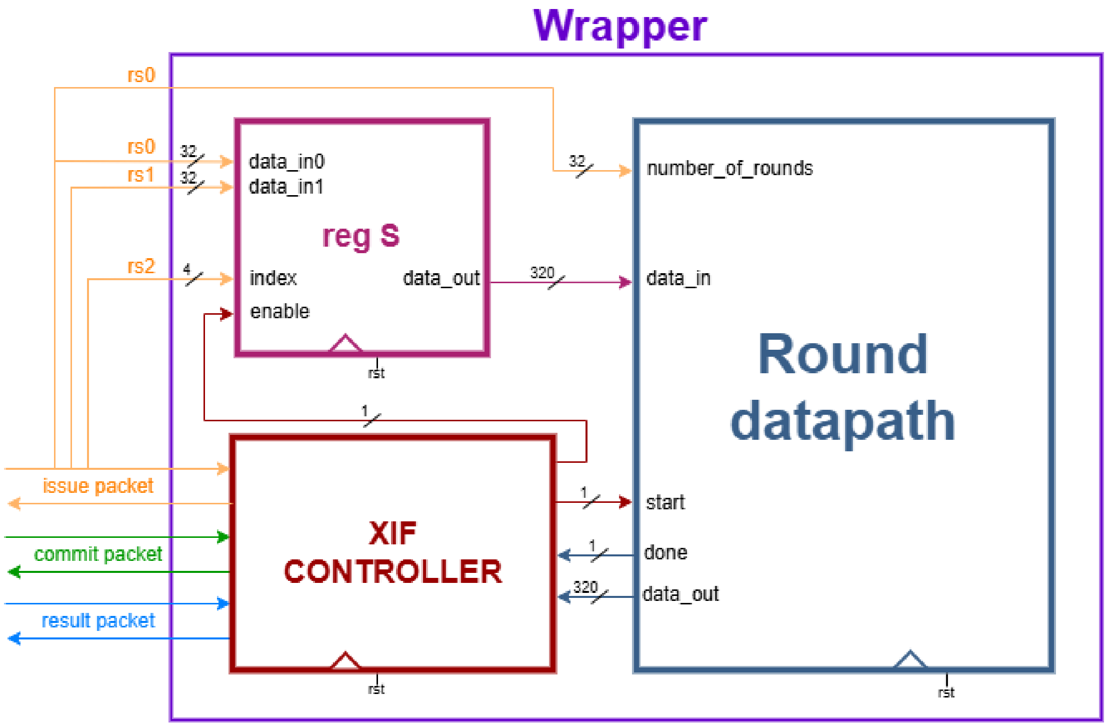


Figure 4.4: Block scheme of the wrapper of the coprocessor

It is composed of:

- **XIF CONTROLLER**, its role is to unpack the data that comes from the dispatcher. It works as a control unit with the ASM of Figure 4.5, where, at the corresponding state, it gives commands to store the input data into the register S or to perform the round permutation. At the end of the permutation, it creates the packet to send back to the dispatcher;

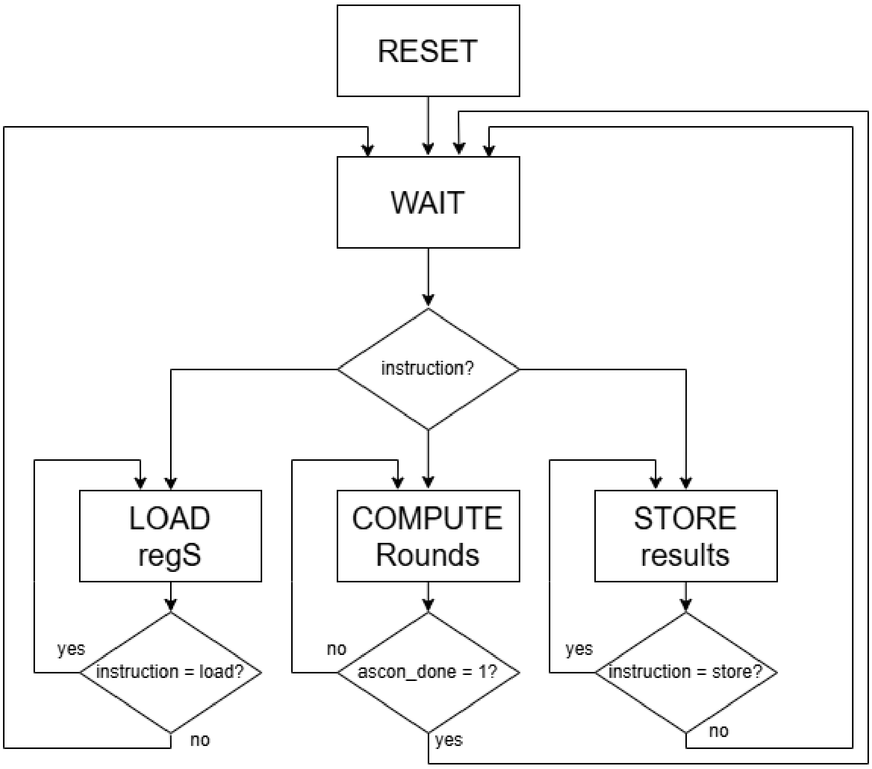


Figure 4.5: ASM of the XIF controller

- **regS**, is the register that stores the state vector. It is composed of ten registers, each of 32 bits, addressed through four index bits. Its input data are given through the source registers of the instruction `rs0` and `rs1`, while `rs2` is used for the index. The enable is active in the “LOAD regS” state;
- **round datapath**, is where the round is computed. Its structure is expanded in Figure 4.6.

  In turn, it consists of a control unit and a datapath. The CU implements the FSM in Figure 4.7, with the timing in Figure 4.8. From these, it is possible to see that the computation can start only when the command is given with the instruction from the microcontroller and the datapath has finished its last computation. This is made to avoid errors if the program wrongly calls the instruction when the one before is not finished.

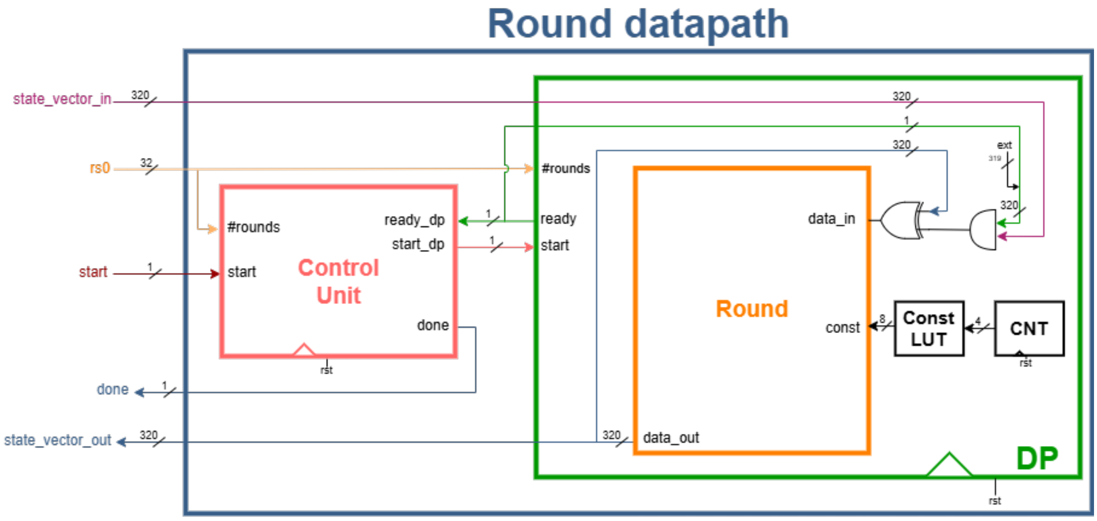


Figure 4.6: Block scheme of the datapath of the coprocessor

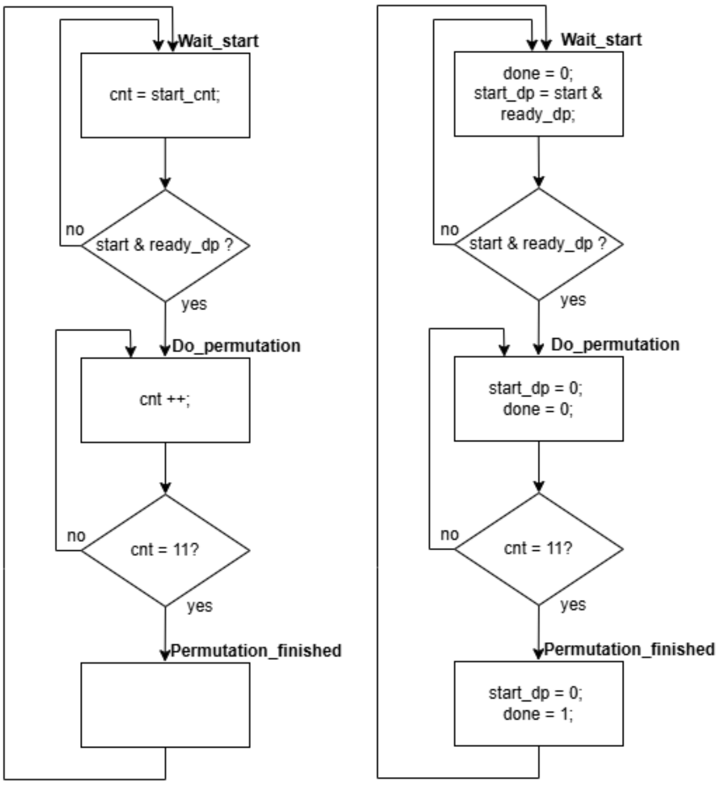


Figure 4.7: FSM of the CU of the datapath

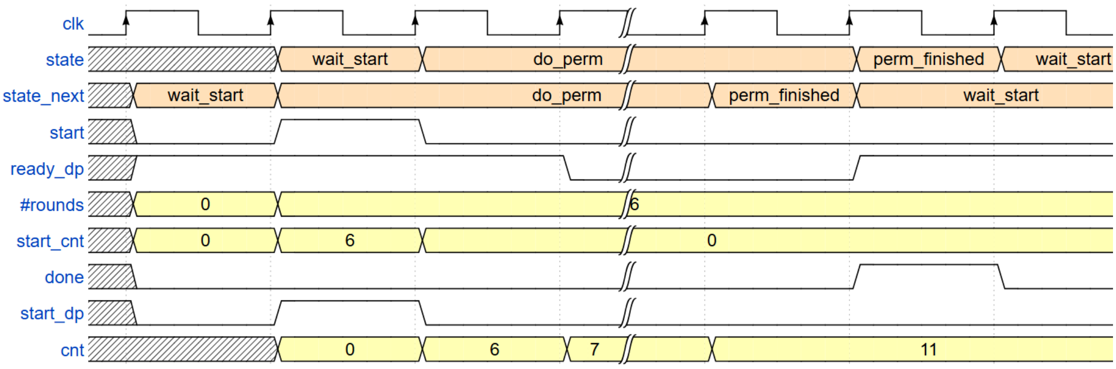


Figure 4.8: Timing of the CU of the datapath

Another important parameter is the number of rounds to be performed, for simplicity, "#rounds" in Figure 4.6. This is given with the permutation instruction and indicates how many rounds have to be executed, a factor that affects the starting value of the internal counter. If twelve rounds are done, then the counter will start at zero, if eight the starting number will be two, and six for six permutations, like in the example in Figure 4.8. In this way, the condition to stop the permutation remains the same for all possibilities and, also, the same counter can be used to choose the appropriate constant to send to the round block.

