PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

NICOLAS SILVA MOURA

ACCELERATION OF AEAD ALGORITHMS FOR RESOURCE-CONSTRAINED EMBEDDED DEVICES

Porto Alegre
2024

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

ACCELERATION OF AEAD ALGORITHMS FOR RESOURCE-CONSTRAINED EMBEDDED DEVICES

NICOLAS SILVA MOURA

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes
Co-Advisor: Prof. Dr. Rafael Fraga Garibotti

Porto Alegre
2024

Ficha Catalográfica

M929a Moura, Nicolas Silva

Acceleration of AEAD Algorithms for Resource-Constrained Embedded Devices / Nicolas Silva Moura. – 2024.
58.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.
Coorientador: Prof. Dr. Rafael Fraga Garibotti.

1. Lightweight Cryptography. 2. Ascon. 3. RISC-V. 4. Hardware Acceleration. I. Moraes, Fernando Gehm. II. Garibotti, Rafael Fraga. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

NICOLAS SILVA MOURA

ACCELERATION OF AEAD ALGORITHMS FOR RESOURCE-CONSTRAINED EMBEDDED DEVICES

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 20, 2024.

COMMITTEE MEMBERS:

Prof. Dr. Rafael Iankowski Soares (PPGC/UFPel)

Prof. Dr. Avelino Zorzo (PPGCC/PUCRS)

Prof. Dr. Rafael Fraga Garibotti (Vector Trading- Co-Advisor)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

ACELERAÇÃO DE ALGORITMOS AEAD PARA DISPOSITIVOS EMBARCADOS COM RECURSOS LIMITADOS

RESUMO

A quantidade de informações sensíveis e dados processados em dispositivos IoT está constantemente aumentando. Como resultado, a segurança tornou-se uma preocupação crucial. Embora a criptografia de dados seja necessária, os custos que os algoritmos de criptografia tipicamente requerem para proteger os dados raramente são toleráveis em dispositivos embarcados de baixo custo. Isso levou ao surgimento de um novo ramo de pesquisa chamado Criptografia Leve (LWC), que visa introduzir algoritmos que proporcionem níveis aceitáveis de segurança enquanto consomem o mínimo de recursos possível. Devido à crescente relevância deste campo e às muitas propostas divergentes, o *National Institute of Standards and Technology* (NIST) lançou uma competição para selecionar um algoritmo LWC para padronizá-lo, de forma semelhante ao que foi feito anteriormente para o AES. Em fevereiro de 2023, o algoritmo Ascon foi anunciado como o vencedor da competição, e espera-se que seja padronizado pelo NIST em 2024. Este trabalho apresenta uma avaliação comparativa entre três algoritmos de criptografia autenticada com dados associados (AEAD), Ascon, AES-128 no modo CCM e ChaCha20-Poly1305, no contexto de um processador RISC-V de baixa complexidade, considerando o algoritmo executando em software e com extensões do conjunto de instruções (ISEs), comparando seu desempenho e compromissos em uma tecnologia FDSOI de 28nm da ST *Microelectronics*. Os resultados apresentam uma avaliação abrangente de PPA (Potência, Desempenho e Área) para os três algoritmos AEAD, mostrando um ganho de desempenho de 95,1%, 60,3% e 5,2%, juntamente com um aumento na eficiência energética de 94,2%, 65,6% e 17,2%, para AES, Ascon e ChaCha20-Poly1305, respectivamente. O custo em área foi até 9%. Tais resultados demonstram que dispositivos com recursos limitados que criptografam um alto volume de mensagens se beneficiam significativamente da aceleração de hardware.

Palavras-Chave: Criptografia leve, Ascon, RISC-V, aceleração de hardware.

ACCELERATION OF AEAD ALGORITHMS FOR RESOURCE-CONSTRAINED EMBEDDED DEVICES

ABSTRACT

The amount of sensitive information and data processed on IoT devices constantly increases. As a result, security has become a crucial concern. Although data encryption is necessary, the large overheads that encryption algorithms typically require to protect data are rarely tolerable on low-end devices. This has led to the emergence of a new branch of research called Lightweight Cryptography (LWC), which aims to introduce new algorithms that provide acceptable levels of security while consuming as few resources as possible. Due to the growing relevance of this field and the many divergent proposals, the National Institute of Standards and Technology (NIST) launched a competition to select an LWC algorithm to standardize similarly to that previously done for the Advanced Encryption Standard (AES). In February 2023, the Ascon algorithm was announced as the competition winner, and it is expected to be standardized by NIST in 2024. This work presents a comparative evaluation between three authenticated encryption algorithms with associated data (AEAD), namely, Ascon, AES-128 in CCM mode, and ChaCha20-Poly1305 in the context of a low-complexity RISC-V processor, considering the algorithm executing in software and with instruction set extensions (ISEs), comparing their performance and trade-offs in a 28nm FDSOI technology from ST Microelectronics. The results present a comprehensive evaluation of PPA (Power, Performance and Area) for the three AEAD algorithms, showing a performance gain of 95.1%, 60.3%, and 5.2%, along with an increase in energy efficiency of 94.2%, 65.6%, and 17.2%, for AES, Ascon, and ChaCha20-Poly1305, respectively. The area overheads were observed to be up to 9%. Such results demonstrate that devices with limited resources that encrypt a high message volume benefit significantly from hardware acceleration.

Keywords: Lightweight Cryptography, Ascon, RISC-V, Hardware Acceleration.