The last noteworthy aspect of the datapath in Figure 4.6 is the logic that precedes the "data_in" of the round block, which consists of the equation:

$$data\_in = (ready \cdot state\_vector\_in) \oplus data\_out$$

The input data can either be the state_vector_in, coming from the instruction and saved in the register S, or from the last output of the round datapath. If the operation to be performed is the first round of the permutation, the ready signal will be one and data_out will be all zeros, simplifying the equation:

$$data\_in = (1 \cdot state\_vector\_in) \oplus 0$$

Meaning that the data_in will be equal to the state from the reg S. On the other hand, if a round transformation was already executed, the ready signal will be zero, making the output of the parenthesis zero and the data_in equal to the data_out. In the equation:

$$data\_in = (0 \cdot state\_vector\_in) \oplus data\_out$$

Now that the hardware part of the coprocessor is completed, also the software must be modified to give the correct instructions to the dispatcher. The full round function is reported in the appendix as code A.4.

The first operation to be performed is storing the state vector inside the register S. This is done in five instructions, each giving as the two source registers the 64 bits of the x vector and, as an immediate, the location of the register S where the lowest 32 bits are put. The remaining part will be inserted in the following position inside the memory. The code that implements these loads is:

```
asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    [rs1] "r" (s->x[0] ), [rs2] "r" ((s->x[0] >> 32) ), [i] "r" (0): );
asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    [rs1] "r" (s->x[1] ), [rs2] "r" ((s->x[1] >> 32) ), [i] "r" (2): );
asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    [rs1] "r" (s->x[2] ), [rs2] "r" ((s->x[2] >> 32) ), [i] "r" (4): );
asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    [rs1] "r" (s->x[3] ), [rs2] "r" ((s->x[3] >> 32) ), [i] "r" (6): );
asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    [rs1] "r" (s->x[4] ), [rs2] "r" ((s->x[4] >> 32) ), [i] "r" (8): );
```

The `asm` command allows giving an assembler instruction using C expressions, directly assigning the data without the need to explicitly know in which memory location they are stored. The `volatile` keyword prevents the rescheduling of the instructions, and is used in these and the following instructions because they need to be performed in the correct order to work. The other sections associate one-half of the `x` vector to each source register and the memory location inside the register S as an immediate. After storing all the state vector, the command to start the permutation is given:

```
asm volatile (".insn r 0x4b, 0x004, 1, x0, %[rs1], %[rs2], x0\n\t" : :
    [rs1] "r" (1) , [rs2] "r" (nr): );
```

The first source register represents the start of the computation, while the second indicates how many rounds have to be performed. During the execution, other instructions must not be scheduled, because the state vector is not the correct one yet. For this reason, it is necessary to add as many bubbles as the cycles needed for the execution:

```
for(int i=0;i<nr;i++){
    asm volatile ("nop");
}
```

After the execution, the results are stored as:

```
for(int i=0; i<10; i+=2){
    asm volatile (".insn r 0x4b, 0x004, 2, %[rd_low], %[rs1], x0,
    x0\r\n": [rd_low] "=r" (x_low) : [rs1] "r" (i): );
    asm volatile (".insn r 0x4b, 0x004, 2, %[rd_high], %[rs1], x0,
    x0\r\n": [rd_high] "=r" (x_high): [rs1] "r" (i+1): );
    s->x[i/2] = ((uint64_t) x_high << 32) | x_low;
}
```

To test the functionality of both the hardware and the software code, they were tested alone, creating a function that calls just one round transformation. The coprocessor was added to the X-HEEP microcontroller, where it was tested with all possible number of iterations. The standard version of the microcontroller uses 1053 cycles to carry out a permutation of six rounds, with the coprocessor the cycles come down to 117. This is a **speedup of ×9** on the permutation only, which is an optimal outcome.

The components are ready to be put together in the RISC-V and be validated. The correct way of testing cryptographic primitives is to use the generalized key agreement testing (**GenKat**) files, which provide test vectors for inputs and outputs, in this case: key, plaintext, nonce, associated data, and ciphertext. If the result of the computation is the same as the one of the GenKat, it means that the addition of the coprocessor did not produce unwanted errors. The Ascon submission already provided a series of these files, as they were added during the standardization processes, but more were generated to fully test the architecture and to assure reliability. This procedure was performed just on the ascon128 version because it is long to perform and just one well-constructed process can tell if any changes were brought to the algorithm. The other algorithms were tested, just for safety, in a simpler manner comparing the original message to the one decrypted. This method is the one followed to obtain the results in chapter 5, where all outcomes will be discussed.

4.4 Synthesis

The processor can now be integrated into a hardware platform such as Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). This is a key step of the design because the performance evaluation cannot be done only in terms of clock cycles, but it also needs to be assessed in regard to occupied area, operating frequency, and throughput.

4.4.1 FPGA

FPGAs are reconfigurable structures that allow rapid prototyping and development of designs. For this reason, they are ideal for scenarios requiring flexibility and shorter manufacturing cycles. This integration emphasizes flexibility and adaptability but at the cost of reduced performance and higher power consumption compared to ASICs. The design will be integrated into the Xilinx® Artix®-7 family of FPGAs, which are the perfect fit for cost-sensitive applications that need high-end features, providing high performance with low power consumption [11].
The design suite used to implement the design is Vivado, also developed by Xilinx. It includes a comprehensive IDE for synthesis, simulation, and analysis, as well as advanced optimization techniques for power and performance [12].

The chosen board has a clock of 300 MHz, that is used to generate the internal clock of the design, which, to have the biggest throughput possible, will be the maximum operating frequency. This value is unknown, but it is possible to find it by putting a constraint on the clock frequency and seeing if it is met or if the slack is negative. This step was repeated until the slack was as small as possible, and the obtained maximum frequency was 40 MHz, as shown in the report below:

```
 Setup                              Hold
-------------------------------  ---------------------------------
Worst Negative Slack: 0.407 ns    Worst Hold Slack: 0.010 ns
Total Negative Slack: 0.000 ns    Total Hold Slack: 0.000 ns
Number of Failing Endpoints: 0    Number of Failing Endpoints: 0
```

```
Total Number of Endpoints:74425   Total Number of Endpoints: 74425
```

The implemented design is reported in Figure 4.9, where it is possible to see the S register and the datapath.

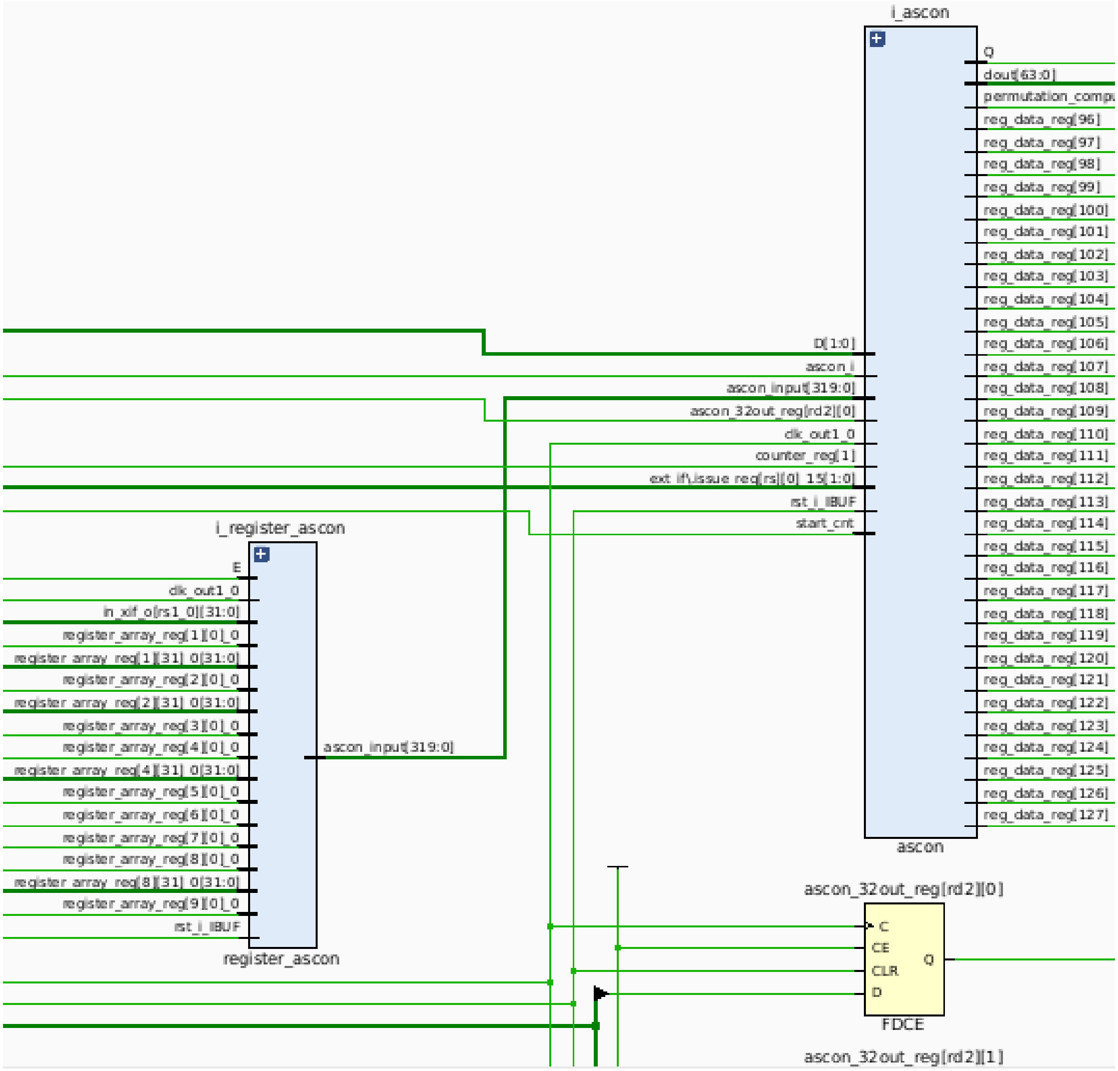


Figure 4.9: Design implemented in the FPGA, register, and datapath of the coprocessor

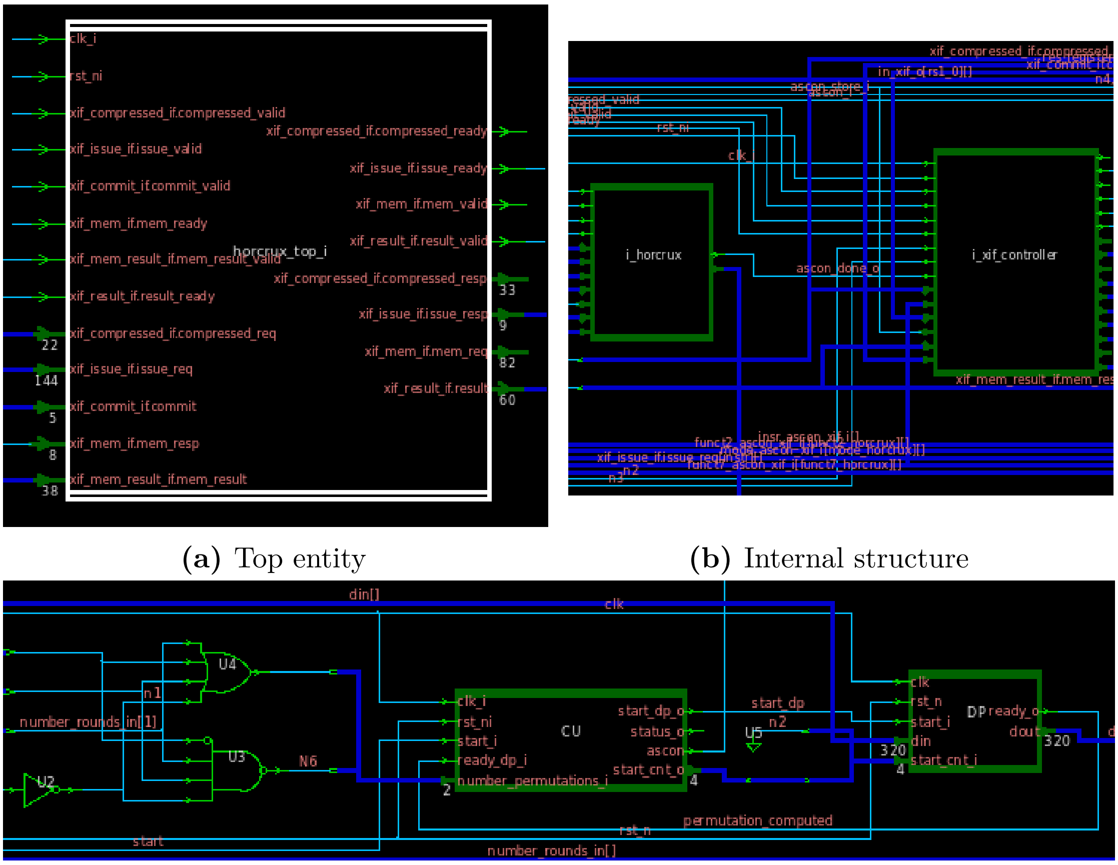
4.4.2 ASIC

ASICs are designed for specialized, high-performance applications where power efficiency and optimization are of the utmost importance. Processor integration requires careful design and verification processes to ensure reliability, as the processor becomes a fixed component of the chip. The library used is the UMC 65nm, which is made for a power-efficient and affordable solutions [13]. The tool suite used to create the ASIC is Design Compiler, a sophisticated synthesis tool from Synopsys that converts high-level RTL descriptions into optimized gate-level netlists. It performs area, timing, and power analysis to ensure that designs meet specified constraints.

As before, the design will be synthesized at its maximum operating frequency, found by modifying the clock until the slack is exactly zero. The achieved result is 637 MHz, as seen in the following timing report:

```
Point                                       Incr       Path
---------------------------------------------------------------
        ...                                 ...        ...
data arrival time                                      1.55
clock INPUT_CLK (rise edge)                 1.57       1.57
clock network delay (ideal)                 0.00       1.57
horcrux_wrapper_i/horcrux_top_i/i_horcrux
  /ascon_32out_reg[rd2][1]/CK (DFQBRM1RA)   0.00       1.57 r
library setup time                         -0.02       1.55
data required time                                     1.55
--------------------------------------------------------------
data required time                                     1.55
data arrival time                                     -1.55
--------------------------------------------------------------
slack (MET)                                            0.00
```

The following images represent the blocks that were not reported in the FPGA implementation before, like the top entity of the implemented design in figure 4.10a, where it is possible to see all signals sent and received to use the CV-X-IF interface. Figure 4.10b shows the controller and the block with the register and datapath. Lastly, in Figure 4.10c there is a more detailed view of the datapath.



(a) Top entity

(b) Internal structure

(c) Zoom of the datapath

Figure 4.10: Design implemented as ASIC

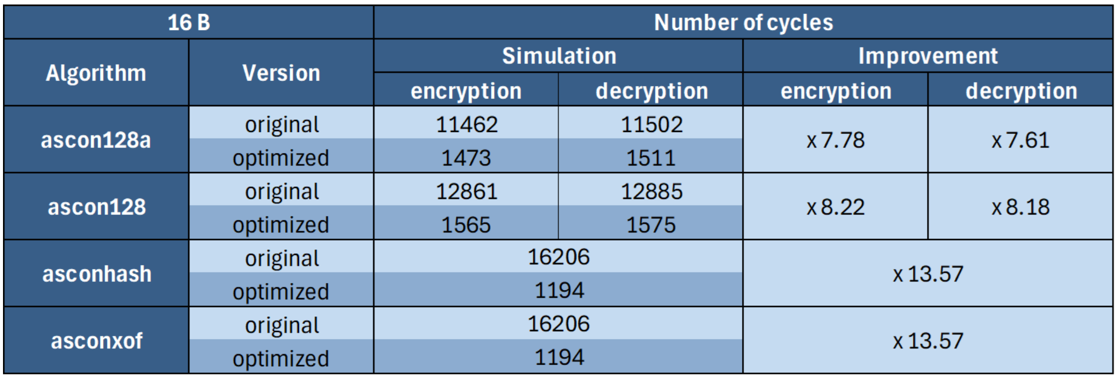
Chapter 5

Results and comparisons

In this chapter, the results obtained from the project's simulation, synthesis, and implementation phases are reviewed in detail and analyzed. The performance and resource utilization of the coprocessor will be examined to evaluate the effectiveness of the proposed design. While the coprocessor developed in this thesis represents a novel contribution to the field, with no directly comparable work available, an effort will be made to benchmark its results against similar designs found in the current literature. This comparison will underline the proposed approach's strengths and weaknesses, offering a comprehensive understanding of its advantages and potential limitations. Finally, the possibilities for future work will be laid out, including potential improvements, optimizations, and extensions of the current design.

5.1 Instruction Set Extension

The results derived from the instruction set extension performed in chapter 3 are reported in Table 5.1 in another arrangement, to give a clearer visualization and facilitate a comparative analysis with the other evaluations. The inputs processed by the algorithms in this case are 16 bytes long. This is an important parameter when evaluating the cycle count because a bigger message needs more permutations to be fully processed, meaning more cycles.

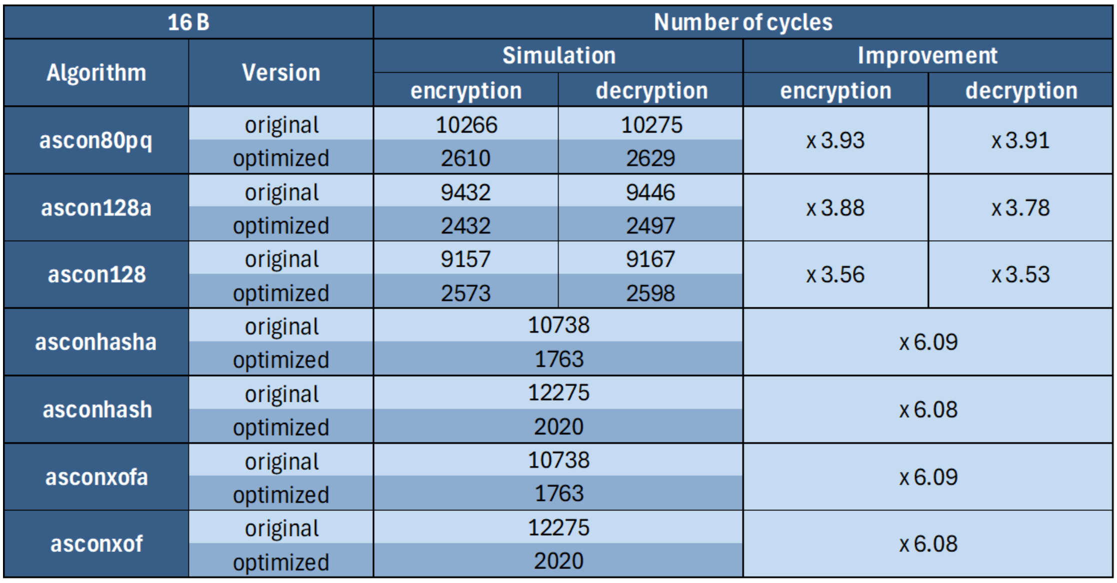


16 B		Number of cycles			
Algorithm	Version	Simulation		Improvement	
		encryption	decryption	encryption	decryption
ascon128a	original	11462	11502	x 7.78	x 7.61
	optimized	1473	1511		
ascon128	original	12861	12885	x 8.22	x 8.18
	optimized	1565	1575		
asconhash	original	16206		x 13.57	
	optimized	1194			
asconxof	original	16206		x 13.57	
	optimized	1194			

Table 5.1: Results of the instruction set extension

5.2 Simulation

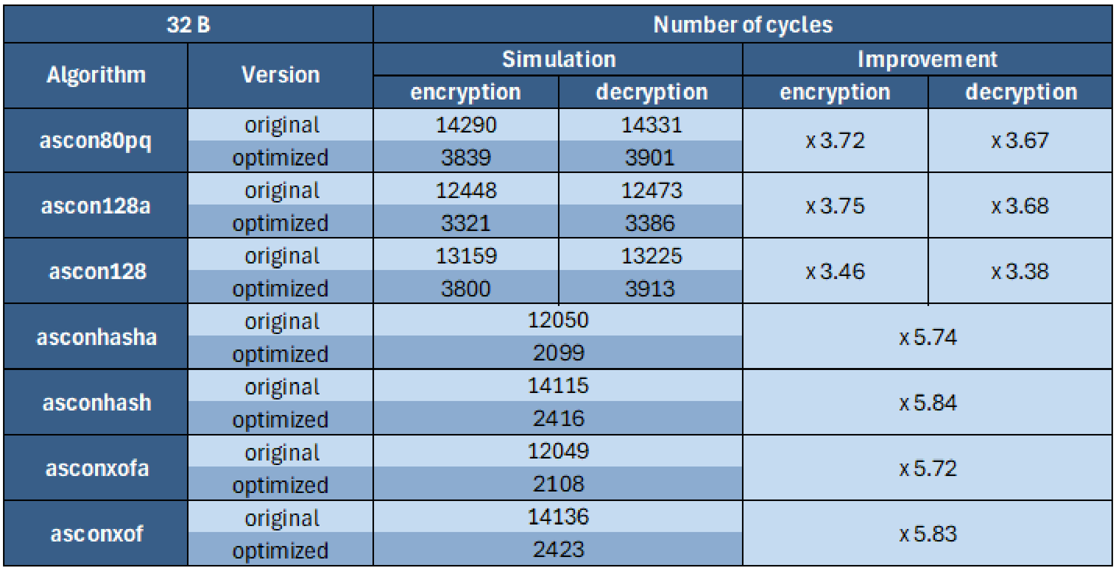
The simulation was carried out with QuestaSim. The testbench used is not the one mentioned before for the genkat file, but another that performs a single test. The results obtained applying an input of 16 and 32 bytes are reported in Table 5.2 and Table 5.3 respectively.



16 B		Number of cycles			
Algorithm	Version	Simulation		Improvement	
		encryption	decryption	encryption	decryption
ascon80pq	original	10266	10275	x 3.93	x 3.91
	optimized	2610	2629		
ascon128a	original	9432	9446	x 3.88	x 3.78
	optimized	2432	2497		
ascon128	original	9157	9167	x 3.56	x 3.53
	optimized	2573	2598		
asconhasha	original	10738		x 6.09	
	optimized	1763			
asconhash	original	12275		x 6.08	
	optimized	2020			
asconxofa	original	10738		x 6.09	
	optimized	1763			
asconxof	original	12275		x 6.08	
	optimized	2020			

Table 5.2: Results of the simulation, inputs of 16 B

These tables represent all the schemes tested for Ascon, with the number of cycles used to perform both the original and the accelerated algorithm. The first thing to notice is a confirmation of what has been said before: changing the length of



32 B		Number of cycles			
Algorithm	Version	Simulation		Improvement	
		encryption	decryption	encryption	decryption
ascon80pq	original	14290	14331	x 3.72	x 3.67
	optimized	3839	3901		
ascon128a	original	12448	12473	x 3.75	x 3.68
	optimized	3321	3386		
ascon128	original	13159	13225	x 3.46	x 3.38
	optimized	3800	3913		
asconhasha	original	12050		x 5.74	
	optimized	2099			
asconhash	original	14115		x 5.84	
	optimized	2416			
asconxofa	original	12049		x 5.72	
	optimized	2108			
asconxof	original	14136		x 5.83	
	optimized	2423			