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

LIST OF ACRONYMS

LWC – Lightweight cryptography
NIST – National Institute of Standards and Technology
AEAD – Authenticated Encryption with Associated Data
AES – Advanced Encryption Standard
IoT – Internet of Things
IoE – Internet of Energy
TLS – Transport Layer Security
DTLS – Datagram Transport Layer Security
ISE – Instruction Set Extensions
ISA – Instruction Set Architecture
SIMD – Single Instruction Multiple Data
CSR – Control and Status Registers
LUT – Look-Up Table
CBC – Cipher-block chaining
CTR – Counter mode
GCM – Galois/Counter Mode
CCM – Counter with Cipher Block Chaining Message Authentication Code
CPU – Central Processing Unit
GPU – Graphics Processing Unit
ECDSA – Elliptic Curve Digital Signature Algorithm
PPA – Power-Performance-Area
FPGA – Field-programmable gate array
SPN – Substitution-Permutation-Network
S-Box – Substitution-Box
GCC – GNU Compiler Collection

CONTENTS

1. INTRODUCTION

The Internet of Things (IoT) era brought a proliferation of edge devices into our lives, ranging from simple sensor-based devices in home automation to high-end systems embedded in autonomous vehicles [Guo et al., 2016]. The growing use of IoT edge devices is due to their improved computing performance and memory resources, combined with low power consumption [da Rocha et al., 2022]. However, the widespread presence of connected devices, as envisaged by the IoT, raises security concerns that must be addressed in the design of the underlying system. IoT end devices are likely to exchange sensitive data. Thus, adopted communication mechanisms must guarantee certain security levels, aiming to avoid, e.g., data corruption, use by nonauthorized entities, and denial of service, to name a few existing threats.

Secure communication has become a fundamental requirement due to the exponential growth of low-power embedded systems. These applications have no high computing power and memory capacity and are often battery-powered or use energy harvesting. Authenticated Encryption with Associated Data (AEAD) ensures confidentiality and authenticity of sensitive data. Some low-power applications use AES-128 [NIST, 2001a] with the Counter with Cipher Block Chaining-Message Authentication Code (CCM) operation mode [NIST, 2007] and ChaCha20-Poly1305 [Nir and Langley, 2015] as the AEAD algorithms of choice. These algorithms are supported in many Internet protocols, notably TLS [Rescorla, 2018] and DTLS [Rescorla et al., 2022].

Given the difficulty of processing standard cryptography algorithms in low-power embedded applications, a new strand of algorithms, currently known as lightweight cryptography algorithms, emerged in recent years. The main problem faced was the many different applications available, some with more memory and less processing and others with more memory and less energy limitation. To solve this issue, the National Institute of Standards and Technology (NIST) organized a competition to standardize an LWC algorithm. In February 2023, they announced that Ascon won the LWC competition.

Hardware acceleration via **Instruction Set Extensions** (ISEs) can improve the performance, memory footprint, and energy efficiency of encryption, which brings benefits to restricted embedded applications, such as (*i*) sharing of generic resources between components such as register file and RAM interface; (*ii*) trivial data transfer between processor components, avoiding unnecessary memory accesses; (*iii*) reduces system complexity, for example, avoiding external buses and interrupt controllers.

This work evaluates acceleration methods for three AEAD algorithms, namely Ascon, AES-128 in CCM mode and ChaCha20-Poly1305. The work compares the effect of acceleration on different algorithms in the context of the low-complexity IBEX RISC-V core. The evaluations consider software and hardware implementations, accelerations with ISEs

(*Zbkb*, *Zkne*, *Xascon*), and also an acceleration with an external Ascon core loosely coupled to RISC-V. The main focus of this work is on the Ascon algorithm, which was recently standardized.

1.1 Motivation

The primary motivation for this MSc dissertation is to explore the field of lightweight cryptography by evaluating the newly standardized Ascon algorithm. Furthermore, this work offers the opportunity to investigate the performance of this algorithm against different hardware acceleration approaches and assess the most relevant aspects of each implementation. Once the algorithm has been standardized, the next generation of constrained devices will benefit from this research, which presents methods for improving efficiency while using fewer resources.

1.2 Objectives

This work has two strategic objectives. The first objective is to evaluate and compare acceleration methods for the Ascon algorithm using the RISC-V Ibex environment. The second objective is to compare these results with two other AEAD algorithms (AES-128 in CCM mode and ChaCha20-Poly1305).

Specific objectives are as follows:

1. Profile a software-only implementation of the Ascon, AES-128 in CCM mode, and ChaCha20-Poly1305 algorithms running on an unmodified Ibex RISC-V core to establish a baseline for the metrics to be used for comparison among different implementations;
2. Modify the Ibex RISC-V core by incorporating the *Zbkb*, *Zkne* and *Xascon* extensions, which contains specific bit manipulation operations that are particularly relevant to cryptographic algorithms;
3. Evaluate a hardware-only Ascon implementation to establish a reference for energy efficiency and area cost;
4. Quantitatively evaluate the effectiveness of the implementations of hardware accelerations using various metrics, such as performance, memory consumption, cycles spent for the algorithm, die area, and power consumption.

1.3 Contribution

The main contribution of this work is to provide a RISC-V processor (IBEX) extended with specialized instruction for cryptography, targeting the Ascon, AES-128 in CCM mode, and ChaCha20-Poly1305 algorithm. The *Zbkb*, *Zkne*, and *Xascon* ISEs have toolchain support through a modified GCC. Note that the proposed extensions can be applied to other AEAD algorithms and are not restricted to the ones evaluated in this work.