Table 5.3: Results of the simulation, input of 32 B

the input modifies the cycle count. This is an important factor when comparing results because often other papers do not provide the size of the inputs, but only parameters normalized to this number. These values are not completely truthful because, as it is possible to observe from Tables 5.2 and 5.3, the ratio between the cycle count increase and the input data increment is not the same.
Another important aspect comes from the comparison of the results of chapter 3, in Table 5.1 and the ones of Table 5.2, where it is possible to observe the big difference in the improvements of the accelerator. The main reason for this change is not due to the performance of the coprocessor but caused by the CV-X-IF. When an ISE is performed, the computation of the operation is the same as every other instruction, with no overhead in the forwarding of data except the cycles for loading and storing the state vector to the register. When using this coprocessor many cycles are wasted for loading and storing data. This is especially the case for store operations, as just one 32-bit word can be written in the central register file per clock cycle. In addition, each time an instruction is dispatched, there is a **handshake protocol** to respect that increases the processing time. All of this translates in a **reduction of the speedup by a factor of 2**, which is again reduced when the inputs increase in size. This may appear as an unfortunate outcome, but comparing it to the work [14] that accelerates ascon128v1.2 with an ISE only to x2, it is a good achievement. Another paper that deals with an instruction set extension is [15], where their best result is to achieve a speedup of ×3.65 for ascon128v1.2, in line with the x3.56

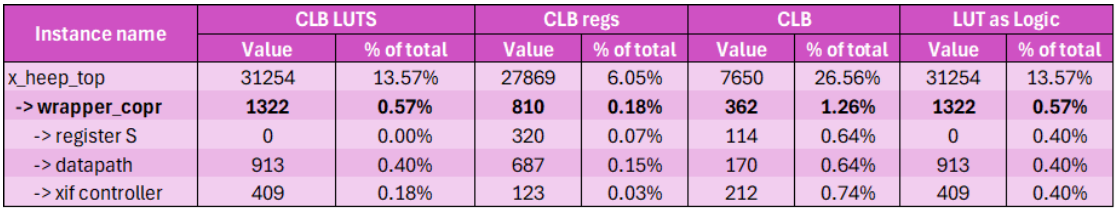
obtained in table 5.2. In this case is also important to remember that this work is more similar to the ISE of chapter 3 which still yields better results.

5.3 Synthesis

After confirming that both the software and the hardware parts work, the next step was to synthesize the processor, to evaluate other factors, like the occupied area and the operating frequency.

5.3.1 FPGA

The **results of the synthesis** and the successive **implementation** of the microprocessor are reported in Tables 5.4 and 5.5, and in Figures 5.1 and 5.2. The implementation results are the most important because they come from a physical mapping to the resources of the target FPGA, which means that many parameters, like the influence of the interconnections, are effectively taken into account. The results shown refer to a synthesis at **40 MHz** that is the maximum achievable frequency of this design.



Instance name	CLB LUTS		CLB regs		CLB		LUT as Logic	
	Value	% of total	Value	% of total	Value	% of total	Value	% of total
x_heep_top	31254	13.57%	27869	6.05%	7650	26.56%	31254	13.57%
-> wrapper_copr	**1322**	**0.57%**	**810**	**0.18%**	**362**	**1.26%**	**1322**	**0.57%**
-> register S	0	0.00%	320	0.07%	114	0.64%	0	0.40%
-> datapath	913	0.40%	687	0.15%	170	0.64%	913	0.40%
-> xif controller	409	0.18%	123	0.03%	212	0.74%	409	0.40%

Table 5.4: Utilization of the FPGA design implemented

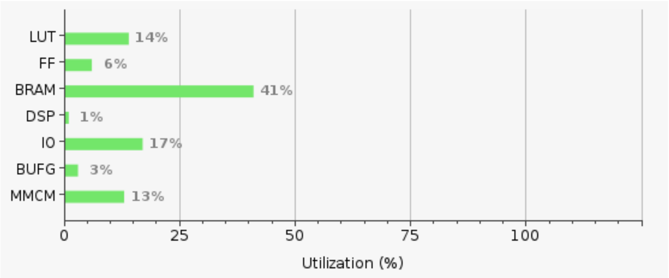
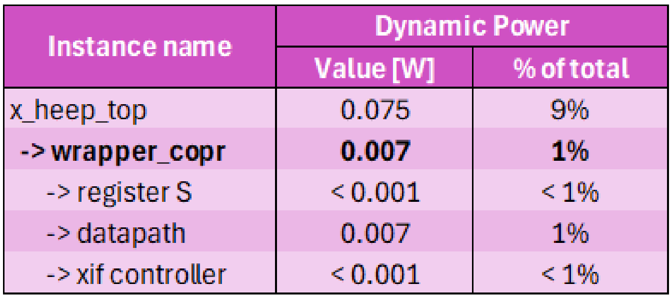


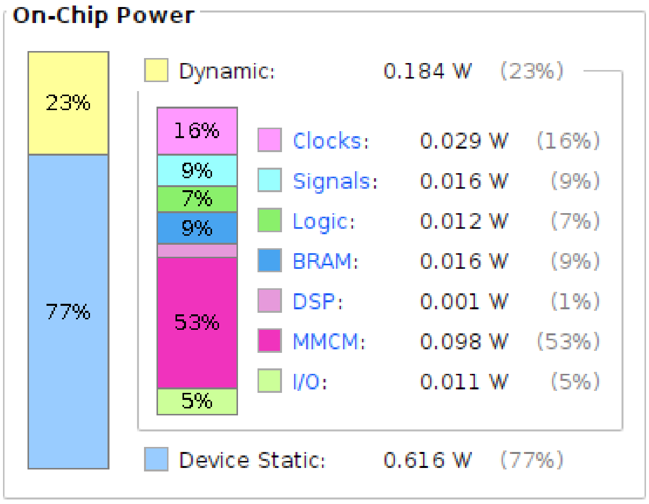
Figure 5.1: Comparative graph of the allocated units in the FPGA implementation

Table 5.4 and Figure 5.1 report the utilization of the available resources. The term CLB stands for Configurable Logic Block, a fundamental building element consisting of a collection of resources that implement combinatorial and sequential logic. The whole processor, denoted as *x_heep_top*, takes up just 26.56% of the CLB of the whole FPGA, while **the coprocessor uses only the 1.26% of the configurable logic blocks**. This demonstrates how little is the impact on the occupied area of the accelerator, also in terms of lookup tables (that takes the 0.57%) and registers (0.18%) usage.



Instance name	Dynamic Power	
	Value [W]	% of total
x_heep_top	0.075	9%
-> wrapper_copr	**0.007**	**1%**
-> register S	< 0.001	< 1%
-> datapath	0.007	1%
-> xif controller	< 0.001	< 1%

Table 5.5: Power usage of the FPGA design implemented



On-Chip Power

Dynamic: 0.184 W (23%)

Clocks: 0.029 W (16%)
Signals: 0.016 W (9%)
Logic: 0.012 W (7%)
BRAM: 0.016 W (9%)
DSP: 0.001 W (1%)
MMCM: 0.098 W (53%)
I/O: 0.011 W (5%)

Device Static: 0.616 W (77%)

Figure 5.2: Power distribution of the FPGA design implemented

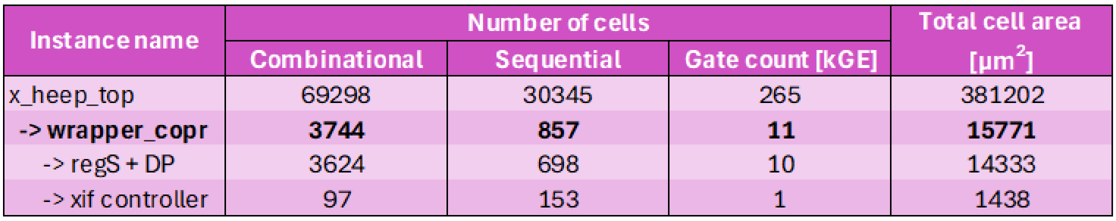
In Table 5.5 is reported the dynamic power consumption. The processor elements alone consume just 9% of the total dynamic power, the rest is distributed as in figure 5.2. It is clear that it is not possible to optimize further these numbers, but it is possible to notice that the **coprocessor is using just the 1% of the total power**.

From the implemented design, also the bit stream to be loaded in the physical board was generated and inserted in the Artix-7 FPGA, and then the various algorithms were tested. The trials were successful, and the same performance reported in Table 5.3 was achieved.

The low area and power consumption results demonstrate the effectiveness of this coprocessor in constrained environments, where the resources are limited and each addition to the hardware requires careful consideration of all parameters. If the achieved performance result is not satisfactory, it could be possible to switch the microprocessor to a faster one to achieve higher throughput. The coprocessor alone can reach up to 667 MHz, meaning that it could be used for much faster processors.

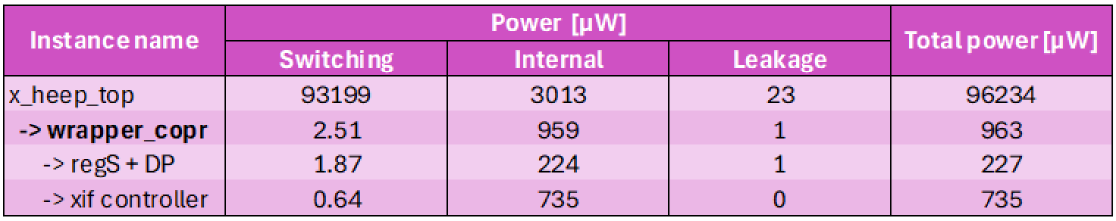
5.3.2 ASIC

The results obtained with the profiling of the synthesized ASIC microcontroller are summarized in Tables 5.6 and 5.7.



Instance name	Number of cells			Total cell area [µm^2]
	Combinational	Sequential	Gate count [kGE]	
x_heep_top	69298	30345	265	381202
-> wrapper_copr	**3744**	**857**	**11**	**15771**
-> regS + DP	3624	698	10	14333
-> xif controller	97	153	1	1438

Table 5.6: Area usage of the synthesized ASIC design



Instance name	Power [µW]			Total power [µW]
	Switching	Internal	Leakage	
x_heep_top	93199	3013	23	96234
-> wrapper_copr	2.51	959	1	963
-> regS + DP	1.87	224	1	227
-> xif controller	0.64	735	0	735

Table 5.7: Power usage of the synthesized ASIC design

Again, it is possible to determine that the coprocessor has a small contribution to the power and area usage, taking up the **5% of the total processor area** and the **1% of the power usage**. The processor was synthesized at the **maximum frequency of 637 MHz**.

It is difficult to compare the FPGA and ASIC results, as they deal with different kinds of cells. Three parameters that can be assessed are the percentage of the occupied area, the proportion of power usage, and the operating frequency. Although the different syntheses share the same values for the first two parameters, the third differs considerably, being 637 MHz for the ASIC and 40 MHz for the FPGA. This distinction is also partially because the FPGA value takes into consideration the non-idealities of the physical designs that extend the critical path. However, the magnitude of this discrepancy is too big and implies that the ASIC is the best choice, trading off the adaptability of the FPGA for much higher throughput.