1.4 Methodology

This work started by selecting an open-source RISC-V distribution. The chosen processor was IBEX [Zurich and of Bologna, 2017] due to its simplified architecture, specifically designed for embedded devices. Subsequently, after configuring the simulation environment, a thorough validation process confirmed the correct functionality of the entire setup. Following this validation, the *Zbkb* extension [Marshall, 2021], a bit manipulation extension derived from the scalar cryptography extension, was incorporated. These additional instructions facilitate the acceleration of the Ascon permutation by employing the *zip* and *unzip* instructions to execute rotations in blocks of 64-bit data on Ibex, which inherently operates on 32-bit data. This extension can also be used in ChaCha20-Poly1305.

After integrating *Zbkb*, specific instructions were introduced to the RISC-V architecture, *XAscon* ISE, and GCC underwent modifications to facilitate *XAscon* acceleration. The next step involved incorporating the AXI Lite communication protocol into IBEX to establish communication between the processor and the dedicated Ascon hardware block provided by the Ascon team.

The extension *Zkne* was also implemented to accelerate AES-128 in CCM mode with the instructions *aes32esmi* and *aes32esi* to compare results with Ascon and ChaCha20-Poly1305.

Throughout these modifications, a system of flags was implemented to parameterize IBEX, enhancing the ease of evaluating different approaches, particularly regarding hardware synthesis. This approach ensures a comprehensive comparison of the area and performance between the baseline and extended processor configurations.

This study compares various implementations of the AEAD algorithms:

1. Evaluation of a software implementation provided by the Ascon team [Ascon, 2022a], written in the C language and executed on the Ibex base processor without any extensions. It is essential to note that this implementation is a reference for the Ascon team.

Repeat this step for AES-128 in CCM mode and ChaCha20-Poly1305 using algorithms from the Tinycrypt library [Intel, 2017].

2. Evaluation of the accelerated Ascon version using the *Zbkb* extension and the bit interleaving technique [Ascon, 2022a]. To achieve this, the Ibex processor is modified accordingly. Repeat the evaluation for ChaCha20-Poly1305 and compare the acceleration reached by both algorithms.
3. Evaluation of the accelerated Ascon using the specific instructions *XAscon*.
4. Evaluation of AES-128 in CCM mode using the *Zkne* extension.
5. Evaluation of the hardware implementation provided by the Ascon team, as documented in [Ascon, 2022b].

These implementations enable the exploration of the design space associated with AEAD algorithms. Moreover, all implementations utilize the post-synthesis netlist, ensuring fair performance and energy evaluation. Instruction profiling was conducted using the performance counters available in the processor.

1.5 Document organization

This manuscript is organized as follows:

- Chapter 2 describes background knowledge related to the RISC-V architecture, hardware acceleration, and AEAD concepts required to follow this work.
- Chapter 3 presents the state-of-the-art of hardware accelerator involving RISC-V, and LWC and AEAD cryptography.
- Chapter 4 describes the core of this work, describing all implementation aspects for *Zbkb*, *Zkne*, *XAscon*, and the crypto-core;
- Chapter 5 presents results for all implementations, discussing the obtained results;
- Chapter 6 concludes this work, pointing out directions for future work.

2. FUNDAMENTAL CONCEPTS

This Chapter introduces concepts required for understanding this work. Section 2.1 covers concepts related to RISC-V. Section 2.2 presents the concept of AEAD encryption and the three algorithms mentioned in this work. Section 2.3 overviews hardware acceleration concepts.

2.1 RISC-V Architecture

RISC-V is an open-source ISA (Instruction set architecture) [RISC-V, 2019], initially developed at the University of Berkeley to provide an architecture designed to follow the principles of simplicity, flexibility, and extensibility, making it suitable for the largest number of applications. To achieve these features, RISC-V was developed with a basic set of instructions and optional extensions to be added depending on the target application. RISC-V ISA has three ratified versions: RVWMO, RV32I, and RV64I. The last two have some specificities, but the main difference is the register size, which is 32 and 64 bits. The RVWMO version targets multiprocessed systems, which are out of the scope of this work. Table 2.1 shows the standard ISA instructions and the M extension.

Table 2.1 – Instructions of the RV32I processor and the *M* extension.

Instruction Class	**Opcodes**
LOAD	`LB`, `LH`, `LW`, `LBU`, `LHU`, `FLW`
STORE	`SB`, `SH`, `SW`, `FSW`
IMM	`LUI`, `AUIPC`, `SLTI`, `SLTIU`, `XORI`, `ORI`, `ANDI`, `SLLI`, `SRLI`, `SRAI`
REG	`SLL`, `SLT`, `SLTU`, `XOR`, `SRL`, `SRA`, `OR`, `AND`
ADDSUB	`ADD`, `ADDI`, `SUB`
BRANCH	`BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`
JUMP	`JAL`, `JALR`
MULT (*M* extension)	`MUL`, `MULH`, `MULHSU`, `MULHU`
DIV (*M* extension)	`DIV`, `DIVU`, `REM`, `REMU`

Several RISC-V processors are proposed in the literature, and some implementations are publicly available, offering a spectrum of functionalities ranging from simple microcontroller-oriented designs to highly complex multi-core superscalar processors optimized for performance. Among the options considered for this study are:

- **Rocket** core [Alliance, 2023], a scalar processor featuring a 5-stage pipeline. It includes a core with a complete Arithmetic Logic Unit (ALU) and an optional Floating-Point Unit (FPU). The Rocket core incorporates a Memory Management Unit (MMU)

supporting page-based virtual memory and a data cache. It also supports RISC-V machine, supervisor, and user privilege levels. The Rocket core is the Rocket Chip System on Chip (SoC) processor.

- The **OpenHW Group CORE-V CV32E40P** [OpenHWGroup, 2023], a 32-bit RISC-V processor with 4 pipeline stages. It supports the multiplication and division (M) and compressed instructions (C) extensions, with the option to include the F extension and the *Zfinx* extension for performing floating-point operations in integer registers. Additionally, the CV32E40P processor can incorporate custom extensions like hardware loops, Single Instruction Multiple Data (SIMD) operations, and fixed-point arithmetic.
- Finally, the processor selected for this study is the **Ibex** [Zurich and of Bologna, 2017] processor. Ibex is a 32-bit RISC-V core implemented in SystemVerilog with a 2-stage pipeline. Highly parameterized, Ibex can be tailored for various embedded applications. It supports both the integer (RV32I) and embedded (RV32E) instruction sets and includes extensions for integer (I), compressed instructions (C), bit manipulation (B), multiplication and division (M).

2.1.1 RISC-V Instruction Set Extensions - ISE

The base RISC-V ISA provides a minimum set of instructions for the operation of the processor (Table 2.1). ISEs refer to a set of additional instructions to the base RISC-V ISA, which are used to provide additional resources depending on the needs of the design. Table 2.2 presents the RISC-V cryptography ISEs.

Table 2.2 – RISC-V cryptography ISEs [Marshall, 2021].

Sub Extension	**Description**
Zbkb	Bit-manipulation for cryptography
Zbkc	Carry-less multiply instructions
Zbkx	Crossbar permutation instructions
Zknd	NIST Suite: AES Decryption
Zkne	NIST Suite: AES Encryption
Zknh	NIST Suite: Hash Function Instructions
Zksed	ShangMi Suite: SM4 Block Cipher Instructions
Zksh	ShangMi Suite: SM3 Hash Function Instructions
Zkr	Entropy Source Extension
Zkn	NIST Algorithm Suite
Zks	ShangMi Algorithm Suite
Zk	Standard scalar cryptography extension
Zkt	Data Independent Execution Latency

In this work, two ISEs are used, namely *Zbkb*, which refers to the bit manipulation subset, and *Zkne*, which refers to the subset dedicated to AES encryption, both subsets of the RISC-V encryption extension.

The URL https://five-embeddev.com/riscv-bitmanip/draft/bitmanip.html presents a detailed list all instructions for the *Zba*, *Zbb*, *Zbc*, *Zbs*, and *Zbkb* extensions. For example, this URL mentions for the "zip" instruction: "*This instruction is useful for implementing the SHA3 cryptographic hash function on a 32-bit architecture, as it implements the bit-interleaving operation used to speed up the 64-bit rotations directly*".

Table 2.3 presents the list of instructions for the *Zbkb* set added in this work. With the implementation of these instructions, the compiler can generate and send these instructions to the processor. Table 2.4 presents the list of instructions from the *Zkne* set. With these instructions is possible to accelerate the AES-128 algorithm and compare it with the acceleration of Ascon to obtain a fair comparison.

Table 2.3 – *Zbkb* - Bitmanip instructions for Cryptography.

Instruction Class	**Opcodes**
`ror`	Rotate right (Register)
`rol`	Rotate left (Register)
`rori`	Rotate right (Immediate)
`andn`	AND with inverted operand
`orn`	OR with inverted operand
`xnor`	Exclusive NOR
`pack`	Pack low halves of registers
`packh`	Pack low bytes of registers
`brev8`	Reverse bits in bytes
`rev8`	Byte-reverse register
`zip`	Scatters the odd and even bits of a source word into the high and low halves of a destination word.
`unzip`	gathers bits from the high and low halves of the source word into odd/even bit positions in the destination word.

Table 2.4 – *Zbkb* - NIST Suite: AES Encryption.

Instruction Class	**Opcodes**
`aes32esi`	AES final round encryption (RV32)
`aes32esmi`	AES middle round encryption (RV32)
`aes64es`	AES encryption final round instruction (RV64)
`aes64esm`	AES encryption middle round instruction (RV64)
`aes64ks1i`	AES Key Schedule Instruction 1 (RV64)
`aes64ks2`	AES Key Schedule Instruction 2 (RV64)

2.1.2 RISC-V Ibex

Ibex, as described in [Zurich and of Bologna, 2017], was designed for embedded systems, focusing on low power consumption. The extensions Ibex supports are outlined in Table 2.5. Ibex also integrates performance counters, encompassing M-Mode (Machine mode) and U-Mode (User mode). It aligns with the RISC-V Privileged specification by incorporating Control and Status Registers (CSRs). Furthermore, Ibex offers parameterization capabilities, enabling the selection of extensions to synthesize the processor with minimal area utilization.

Table 2.5 – Ibex Instruction Set Extensions [Ibex, 2023a].

Extension	**Configurability**
C: Compressed Instructions	always enabled
M: Integer Multiplication and Division	optional
B: Bit-Manipulation Instructions	optional
Zicsr: Control and Status Register Instructions	always enabled
Zifencei: Instruction-Fetch Fence (Fence instruction order device I/O and memory accesses)	always enabled
Smepmp: PMP (Physical Memory Protection) Enhancements for memory access and execution prevention on machine mode	always enabled in configurations with PMP

In Table 2.5, the “configurability” column means whether the extension is obligatory or optional. Certain extensions are recommended to always be enabled to ensure a correct synthesis, while others, such as *M*, can be added or omitted according to design requirements. These extensions can be activated via flags before the synthesis, which is beneficial in managing the extensions and associated blocks. The *Zicsr* is always enabled, as it is responsible for handling interruptions. Figure 2.1 presents the Ibex architecture [Ibex, 2023a].