5.4 Comparisons

To finally understand whether the results obtained are satisfactory, a comparison with other works present in the literature is of the utmost importance. To do so, the parameters that will be evaluated are the technology used, the maximum operating frequency, and the throughput. The last one is the main figure for the evaluation of the performance, and it is calculated as:

$$TP = \frac{InputBits}{Cycles} \cdot f_{max}$$

This parameter is often explicitly presented in the papers describing other works, if this is not the case, then this value was speculated using the formula above, only when all the involved variables are known.
Another parameter that is used to compare ASIC results is the **Gate Equivalent**. The area results of a synthesis depend on the technology used, making the absolute area useless for comparisons. For this reason, the occupied area is instead expressed as:

$$GateEquivalent = \frac{total\_area}{area\_NAND2}$$

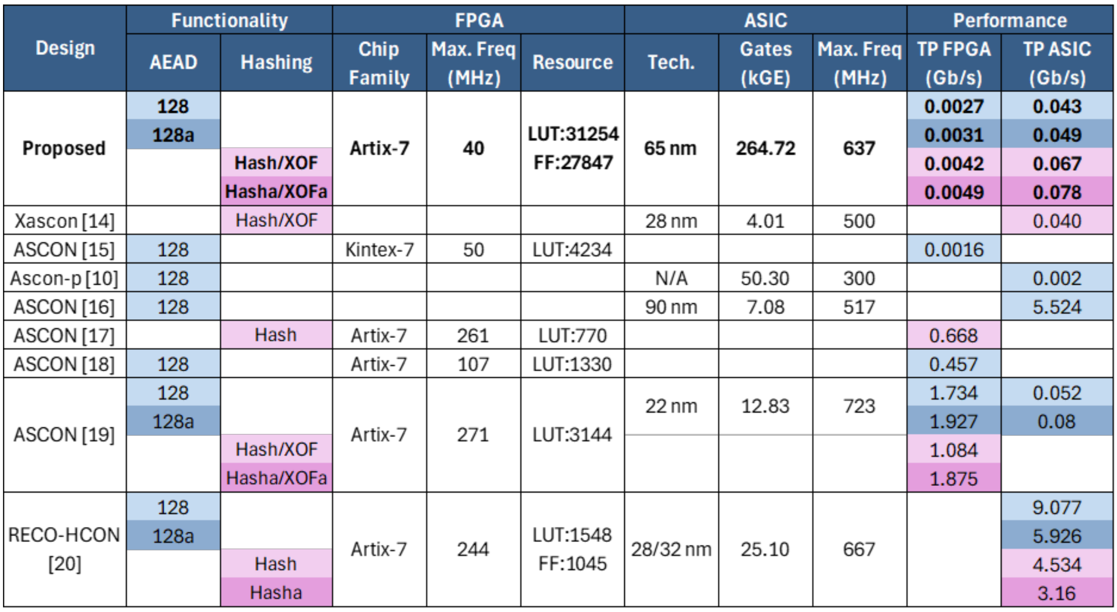
The NAND2 represents the average gate inside the integrated circuit and, for this reason, its area is used to normalize the total one. In this case, with the UMC 65 nm library, a single NAND2 occupies 1.44 μm^2.

Now that all parameters are clear, it is possible to summarize other works as in Table 5.8.

There is a great variety of proposed designs that can be divided into:

- **Instruction set extension of a RISC-V**. This is the case of [14], [15] and [10]. These designs are the closest to the implementation of this thesis, still there is a difference because in this case the ISA of the processor is effectively modified, while the proposed does not apply any change to the architecture.

  [14] aims to explore hardware acceleration through Instruction Set Extensions in a low-end 32-bit RISC-V core (Ibex). It implements four different parametrizations of Kyber symmetric primitives, one of which uses Ascon (Kyber-Ascon), creating the extension Xascon. The Ibex core and Xascon were synthesized in ASIC with a 28 nm library. The area result is really low, because the Ibex RISC-V is made for embedded and IoT applications,



Design	Functionality		FPGA			ASIC			Performance	
	AEAD	Hashing	Chip Family	Max. Freq (MHz)	Resource	Tech.	Gates (kGE)	Max. Freq (MHz)	TP FPGA (Gb/s)	TP ASIC (Gb/s)
Proposed	128		Artix-7	40	LUT:31254 FF:27847	65 nm	264.72	637	0.0027	0.043
	128a								0.0031	0.049
		Hash/XOF							0.0042	0.067
		Hasha/XOFa							0.0049	0.078
Xascon [14]		Hash/XOF				28 nm	4.01	500		0.040
ASCON [15]	128		Kintex-7	50	LUT:4234				0.0016	
Ascon-p [10]	128					N/A	50.30	300		0.002
ASCON [16]	128					90 nm	7.08	517		5.524
ASCON [17]		Hash	Artix-7	261	LUT:770				0.668	
ASCON [18]	128		Artix-7	107	LUT:1330				0.457	
ASCON [19]	128		Artix-7	271	LUT:3144	22 nm	12.83	723	1.734	0.052
	128a								1.927	0.08
		Hash/XOF							1.084	
		Hasha/XOFa							1.875	
RECO-HCON [20]	128		Artix-7	244	LUT:1548 FF:1045	28/32 nm	25.10	667		9.077
	128a									5.926
		Hash								4.534
		Hasha								3.16

Table 5.8: Comparison with other accelerators

but the throughput of 0.04 Gbps is lower than the proposed 0.067 Gbps. The results obtained by the design is similar, the main difference is the area occupation, but this factor depends more on the chosen microprocessor than the implemented design.

Cheng et al. [15] propose ISEs for the ten submitted algorithms of the final round LWC competition. The implementations evaluated were: software only, Zbkb/x, and Zbkb/x+ ISE. The last one shows 2.28× and 1.18× performance gains compared to the other two. The metrics are evaluated in FPGA, with the extended Rocket RV32GC processor. The maximum operating frequency is 50 MHz and the occupied area of the core and accelerator, in terms of LUT, is 4234. The difference between the base core and the accelerated version is 1180 LUTs, a result close to the 1322 LUTs of the proposed accelerator. Also, the throughput determines a similar performance, although the designed coprocessor is faster, despite the additional cycles dedicated to the handshake protocols.

[10] is an interesting work that achieves a speedup of 97% on the clock cycles needed to perform the ascon128 algorithm. The implemented extension, called ASCON-p, performs a hardware acceleration similar to the proposed one, but modifies the approach of the state vector storage. Instead of adding a dedicated

register that occupies an area and many cycles to charge data, [10] reuses the internal register file of the processor, dedicating some register exclusively to the state vector during the execution of the round transformation, modifying the register file substantially. This is done by extending to ten the number of input and output ports and adding 500 GE of muxes that toggle the input data from the usual input to the result of the previous transformation. The area value presented pertains to the RI5CY core, while the coprocessor alone occupies 4.7 kGE, less than the proposed 10 kGE because the additional register was not introduced. The throughput of 0.002 Gbps is still inferior to the proposed 0.043 Gbps.

- **Standalone implementations, single instance**. This is the case of [16], [17], and [18], where a single algorithm of the Ascon family is chosen and entirely implemented in the accelerator. These works lack the flexibility of the design implemented in this thesis, trading off this parameter for higher performances. The area and throughput results seem better than the proposed because they do not account for the overhead of moving data between the main core and the crypto-processor, which affects the overall efficiency.

  [16] implements a variety of loosely coupled accelerators, each specialized for an application, like low area and power for RFID, high throughput designs and wireless sensor nodes. Furthermore, they demonstrate how the design can be protected against first-order differential power analysis (DPA) attacks. As mentioned before, this work is different from the proposed one and provides higher throughput, while occupying just 7 kGE, which is somewhat comparable to the 10 kGE of the designed coprocessor alone.

  [17] works on just the hash function, proposing unrolling to execute several rounds in one clock cycle. After an evaluation of the design space, they concluded that computing four rounds per clock cycle achieves the maximum throughput. Performing more would lead to a higher critical path that would hinder the attained performance, limiting the maximum operating frequency. The area utilization of [17] is of 770 LUTs, comparing it with the proposed one of 1322 LUTs, there is a difference due to the addition of the eXtension interface logic. The frequency result has a significant impact on the throughput, which is higher because just the standalone component was considered. For comparison, if the proposed coprocessor had been implemented as standalone, the maximum operating frequency would have exceeded [17], reaching 667 MHz and resulting in much higher throughput.

[18] proposes an architecture for the ascon128 scheme, whose main objective is to implement a lightweight design for IoMT (internet of medical things) devices based on a single round transformation per clock cycle. The area utilization result is again really close to the proposed coprocessor alone, while the achieved throughput is higher.

- **Standalone implementations, reconfigurable**. This is the case of [19] and [20], which are standalone implementations that include interfaces and connections in the design. They are also able to implement more algorithms in the same accelerator. As before, the performances of these works are higher than the proposed because they do not involve the implementation of the processor and the overhead of performing some operations in software. They also do not account for the overhead involved in transferring data between the main core and the coprocessor.

  [19] presents one of the first coprocessors that realizes all ASCON main functionalities. The processor is self-contained, and it includes the AMBA AHB (High-performance Bus) and APB (Advanced Peripheral Bus) interfaces, two asynchronous FIFOs, a Register File, a Control and Interrupt Generator block to be integrated into a SoC. This work was synthesized both in FPGA and ASIC technologies. The FPGA implementation yields better results than the proposed ones, but again the causes all relate to the fact that the area is greatly increased by the whole structure of the microcontroller not present in [19] as it is a standalone coprocessor, while the throughput of the proposed is heavily limited by the processor and not by the accelerator. On the other hand, the ASIC presents results close to the proposed ones.

  [20] is a compact processor that is reconfigurable, to support six different algorithms of the Ascon family. Its main goal is to achieve the maximum throughput possible while still being compact, and therefore be embedded as an IP for SoCs. The reconfigurable setup uses the same sponges interface, and the difference between instances is handled with padding FIFOs, a variable permutation, and an XOR operand selection & shift stage. Although the architecture was implemented also in FPGA, only the throughput of the ASIC implementation is reported in [20]. As said before, the main concern for this design is high performance, in fact, it is the highest among the works presented in Table 5.8, but at the expense of the area that is the highest of the works similar to it.

In the end, the comparison with other works determined that, though the operations carried out are similar, there is a big difference between standalone and accelerators coupled to the processor. The detached architectures inevitably have higher throughput because they do not take into account the overheads involved in the connection to another system. On the other hand, compared to more similar works, the proposed coprocessor is faster despite having additional costs deriving from the exchange of data. However, it occupies more area because it has to accommodate extra hardware to perform all the operations that, in a tightly coupled accelerator, would have been done by the core of the processor.

5.5 Future work

The work proposed until now is just a glimpse of the possibilities that lie ahead. First of all, in November 2024 the new version of Ascon (v 1.3) was released and a good starting point for future work would include the analysis of the improved version. Another opportunity would be to evaluate the performances of also the MAC (message authentication code) and PRF (pseudorandom function) algorithms of the Ascon family, as they are one of the few alternatives in the lightweight cryptography world.

For what concerns the coprocessor, another approach could be to effectively implement the unrolling proposed to speed up the round execution or to act on the interface. This can be done either by switching to another one, or by utilizing the full potential of the CV-X-IF, using more advanced functions that allow loading and storing data in fewer operations.

In terms of security, an analysis of the robustness of side-channel attacks is essential to understand whether the coprocessor has some criticalities and weak points to breaches. In the case they are present, future work should focus more on this aspect, implementing strategies to improve the security of the whole system.

Chapter 6

Conclusion

This thesis presents one of the few coprocessors for the instruction set extension of a RISC-V made to accelerate the main algorithms of the Ascon family. Starting from the exploration of the design space, aimed to find the best tradeoff between performance improvement and resource utilization, then physically implementing the design and testing it in a real application.

The results demonstrate the proposed coprocessor's efficacy in reducing computation time for ASCON operations, accelerating by 3.6× and 6× the AEAD and hash schemes respectively, achieving higher throughput than other similar works. However, these enhancements come at the expense of increased hardware area due to the additional functionalities integrated into the design. Despite these tradeoffs, the coprocessor achieves a remarkable balance between speed and resource requirements, making it highly suitable for deployment in resource-constrained environments.

There are still many possibilities for future enhancements, including the exploration of other Ascon family algorithms and additional hardware optimizations. There is still more work to be performed on the security evaluation, that is currently being assessed.

The results of this study contribute positively to the fast-evolving field of lightweight cryptography, offering an effective strategy to cope with the need to conduct cryptographic processes with good performance and high flexibility, addressing the security concerns of resource-limited devices.