Ibex implements extra features (when the SecureIbex parameter is set) to support security-critical applications. All features are runtime configurable via bits in the *cpuctrl* custom CSR. For example, bits [19:16] in this CSR specifies the number of dummy instructions inserted every *n* instructions where *n* is a range set based on the value written to this register, where: 0x0 = 1-4, 0x1 = 1-8, 0x3 = 1-16, 0x7 = 1-32, 0xF = 1-64. The purpose of inserting dummy instructions is to avoid side-channel attacks (SCAs). Section 2.1.3 details these security features.

Ibex incorporates CSR performance counters. These can be accessed to provide the values associated with each counter. Chapter 5 displays the event selectors used to monitor these performance counters. Evaluating the algorithm and creating the instruction profile for Ascon was possible using these performance counters. By default, performance

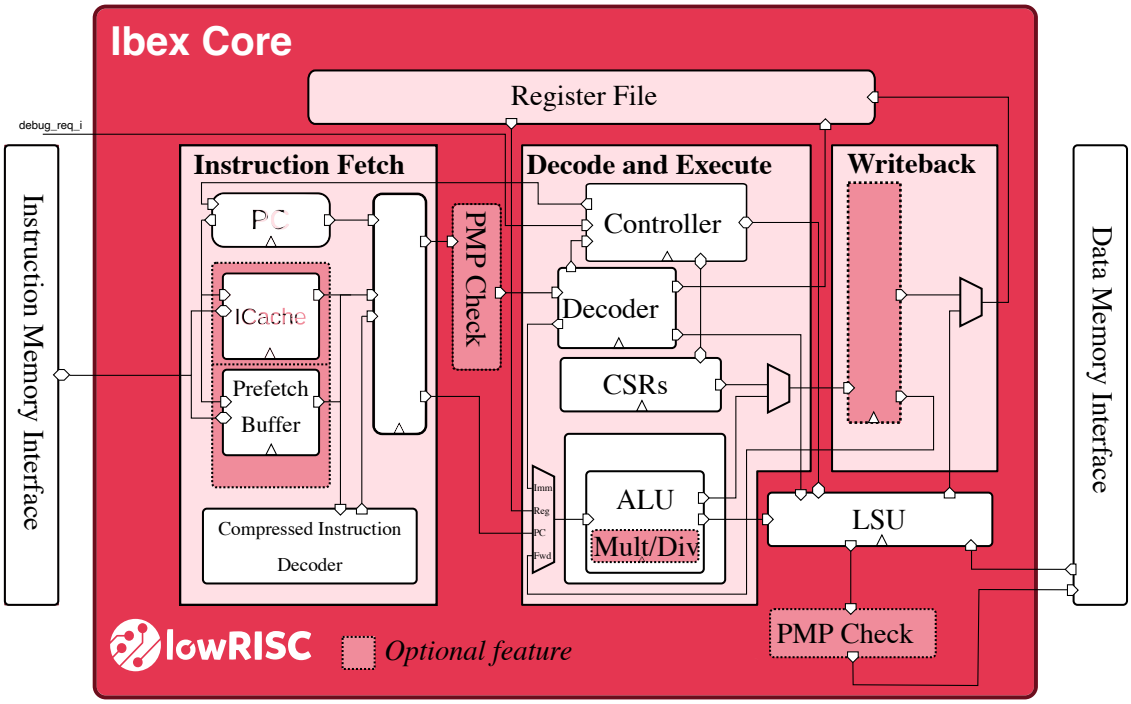


Figure 2.1 – RISC-V Ibex Architecture.

counters are activated after reset but can be enabled by software. This can be used to gather results from a specific function within the software implementation. This strategy was employed in the *Ascon_Permut* function to isolate and record the results from this portion of the code.

2.1.3 Ibex Security

The work does not focus on evaluating vulnerabilities to attacks. We consider that the published and used extensions have already been carried out in this previous analysis, and no new vulnerabilities were added respecting the implementation models. Source code of algorithms made available by the creators themselves or from standard libraries were also used.

Another point to highlight is the security part of the Ibex itself. ibex has a security system that implements a set of features when the *SecureIbex* [Ibex, 2023b] parameter is defined as follows:

- **Data Independent Timing**: This feature ensures that the execution time and energy consumption of all instructions remain independent of the input data. This feature makes it challenging for potential attackers to infer sensitive data through observations of power dissipation or by exploiting timing-based attacks.

- **Dummy Instruction Insertion**. This corresponds to inserting dummy instructions at random intervals within the execution pipeline. The objective is to thwart attempts by attackers to discern the executed code and to inhibit precisely timed fault injection attacks. These instructions have no functional impact on the code being executed.
- **Bus Integrity Checking**. Instruction memory channels are equipped with signals to verify the integrity of the bus. This validation process is conducted in relation to the provided check bit.
- **Register File ECC (Error Correction Code)**. This functionality involves performing an ECC check on all reads from the register file. It serves to detect fault injection attacks. Notably, the ECC does not correct detected errors but instead triggers an internal alert upon detection.

Ibex has three alert outputs for signaling security issues. The internal major alert indicates a critical security issue from which the core cannot recover, which was detected internally. The bus major alert indicates a critical security issue from which the core cannot recover, which was detected on incoming bus data. The minor alert indicates potential security issues that a system can monitor over time.

2.2 Introduction to AEAD

AEAD (Authenticated Encryption with Associated Data) algorithms are designed to offer both confidentiality and authentication for a given plaintext P while also providing authentication (but not necessarily confidentiality) for associated data A. An example use case for AEAD is in network packets, where headers (A) are visible to routers while the payloads remain private. In addition to A and P, the AEAD process E (Equation (2.1)) requires a key K and a unique nonce N. These additional values are utilized to initialize the algorithm's state. In the case of Ascon, this initialization also involves a static initialization vector IV, while in AES-CCM mode, the message length determines the value of CTR_0. The outputs of the AEAD process are the ciphertext C and an authentication tag T. Figure 2.2 presents the AEAD flow.

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

Common AEAD algorithms include AES-GCM (Galois/Counter Mode), AES-CCM (Counter with CBC-MAC), and ChaCha20-Poly1305. These algorithms are widely used in various security protocols, such as TLS (Transport layer Security), to secure Internet communication.

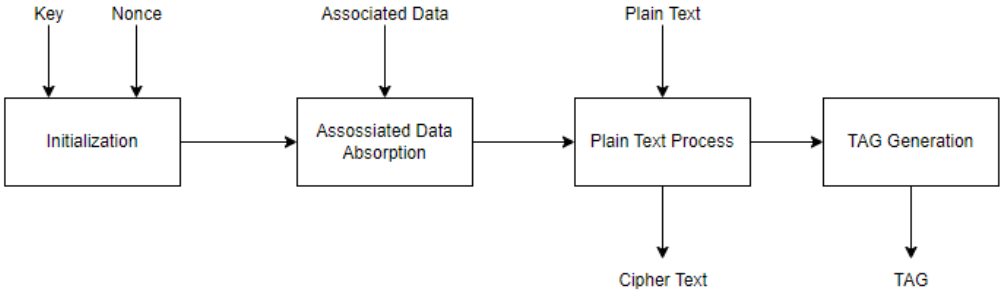


Figure 2.2 – AEAD top-level architecture.

Similar to Cheng *et al.* [Cheng et al., 2022], this work focuses on Ascon-128 AEAD. This decision is based on the frequent evaluation of Ascon-128 in the existing literature, which enables a comprehensive comparison of results. To implement AEAD, several parameters must be specified, such as the key length, denoted as $k \leq 160$ bits; the size of the data block (also referred to as the *rate* and symbolized by r); and the internal round numbers a and b, which are required for the permutation operation. The encryption algorithm receives a secret key (K), a plaintext (P) of arbitrary length, a 128-bit nonce (N), and associated data (A) of arbitrary length as inputs. The encryption process generates an output ciphertext (C), whose length mirrors is the same of the plaintext, alongside a 128-bit authentication tag (T). Equation (2.2) presents this process.

$$\varepsilon_{k,r,a,b}(K, N, A, P) = (C, T) \tag{2.2}$$

For the decryption process, Equation (2.3), the inputs include the key (K), nonce (N), associated data (A), ciphertext (C), and authentication tag (T). If the authentication tag passes verification, the output is the corresponding plaintext. If a discrepancy is detected in the tag authentication, an error symbol ($\perp$) is returned.

$$D_{k,r,a,b}(K, N, A, C, T) = (P, \perp) \tag{2.3}$$

2.2.1 Ascon Algorithm

Prior to 2023, in the context of IoT and constrained embedded systems, it was imperative to evaluate various LWC solutions to align with product requirements. In February 2023, NIST endorsed Ascon [Dobraunig et al., 2021] as a robust defense mechanism for data generated by small devices, publishing a standard that recommends this algorithm. Ascon is specifically engineered to safeguard information produced and transmitted by the Internet of Things (IoT), encompassing an array of sensors and actuators. According to NIST, this algorithm is deemed suitable for most IoT devices, thereby removing the need to select a new algorithm for each project.

Ascon encompasses three main branches: Ascon-128 and Ascon-128a, which constitute standard processes consisting of authenticated ciphers, and a novel variant known as Ascon-80pq, designed to enhance resistance against post-quantum computers. Additionally, there are Ascon-Hash and Ascon-HashA variants.

For this work, the focus is on Ascon-128, as it is the most commonly used function, and readily available implementations can be found. Table 2.6 outlines the notation employed for Ascon's interface, mode, and permutation:

Table 2.6 – Notation used for Ascon's interface, mode, and permutation.

Variable	**Meaning**
i	index i of state bits
K	Secret key K, K <= 160 bits
N, T	128-bit Nonce N and tag T
P, C, A	Plaintext P, ciphertext C, associated data A (in r-bit blocks Pi, Ci, Ai)
M, H	Message M, hash value H (in r-bit blocks Mi, Hi)
$\perp$	Error, verification of authenticated ciphertext failed
S	The 320-bit state S of the sponge construction
Sr, Sc	The r-bit rate and c-bit capacity part of the state S

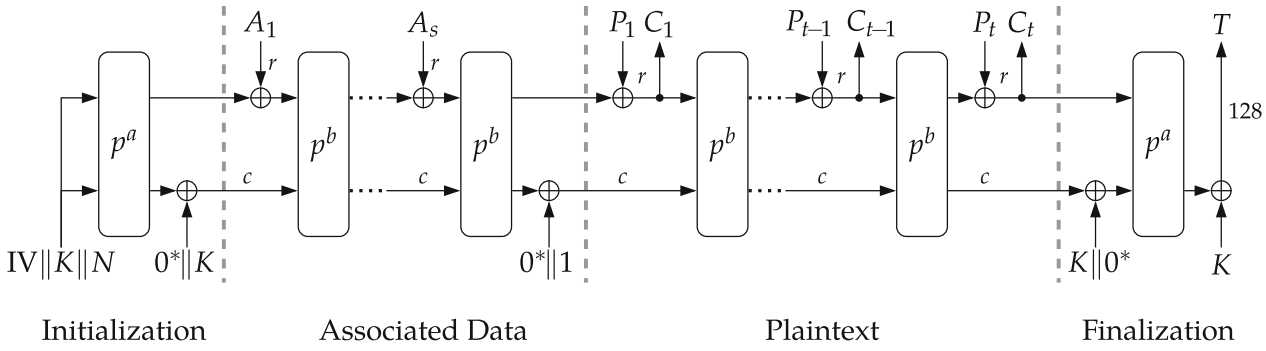
The encryption process for Ascon uses a duplex-sponge-based mode operation[1] and can be split into four steps, represented in Figure 2.3:

- **Initialization**, where the state is initialized with the key and nonce;
- **Associated Data** processing, where the state is updated with the associated blocks A_i;
- **Plaintext** processing, which injects the plaintext block P_i into the state and obtains the cipher blocks C_i;
- **Finalization** where the key is injected again, and the tag is obtained for authentication.

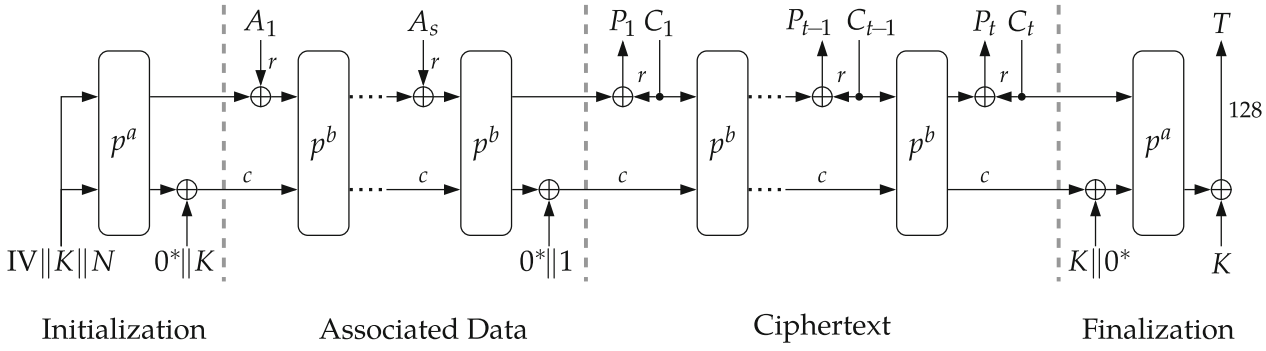
The 320-bit initial state of Ascon is composed of the secret key K with k bits, the nonce N with 128 bits, and an Initialization Vector (IV) specifying various algorithm parameters, including the key size k, the rate r, the initialization and finalization round numbers a, and the intermediate round number b, each represented as an 8-bit integer. Figure 2.4 illustrates the Initialization Vector (IV) and the initial state S.

During the initialization phase, the initial state undergoes a rounds of the round transformation p, followed by an XOR operation with the secret key K.

[1] In the context of cryptography, the sponge construction is a mode of operation, based on a fixed-length permutation (or transformation) and on a padding rule, which builds a function mapping variable-length input to variable-length output (source: https://keccak.team/sponge_duplex.html).



(a) Encryption $\mathcal{E}_{k,r,a,b}$



(b) Decryption $\mathcal{D}_{k,r,a,b}$

Figure 2.3 – Ascon's modes of operation [Dobraunig et al., 2021].

$$\mathrm{IV}_{k,r,a,b} \leftarrow k\|r\|a\|b\|0^{160-k} = \begin{cases} \texttt{80400c0600000000} & \text{for Ascon-128} \\ \texttt{80800c0800000000} & \text{for Ascon-128a} \\ \texttt{a0400c06} & \text{for Ascon-80pq} \end{cases}$$

$$S \leftarrow \mathrm{IV}_{k,r,a,b}\|K\|N$$

Figure 2.4 – Initialization Vector and initial state S.

During the associated data processing phase, the objective is to incorporate the associated data (A) into the state. To achieve this, the data is processed in blocks of r bits. In cases where the data size is not a multiple of r, a padding process is initiated to align the data with the rate r. The padding process appends a single 1 and the smallest number of 0s to the plaintext P such that the length of the padded plaintext is a multiple of r bits ($S \leftarrow S \oplus (0^{319}||1)$). Each block A_i is XORed with the first r bits of the state, and subsequently, this data undergoes b rounds of permutation denoted as p^b. If the block A_i is non-zero, it is also XORed with the state S_r. Following the processing of the associated data, a 1-bit domain separation constant is XORed with the state to finalize the operation.

The same process is executed for the Plaintext processing, splitting the plain text block in r bits, using the same rule for the padding to align with the rate bits (r). The encryption process starts in this step. In each iteration, one padded plaintext block P_i is xored to

the internal state's first r bits S_r using the permutation P^b, except for the last. In the whole plaintext step, after a permutation is completed, a ciphertext block is generated.

Subsequently, in the finalization step, the state undergoes transformation via permutation P^a, utilizing the secret key for a rounds. In this phase, the tag is generated from the 128 least significant bits of the state and the least significant bits of the key. Finally, the encryption algorithm yields the ciphertext along with the tag. During decryption, if the values match, the plaintext is returned.

The main structure, called permutation, has 3 main operations, detailed in Section 4.1.