Appendix A

Source Code

```c
#include "round.h"

void ROUND(ascon_state_t* s, uint8_t C) {
  uint64_t xtemp;
  /* round constant */
  s->x[2] ^= C;
  /* s-box layer */
  s->x[0] ^= s->x[4];
  s->x[4] ^= s->x[3];
  s->x[2] ^= s->x[1];

  xtemp = s->x[0] & ~s->x[4];
  s->x[0] ^= s->x[2] & ~s->x[1];
  s->x[2] ^= s->x[4] & ~s->x[3];
  s->x[4] ^= s->x[1] & ~s->x[0];
  s->x[1] ^= s->x[3] & ~s->x[2];
  s->x[3] ^= xtemp;
  s->x[1] ^= s->x[0];
  s->x[3] ^= s->x[2];
  s->x[0] ^= s->x[4];
  s->x[2] = ~s->x[2];
  /* linear layer */
  s->x[0] ^=
      (s->x[0] >> 19) ^ (s->x[0] << 45) ^ (s->x[0] >> 28) ^ (s->x[0] << 36);
  s->x[1] ^=
      (s->x[1] >> 61) ^ (s->x[1] << 3) ^ (s->x[1] >> 39) ^ (s->x[1] << 25);
  s->x[2] ^=
      (s->x[2] >> 1) ^ (s->x[2] << 63) ^ (s->x[2] >> 6) ^ (s->x[2] << 58);
  s->x[3] ^=
      (s->x[3] >> 10) ^ (s->x[3] << 54) ^ (s->x[3] >> 17) ^ (s->x[3] << 47);
  s->x[4] ^=
      (s->x[4] >> 7) ^ (s->x[4] << 57) ^ (s->x[4] >> 41) ^ (s->x[4] << 23);
  printstate(" round output", s);
}

 void PROUNDS(ascon_state_t* s, int nr) {
  int i = START(nr);
  do {
    ROUND(s, RC(i));
    i += INC;
  } while (i != END);
}
```

Code A.1: round.c before ISE

```
#include "round.h"

void ROUND(ascon_state_t* s, int i) {
  ascon64_t x0, x1, x2, x3, x4;

  ascon64_t* state1 = (ascon64_t*) chess_copy(s->x);

        x0 = state1[0];
        x1 = state1[1];
        x2 = state1[2];
        x3 = state1[3];
        x4 = state1[4];

          round( i, x0,x1,x2,x3,x4,
                                x0,x1,x2,x3,x4);

        state1[0] = x0;
        state1[1] = x1;
        state1[2] = x2;
        state1[3] = x3;
        state1[4] = x4;

        }
}

 void PROUNDS(ascon_state_t* s, int nr) {
  int i = START(nr);
  do {
    ROUND(s, i);
    i += INC;
  } while (i != END);
}
```

Code A.2: round.c during ISE

```
#include "round.h"
```

```
 void PROUNDS(ascon_state_t* s, int nr) {
  int i = START(nr);
      ascon64_t x0, x1, x2, x3, x4;
    ascon64_t* state1 = (ascon64_t*) chess_copy(s->x);

        x0 = state1[0];
        x1 = state1[1];
        x2 = state1[2];
        x3 = state1[3];
        x4 = state1[4];
  do {
          round( RC(i), x0,x1,x2,x3,x4,
                                x0,x1,x2,x3,x4);
    i += INC;
  } while (i != END);

        state1[0] = x0;
        state1[1] = x1;
        state1[2] = x2;
        state1[3] = x3;
        state1[4] = x4;
}
```

Code A.3: round.c after ISE

```
#ifndef ROUND_H_
#define ROUND_H_

#include "ascon.h"
#include "printstate.h"

 static inline void ROUND(state_t* s, int nr) {
        uint32_t x_low;
    uint32_t x_high;

        asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
        ↪   [rs1] "r" (s->x[0] ), [rs2] "r" ((s->x[0] >> 32) ), [i] "r" (0): );
    asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    ↪   [rs1] "r" (s->x[1] ), [rs2] "r" ((s->x[1] >> 32) ), [i] "r" (2): );
    asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    ↪   [rs1] "r" (s->x[2] ), [rs2] "r" ((s->x[2] >> 32) ), [i] "r" (4): );
    asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    ↪   [rs1] "r" (s->x[3] ), [rs2] "r" ((s->x[3] >> 32) ), [i] "r" (6): );
    asm volatile (".insn r 0x4b, 0x004, 0, x0, %[rs1], %[rs2], %[i]\r\n": :
    ↪   [rs1] "r" (s->x[4] ), [rs2] "r" ((s->x[4] >> 32) ), [i] "r" (8): );

    asm volatile (".insn r 0x4b, 0x004, 1, x0, %[rs1], %[rs2], x0\n\t" : :
    ↪   [rs1] "r" (1) , [rs2] "r" (nr): );
```

```
        for(int i=0;i<nr;i++){
                asm volatile ("nop");
        }


        for(int i=0; i<10; i+=2){
                asm volatile (".insn r 0x4b, 0x004, 2, %[rd_low], %[rs1], x0,
                ↪   x0\r\n": [rd_low] "=r" (x_low) : [rs1] "r" (i): );
                asm volatile (".insn r 0x4b, 0x004, 2, %[rd_high], %[rs1], x0,
                ↪   x0\r\n": [rd_high] "=r" (x_high): [rs1] "r" (i+1): );
                s->x[i/2] = ((uint64_t)x_high << 32) | x_low;
        }

}

#endif /* ROUND_H_ */
```

Code A.4: round.h for the coprocessor

Appendix B

Reports

```
Storage profiling information for ::iss generated by Checkers U-2022.12#33f3808fcb#221128 on Thu Aug 29 18:37:17 2024

Program being simulated:

    /home/thesis/federica.bader/Desktop/AEAD_accelerator/step2_primitives/Release/Reference

    Total cycle count           :                25746
    Report cycle count          :                25746
    Total instruction count     :                24810
    Report instruction count    :                24810
    Total size in program memory:                 7320

Command used to generate this report: ::iss profile storage_access_save -file
"/home/thesis/federica.bader/Desktop/AEAD_accelerator/step2_primitives/storage_access_report_start.txt" -function_details Off
-field_details Off -function_summary Off -data "" -cycle_count 0 -instruction_count 0 -hide_instruction_bits Off


Function storage access summary:

    Storage               Read-count  Read-count   Read-count Write-count Write-count  Write-count Function name
                             (total)  (function) (% of total)     (total)  (function) (% of total)
    -------------------- ----------- ----------- ------------ ----------- ----------- ------------
   --------------------------------------------------
    DMb                        29464       28080       95.30%       17000       15552       91.48% ROUND _Z5ROUNDP13ascon_state_th
                                             256        0.87%                     216        1.27% ascon_adata
                                             _Z11ascon_adataP13ascon_state_tPKhy
                                              44        0.15%                      32        0.19% ascon_aead_decrypt
                                              _Z18ascon_aead_decryptPhPKhS1_y
                                              44        0.15%                      32        0.19% ascon_aead_encrypt
                                              _Z18ascon_aead_encryptPhS_PKhyS
                                             148        0.50%                     148        0.87% ascon_decrypt
                                             _Z13ascon_decryptP13ascon_state_tPhP
                                             132        0.45%                     112        0.66% ascon_encrypt
                                             _Z13ascon_encryptP13ascon_state_tPhP
                                             152        0.52%                      88        0.52% ascon_final
                                             _Z11ascon_finalP13ascon_state_tPK11asc
                                              32        0.11%                      32        0.19% ascon_gettag
                                              _Z12ascon_gettagP13ascon_state_tPh
                                             176        0.60%                     192        1.13% ascon_initaead
                                             _Z14ascon_initaeadP13ascon_state_tP
                                              72        0.24%                     104        0.61% ascon_loadkey
                                              _Z13ascon_loadkeyP11ascon_key_tPKh
                                              64        0.22%                      52        0.31% ascon_verify
                                              _Z12ascon_verifyP13ascon_state_tPKh
                                              20        0.07%                      24        0.14% crypto_aead_decrypt
                                              _Z19crypto_aead_decryptPhPyS_P
                                              20        0.07%                      24        0.14% crypto_aead_encrypt
                                              _Z19crypto_aead_encryptPhPyPKh
                                              20        0.07%                     180        1.06% main _main
                                             192        0.65%                     192        1.13% memcpy memcpy
                                               4        0.01%                       0        0.00% printf printf
```

```
                                           8        0.03%                      20        0.12% vfprintf vfprintf
PMb                       203640         128        0.06%           0           0        0.00% P _Z1PP13ascon_state_ti
                                        5856        2.88%                       0        0.00% PROUNDS
                                        _Z7PROUNDSP13ascon_state_ti
                                      178008       87.41%                       0        0.00% ROUND _Z5ROUNDP13ascon_state_th
                                          48        0.02%                       0        0.00% _main_init _main_init
                                          40        0.02%                       0        0.00% _start_basic _start_basic
                                        2400        1.18%                       0        0.00% ascon_adata
                                        _Z11ascon_adataP13ascon_state_tPKhy
                                         280        0.14%                       0        0.00% ascon_aead_decrypt
                                         _Z18ascon_aead_decryptPhPKhS1_y
                                         288        0.14%                       0        0.00% ascon_aead_encrypt
                                         _Z18ascon_aead_encryptPhS_PKhyS
                                        1704        0.84%                       0        0.00% ascon_decrypt
                                        _Z13ascon_decryptP13ascon_state_tPhP
                                        1680        0.82%                       0        0.00% ascon_encrypt
                                        _Z13ascon_encryptP13ascon_state_tPhP
                                         512        0.25%                       0        0.00% ascon_final
                                         _Z11ascon_finalP13ascon_state_tPK11asc
                                         664        0.33%                       0        0.00% ascon_gettag
                                         _Z12ascon_gettagP13ascon_state_tPh
                                        1744        0.86%                       0        0.00% ascon_initaead
                                        _Z14ascon_initaeadP13ascon_state_tP
                                        1392        0.68%                       0        0.00% ascon_loadkey
                                        _Z13ascon_loadkeyP11ascon_key_tPKh
                                         848        0.42%                       0        0.00% ascon_verify
                                         _Z12ascon_verifyP13ascon_state_tPKh
                                          40        0.02%                       0        0.00% clib_hosted_io clib_hosted_io
                                         176        0.09%                       0        0.00% crypto_aead_decrypt
                                         _Z19crypto_aead_decryptPhPyS_P
                                         112        0.05%                       0        0.00% crypto_aead_encrypt
                                         _Z19crypto_aead_encryptPhPyPKh
                                        1256        0.62%                       0        0.00% main _main
                                        5376        2.64%                       0        0.00% memcpy memcpy
                                          40        0.02%                       0        0.00% printf printf
                                        1048        0.51%                       0        0.00% vfprintf vfprintf
X                          71453          14        0.02%       33553           0        0.00% P _Z1PP13ascon_state_ti
                                        1088        1.52%                     544        1.62% PROUNDS
                                        _Z7PROUNDSP13ascon_state_ti
                                       63828       89.33%                   30780       91.74% ROUND _Z5ROUNDP13ascon_state_th
                                          15        0.02%                       2        0.01% _main_init _main_init
                                           5        0.01%                       1        0.00% _start_basic _start_basic
                                         724        1.01%                     294        0.88% ascon_adata
                                         _Z11ascon_adataP13ascon_state_tPKhy
                                          68        0.10%                      28        0.08% ascon_aead_decrypt
                                          _Z18ascon_aead_decryptPhPKhS1_y
                                          68        0.10%                      29        0.09% ascon_aead_encrypt
                                          _Z18ascon_aead_encryptPhS_PKhyS
                                         538        0.75%                     210        0.63% ascon_decrypt
                                         _Z13ascon_decryptP13ascon_state_tPhP
                                         514        0.72%                     205        0.61% ascon_encrypt
                                         _Z13ascon_encryptP13ascon_state_tPhP
                                         186        0.26%                      96        0.29% ascon_final
                                         _Z11ascon_finalP13ascon_state_tPK11asc
                                         188        0.26%                      86        0.26% ascon_gettag
                                         _Z12ascon_gettagP13ascon_state_tPh
                                         558        0.78%                     250        0.75% ascon_initaead
                                         _Z14ascon_initaeadP13ascon_state_tP
                                         390        0.55%                     174        0.52% ascon_loadkey
                                         _Z13ascon_loadkeyP11ascon_key_tPKh
                                         262        0.37%                     121        0.36% ascon_verify
                                         _Z12ascon_verifyP13ascon_state_tPKh
                                           2        0.00%                       0        0.00% clib_hosted_io clib_hosted_io
                                          52        0.07%                      17        0.05% crypto_aead_decrypt
                                          _Z19crypto_aead_decryptPhPyS_P
                                          40        0.06%                      14        0.04% crypto_aead_encrypt
                                          _Z19crypto_aead_encryptPhPyPKh
                                         538        0.75%                      66        0.20% main _main
                                        2352        3.29%                     624        1.86% memcpy memcpy
                                           3        0.00%                       3        0.01% printf printf
                                          20        0.03%                       9        0.03% vfprintf vfprintf
```

Report B.1: Storage access report of the starting algorithm of Ascon

```
Function profiling report information for ::iss generated by Checkers U-2022.12#33f3808fcb#221128 on Thu Aug 29 18:36:58 2024


Program being simulated:

    /home/thesis/federica.bader/Desktop/AEAD_accelerator/step2_primitives/Release/Reference

    Total cycle count          :                25746
    Report cycle count         :                25738
    Total instruction count    :                24810
    Total size in program memory:                7320

Command used to generate this report: ::iss profile save "/home/thesis/federica.bader/Desktop/AEAD_accelerator/step2_primitives/function_report_start.txt" -type function_profiling -xml
0 -function_details 0 -one_file 1 -call_details Off -call_tree Off -data "" -cycle_count 0 -instruction_count 0

          Calls  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max     Low PC    High PC     Function
                     (func)      (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    -------------------------------------------------
              1           8        0.03%           8           8           8       25738      100.00%       25738       25738       25738          0         23 _main_init
              1         158        0.61%         158         158         158       25730       99.97%       25730       25730       25730       5392       6235 main
             14          28        0.11%           2           2           2       23346       90.71%        1299        1667        2589         36         39 P
             14         854        3.32%          49          61          91       23318       90.60%        1297        1665        2587       7192       7247 PROUNDS
            108       22464       87.28%         208         208         208       22464       87.28%         208         208         208       6236       7187 ROUND
              1          25        0.10%          25          25          25       12796       49.72%       12796       12796       12796       5264       5387 crypto_aead_decrypt
              1          36        0.14%          36          36          36       12771       49.62%       12771       12771       12771       5008       5171 ascon_aead_decrypt
              1          17        0.07%          17          17          17       12759       49.57%       12759       12759       12759       5176       5259 crypto_aead_encrypt
              1          36        0.14%          36          36          36       12742       49.51%       12742       12742       12742       4840       5007 ascon_aead_encrypt
              2         304        1.18%         152         152         152        8218       31.93%        4109        4109        4109       1000       1707 ascon_adata
              2         220        0.85%         110         110         110        5518       21.44%        2759        2759        2759        448        995 ascon_initaead
              2          68        0.26%          34          34          34        5246       20.38%        2623        2623        2623       3728       3919 ascon_final
              1         216        0.84%         216         216         216        2934       11.40%        2934        2934        2934       2648       3723 ascon_decrypt
              1         213        0.83%         213         213         213        2931       11.39%        2931        2931        2931       1712       2643 ascon_encrypt
             24         720        2.80%          30          30          30         720        2.80%          30          30          30       7328       7359 memcpy
              2         164        0.64%          82          82          82         284        1.10%         142         142         142         40        443 ascon_loadkey
              1         108        0.42%         108         108         108         168        0.65%         168         168         168       4312       4839 ascon_verify
              1          82        0.32%          82          82          82         142        0.55%         142         142         142       3920       4307 ascon_gettag
              1           3        0.01%           3           3           3          17        0.07%          17          17          17       7308       7323 printf
              1          11        0.04%          11          11          11          14        0.05%          14          14          14       7248       7307 vfprintf
              1           3        0.01%           3           3           3           3        0.01%           3           3           3        444        447 clib_hosted_io
              0           0        0.00%           -           -           -           0        0.00%           -           -           -         24         35 _start_basic

          Calls  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max     Low PC    High PC     Function
                     (func)      (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    -------------------------------------------------
            108       22464       87.28%         208         208         208       22464       87.28%         208         208         208       6236       7187 ROUND
             14         854        3.32%          49          61          91       23318       90.60%        1297        1665        2587       7192       7247 PROUNDS
             24         720        2.80%          30          30          30         720        2.80%          30          30          30       7328       7359 memcpy
              2         304        1.18%         152         152         152        8218       31.93%        4109        4109        4109       1000       1707 ascon_adata
              2         220        0.85%         110         110         110        5518       21.44%        2759        2759        2759        448        995 ascon_initaead
              1         216        0.84%         216         216         216        2934       11.40%        2934        2934        2934       2648       3723 ascon_decrypt
              1         213        0.83%         213         213         213        2931       11.39%        2931        2931        2931       1712       2643 ascon_encrypt
              2         164        0.64%          82          82          82         284        1.10%         142         142         142         40        443 ascon_loadkey
              1         158        0.61%         158         158         158       25730       99.97%       25730       25730       25730       5392       6235 main
              1         108        0.42%         108         108         108         168        0.65%         168         168         168       4312       4839 ascon_verify
```

```
 1    82   0.32%   82   82   82    142   0.55%    142    142    142  3920  4307 ascon_gettag
 2    68   0.26%   34   34   34   5246  20.38%   2623   2623   2623  3728  3919 ascon_final
 1    36   0.14%   36   36   36  12771  49.62%  12771  12771  12771  5008  5171 ascon_aead_decrypt
 1    36   0.14%   36   36   36  12742  49.51%  12742  12742  12742  4840  5007 ascon_aead_encrypt
14    28   0.11%    2    2    2  23346  90.71%   1299   1667   2589    36    39 P
 1    25   0.10%   25   25   25  12796  49.72%  12796  12796  12796  5264  5387 crypto_aead_decrypt
 1    17   0.07%   17   17   17  12759  49.57%  12759  12759  12759  5176  5259 crypto_aead_encrypt
 1    11   0.04%   11   11   11     14   0.05%     14     14     14  7248  7307 vfprintf
 1     8   0.03%    8    8    8  25738 100.00%  25738  25738  25738     0    23 _main_init
 1     3   0.01%    3    3    3      3   0.01%      3      3      3   444   447 clib_hosted_io
 1     3   0.01%    3    3    3     17   0.07%     17     17     17  7308  7323 printf
 0     0   0.00%    -    -    -      0   0.00%      -      -      -    24    35 _start_basic
```

Report B.2: Function report of the starting algorithm of Ascon

```
Function profiling report information for ::iss generated by Checkers U-2022.12#33f3808fcb#221128 on Sun Sep 15 20:10:26 2024


Program being simulated:

    /home/thesis/federica.bader/Desktop/AEAD_accelerator/step3_improvingRound/Release/Improved_Round

    Total cycle count          :                3140
    Report cycle count         :                3132
    Total instruction count    :                2588
    Total size in program memory:               7468

Command used to generate this report: ::iss profile save "/home/thesis/federica.bader/Desktop/AEAD_accelerator/step3_improvingRound/function_report_ImprovedRound.txt" -type
function_profiling -xml 0 -function_details 0 -one_file 1 -call_details Off -call_tree Off -data "" -cycle_count 0 -instruction_count 0

          Calls  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max      Low PC     High PC     Function
                     (func)      (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    -------------------------------------------------
              1           8        0.26%           8           8           8        3132      100.00%        3132        3132        3132           0          23 _main_init
              1         158        5.04%         158         158         158        3124       99.74%        3124        3124        3124        6416        7259 main _main
              1          25        0.80%          25          25          25        1494       47.70%        1494        1494        1494        6288        6411 crypto_aead_decrypt
              1         664       21.20%         664         664         664        1469       46.90%        1469        1469        1469        2968        6195 ascon_aead_decrypt
              1          17        0.54%          17          17          17        1455       46.46%        1455        1455        1455        6200        6283 crypto_aead_encrypt
              1         633       20.21%         633         633         633        1438       45.91%        1438        1438        1438          40        2967 ascon_aead_encrypt
             14         890       28.42%          55          63          85         890       28.42%          55          63          85        7264        7375 PROUNDS
             24         720       22.99%          30          30          30         720       22.99%          30          30          30        7456        7487 memcpy memcpy
              1           3        0.10%           3           3           3          17        0.54%          17          17          17        7436        7451 printf printf
              1          11        0.35%          11          11          11          14        0.45%          14          14          14        7376        7435 vfprintf vfprintf
              1           3        0.10%           3           3           3           3        0.10%           3           3           3          36          39 clib_hosted_io
              0           0        0.00%           -           -           -           0        0.00%           -           -           -          24          35 _start_basic

          Calls  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max      Low PC     High PC     Function
                     (func)      (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    -------------------------------------------------
             14         890       28.42%          55          63          85         890       28.42%          55          63          85        7264        7375 PROUNDS
             24         720       22.99%          30          30          30         720       22.99%          30          30          30        7456        7487 memcpy memcpy
              1         664       21.20%         664         664         664        1469       46.90%        1469        1469        1469        2968        6195 ascon_aead_decrypt
              1         633       20.21%         633         633         633        1438       45.91%        1438        1438        1438          40        2967 ascon_aead_encrypt
              1         158        5.04%         158         158         158        3124       99.74%        3124        3124        3124        6416        7259 main _main
              1          25        0.80%          25          25          25        1494       47.70%        1494        1494        1494        6288        6411 crypto_aead_decrypt
              1          17        0.54%          17          17          17        1455       46.46%        1455        1455        1455        6200        6283 crypto_aead_encrypt
              1          11        0.35%          11          11          11          14        0.45%          14          14          14        7376        7435 vfprintf vfprintf
              1           8        0.26%           8           8           8        3132      100.00%        3132        3132        3132           0          23 _main_init
              1           3        0.10%           3           3           3           3        0.10%           3           3           3          36          39 clib_hosted_io
              1           3        0.10%           3           3           3          17        0.54%          17          17          17        7436        7451 printf printf
              0           0        0.00%           -           -           -           0        0.00%           -           -           -          24          35 _start_basic
```

Report B.3: Function report of the accelerated algorithm Ascon

```
Storage profiling information for ::iss generated by Checkers U-2022.12#33f3808fcb#221128 on Sun Sep 15 20:10:43 2024

Program being simulated:

    /home/thesis/federica.bader/Desktop/AEAD_accelerator/step3_improvingRound/Release/Improved_Round

    Total cycle count           :                 3140
    Report cycle count          :                 3140
    Total instruction count     :                 2588
    Report instruction count    :                 2588
    Total size in program memory:                 7468

Command used to generate this report: ::iss profile storage_access_save -file
"/home/thesis/federica.bader/Desktop/AEAD_accelerator/step3_improvingRound/storage_access_report_ImprovedRound.txt"
-function_details Off -field_details Off -function_summary Off -data "" -cycle_count 0 -instruction_count 0 -hide_instruction_bits
Off


Function storage access summary:

    Storage               Read-count  Read-count   Read-count Write-count Write-count  Write-count Function name
                             (total)  (function) (% of total)     (total)  (function) (% of total)
    -------------------- ----------- ----------- ------------ ----------- ----------- ------------
    --------------------------------------------------
    DMb                         1760         560       31.82%        1820         560       30.77% PROUNDS
    _Z7PROUNDSP13ascon_state_ti
                                             492       27.95%                     436       23.96% ascon_aead_decrypt
                                             _Z18ascon_aead_decryptPhPKhS1_y
                                             444       25.23%                     384       21.10% ascon_aead_encrypt
                                             _Z18ascon_aead_encryptPhS_PKhyS
                                              20        1.14%                      24        1.32% crypto_aead_decrypt
                                              _Z19crypto_aead_decryptPhPyS_P
                                              20        1.14%                      24        1.32% crypto_aead_encrypt
                                              _Z19crypto_aead_encryptPhPyPKh
                                              20        1.14%                     180        9.89% main _main
                                             192       10.91%                     192       10.55% memcpy memcpy
                                               4        0.23%                       0        0.00% printf printf
                                               8        0.45%                      20        1.10% vfprintf vfprintf

    PMb                        24640        6896       27.99%           0           0        0.00% PROUNDS
    _Z7PROUNDSP13ascon_state_ti
                                              48        0.19%                       0        0.00% _main_init _main_init
                                              40        0.16%                       0        0.00% _start_basic _start_basic
                                            5312       21.56%                       0        0.00% ascon_aead_decrypt
                                            _Z18ascon_aead_decryptPhPKhS1_y
                                            5064       20.55%                       0        0.00% ascon_aead_encrypt
                                            _Z18ascon_aead_encryptPhS_PKhyS
                                              24        0.10%                       0        0.00% clib_hosted_io clib_hosted_io
                                             176        0.71%                       0        0.00% crypto_aead_decrypt
                                             _Z19crypto_aead_decryptPhPyS_P
                                             112        0.45%                       0        0.00% crypto_aead_encrypt
                                             _Z19crypto_aead_encryptPhPyPKh
                                            1256        5.10%                       0        0.00% main _main
                                            5376       21.82%                       0        0.00% memcpy memcpy
                                              40        0.16%                       0        0.00% printf printf
                                             296        1.20%                       0        0.00% vfprintf vfprintf
    S                            610         610      100.00%         610         610      100.00% PROUNDS
    _Z7PROUNDSP13ascon_state_ti
    X                           7742        1484       19.17%        2343         304       12.97% PROUNDS
    _Z7PROUNDSP13ascon_state_ti
                                              15        0.19%                       2        0.09% _main_init _main_init
                                               5        0.06%                       1        0.04% _start_basic _start_basic
                                            1664       21.49%                     668       28.51% ascon_aead_decrypt
                                            _Z18ascon_aead_decryptPhPKhS1_y
                                            1567       20.24%                     635       27.10% ascon_aead_encrypt
                                            _Z18ascon_aead_encryptPhS_PKhyS
                                               2        0.03%                       0        0.00% clib_hosted_io clib_hosted_io
                                              52        0.67%                      17        0.73% crypto_aead_decrypt
                                              _Z19crypto_aead_decryptPhPyS_P
                                              40        0.52%                      14        0.60% crypto_aead_encrypt
                                              _Z19crypto_aead_encryptPhPyPKh
                                             538        6.95%                      66        2.82% main _main
                                            2352       30.38%                     624       26.63% memcpy memcpy
                                               3        0.04%                       3        0.13% printf printf
                                              20        0.26%                       9        0.38% vfprintf vfprintf
```

Report B.4: Storage access report of the accelerated algorithm Ascon

```
Function profiling report information for ::iss generated by Checkers U-2022.12#33f3808fcb#221128 on Tue Sep  3 18:21:08 2024


Program being simulated:

    /home/thesis/federica.bader/Desktop/AEAD_accelerator/step4_unrolling/UnrollingP2/Release/Unrolling_P2

    Total cycle count           :                 2744
    Report cycle count          :                 2736
    Total instruction count     :                 2340
    Total size in program memory:                 7492

Command used to generate this report: ::iss profile save "/home/thesis/federica.bader/Desktop/AEAD_accelerator/step4_unrolling/UnrollingP2/function_report_P2.txt" -type
function_profiling -xml 0 -function_details 0 -one_file 1 -call_details Off -call_tree Off -data "" -cycle_count 0 -instruction_count 0

          Calls  Cycles tot  Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max     Low PC     High PC     Function
                     (func)     (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    --------------------------------------------------
              1           8        0.29%           8           8           8        2736      100.00%        2736        2736        2736          0          23 _main_init
              1         158        5.77%         158         158         158        2728       99.71%        2728        2728        2728       6416        7259 main
              1          25        0.91%          25          25          25        1296       47.37%        1296        1296        1296       6288        6411 crypto_aead_decrypt
              1         664       24.27%         664         664         664        1271       46.45%        1271        1271        1271       2968        6195 ascon_aead_decrypt
              1          17        0.62%          17          17          17        1257       45.94%        1257        1257        1257       6200        6283 crypto_aead_encrypt
              1         633       23.14%         633         633         633        1240       45.32%        1240        1240        1240         40        2967 ascon_aead_encrypt
             24         720       26.32%          30          30          30         720       26.32%          30          30          30       7480        7511 memcpy
             14         494       18.06%          35          35          36         494       18.06%          35          35          36       7264        7399 P
              1           3        0.11%           3           3           3          17        0.62%          17          17          17       7460        7475 printf
              1          11        0.40%          11          11          11          14        0.51%          14          14          14       7400        7459 vfprintf
              1           3        0.11%           3           3           3           3        0.11%           3           3           3         36          39 clib_hosted_io
              0           0        0.00%           -           -           -           0        0.00%           -           -           -         24          35 _start_basic

          Calls  Cycles tot  Cycles tot  Cycles min  Cycles avg  Cycles max  Cycles tot   Cycles tot  Cycles min  Cycles avg  Cycles max     Low PC     High PC     Function
                     (func)     (%func)      (func)      (func)      (func) (func+desc) (%func+desc) (func+desc) (func+desc) (func+desc)
    ----------- ----------- ------------ ----------- ----------- ----------- ----------- ------------ ----------- ----------- ----------- ---------- ----------
    --------------------------------------------------
             24         720       26.32%          30          30          30         720       26.32%          30          30          30       7480        7511 memcpy memcpy
              1         664       24.27%         664         664         664        1271       46.45%        1271        1271        1271       2968        6195 ascon_aead_decrypt
              1         633       23.14%         633         633         633        1240       45.32%        1240        1240        1240         40        2967 ascon_aead_encrypt
             14         494       18.06%          35          35          36         494       18.06%          35          35          36       7264        7399 P
              1         158        5.77%         158         158         158        2728       99.71%        2728        2728        2728       6416        7259 main
              1          25        0.91%          25          25          25        1296       47.37%        1296        1296        1296       6288        6411 crypto_aead_decrypt
              1          17        0.62%          17          17          17        1257       45.94%        1257        1257        1257       6200        6283 crypto_aead_encrypt
              1          11        0.40%          11          11          11          14        0.51%          14          14          14       7400        7459 vfprintf
              1           8        0.29%           8           8           8        2736      100.00%        2736        2736        2736          0          23 _main_init
              1           3        0.11%           3           3           3           3        0.11%           3           3           3         36          39 clib_hosted_io
              1           3        0.11%           3           3           3          17        0.62%          17          17          17       7460        7475 printf
              0           0        0.00%           -           -           -           0        0.00%           -           -           -         24          35 _start_basic
```

Report B.5: Function report of the P2 algorithm Ascon

Bibliography

[1] J. Black. «Authenticated encryption». In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 11–21. ISBN: 978-0-387-23483-0. DOI: `10.1007/0-387-23483-7_15`. URL: `https://doi.org/10.1007/0-387-23483-7_15` (cit. on p. 6).

[2] David Salomon. «Hashing». In: *Data Compression: The Complete Reference*. New York, NY: Springer New York, 1998, pp. 357–360. ISBN: 978-1-4757-2939-9. DOI: `10.1007/978-1-4757-2939-9_13`. URL: `https://doi.org/10.1007/978-1-4757-2939-9_13` (cit. on p. 7).

[3] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. «Duplexing the sponge: single-pass authenticated encryption and other applications». In: *Selected Areas in Cryptography: 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers 18*. Springer. 2012, pp. 320–337 (cit. on p. 8).

[4] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. «Ascon v1.2: Lightweight Authenticated Encryption and Hashing». In: *Journal of Cryptology* 34 (2021). DOI: `10.1007/s00145-021-09398-9` (cit. on pp. 8, 11).

[5] The RISC-V Foundation. *RISC-V Unprivileged ISA Specification*. Available at: `https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view`. Accessed: 2024-11-06. 2024 (cit. on p. 14).

[6] Synopsys, Inc. *ASIP Designer*. 2024. URL: `https://www.synopsys.com/dw/ipdir.php?ds=asip-designer` (cit. on p. 15).

[7] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. «An analysis of accelerator coupling in heterogeneous architectures». In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: `10.1145/2744769.2744794` (cit. on p. 16).

[8] Ascon Team. *Ascon Cryptographic Algorithm C Implementations*. 2024. URL: `https://github.com/ascon/ascon-c.git` (cit. on p. 17).

[9] Jack W Davidson and Sanjay Jinturkar. *An aggressive approach to loop unrolling*. Tech. rep. Citeseer, 1995 (cit. on p. 31).

[10] Stefan Steinegger and Robert Primas. «A Fast and Compact RISC-V Accelerator for Ascon and Friends». In: *Smart Card Research and Advanced Applications*. Ed. by Pierre-Yvan Liardet and Nele Mentens. Cham: Springer International Publishing, 2021, pp. 53–67. ISBN: 978-3-030-68487-7 (cit. on pp. 34, 57–59).

[11] Inc. Advanced Micro Devices. *Artix-7 FPGAs - Advantages*. `https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/artix-7.html` (cit. on p. 46).

[12] AMD. *Vivado™ Design Suite*. `https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html`. Accessed: 2024-11-17. URL: `https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html` (cit. on p. 46).

[13] Inc. Synopsys and UMC. *Synopsys and UMC Release 65-Nanometer Low Power Design Flow Enabled by the Unified Power Format*. `https://news.synopsys.com/home?item=122944` (cit. on p. 48).

[14] Carlos Gewehr, Lucas Luza, and Fernando Gehm Moraes. «Hardware Acceleration of Crystals-Kyber in Low-Complexity Embedded Systems with RISC-V Instruction Set Extensions». In: *IEEE Access* (2024) (cit. on pp. 52, 57).

[15] Hao Cheng, Johann Grosschadl, Ben Marshall, Dan Page, and Thinh Pham. «RISC-V instruction set extensions for lightweight symmetric cryptography». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023), pp. 193–237 (cit. on pp. 52, 57, 58).

[16] Hannes Gross, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. «Ascon hardware implementations and side-channel evaluation». In: *Microprocessors and Microsystems* 52 (2017), pp. 470–479. ISSN: 0141-9331. URL: `https://www.sciencedirect.com/science/article/pii/S0141933116302721` (cit. on p. 59).

[17] Safiullah Khan, Wai-Kong Lee, Angshuman Karmakar, Jose Maria Bermudo Mera, Abdul Majeed, and Seong Oun Hwang. *Area-time Efficient Implementation of NIST Lightweight Hash Functions Targeting IoT Applications*. Cryptology ePrint Archive, Paper 2022/1716. 2022. URL: `https://eprint.iacr.org/2022/1716` (cit. on p. 59).

[18] Kamal Raj and Srinivasu Bodapati. «FPGA Based Light Weight Encryption of Medical Data for IoMT Devices using ASCON Cipher». In: *2022 IEEE International Symposium on Smart Electronic Systems (iSES)*. 2022, pp. 196–201. DOI: `10.1109/iSES54909.2022.00048` (cit. on pp. 59, 60).

[19] George Athanasiou, Dimitris Boufeas, and Evangelia Konstantopoulou. «A Robust ASCON Cryptographic Coprocessor for Secure IoT Applications». In: Mar. 2024, pp. 1–6. DOI: 10.1109/PACET60398.2024.10497076 (cit. on p. 60).

[20] Xiangdong Wei, Mohamed El-Hadedy, Sergiu Mosanu, Zhengping Zhu, Wen-Mei Hwu, and Xinfei Guo. «RECO-HCON: A High-Throughput Reconfigurable Compact ASCON Processor for Trusted IoT». In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022, pp. 1–6. DOI: 10.1109/SOCC56010.2022.9908100 (cit. on p. 60).