2.2.2 AES-128 CCM

AES (Advanced Encryption Standard) [NIST, 2001a] is a widely adopted block cipher standard used for confidential transmission of information. Among its variants, AES-128 is particularly prevalent in embedded systems. An optimization technique employed in software implementations of AES is the utilization of T-Tables. This approach condenses an iteration in the AES cipher operation to 16 Look-Up Table (LUT) accesses and 16 bitwise XOR operations. However, this optimization comes at the cost of significant memory consumption, typically requiring 4 KB of memory.

The RISC-V *Zkne* extension introduces specialized instructions such as `aes32esmi` and `aes32esi`. These instructions enable the computation of T-Table entries directly in hardware, eliminating the need to store LUTs in memory. Additionally, these instructions optionally perform XOR operations on the current entry with previous entries for the same output column. This hardware acceleration results in improved performance, reduced code size, and enhanced energy efficiency by mitigating the overhead associated with memory accesses, as highlighted in [Saarinen, 2020, Gewehr and Moraes, 2023].

The CCM operation mode [NIST, 2007], illustrated in Figure 2.5, uses two simultaneous operation modes, CTR for encryption and CBC for generating authentication tags [NIST, 2001b]. The ciphertext is computed by encrypting the plaintext using the CTR mode, and the authentication tag is computed by encrypting the associated data and plaintext using the CBC mode. The CCM mode of operation is interesting, since it uses the underlying block cipher both for encryption and authentication, unlike other AEAD modes of operation such as GCM, which employ other means for generating authentication tags. Hardware acceleration of the underlying block cipher (here, AES-128) improves the efficiency of both encryption/decryption and authentication. Note that one AES-128 operation is performed for each block of associated data, while two AES-128 operations are performed for each block of plaintext.

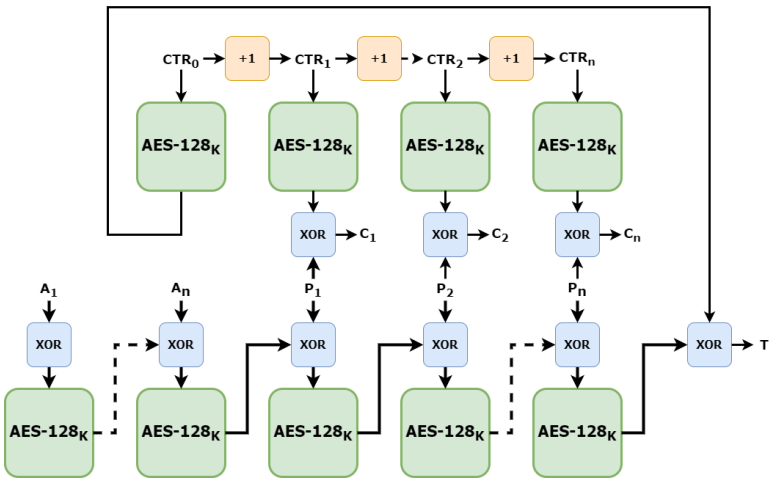


Figure 2.5 – The CCM mode of operation.

2.2.3 ChaCha20-Poly1305

The ChaCha cipher and Poly1305 authenticator [Nir and Langley, 2015] emerged as alternatives to AES, primarily due to their superior performance in pure software implementations, particularly on low-complexity microcontrollers. Among the variants, ChaCha20 is the most widely adopted. Internally, ChaCha operates on 32-bit variables, employing three main operations: addition, rotations by constant amounts, and XOR operations. Its internal state comprises 512 bits, divided into 16 32-bit variables. The quarter-round function, depicted in Figure 2.6, processes four variables simultaneously.

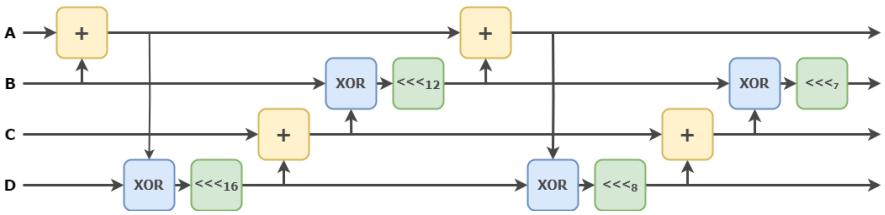


Figure 2.6 – ChaCha quarter-round function.

Hardware acceleration of ChaCha on 32-bit RISC-V cores via specialized algorithm-specific instructions has been briefly explored in [Marshall et al., 2021]. However, each group of Add-XOR-Rotate macro-operations requires three inputs, which cannot be directly implemented in a single RISC-V instruction, as each instruction should read at most two inputs from the register file. Introducing a third input would necessitate significant changes in the register file implementation, resulting in a significant area overhead. Attempting to merge two operations instead of three would not yield substantial gains. However, modest improvements can be achieved through the *Zbkb* extension, as the right-rotation operation can be executed using a single instruction (`rori`), as opposed to three RV32I instructions.

2.3 Hardware Acceleration

Most electronic devices employ a central processing unit (CPU) to perform various operations. Each instruction executed by the CPU typically corresponds to a fundamental operation, such as addition, subtraction, or data manipulation through loading and storing. Hardware acceleration, as outlined in [Dally et al., 2020], involves offloading specific functionalities from the CPU to dedicated hardware units, thereby enhancing data processing efficiency. An example of a hardware accelerator is the graphics processing unit (GPU), which specializes in graphical computations, thus relieving the CPU from such tasks.

Two primary approaches exist for integrating accelerators into computing systems: co-processor acceleration (**loosely coupled**) and **tightly coupled** acceleration. Each method presents distinct advantages and drawbacks.

A separate processing unit, loosely coupled with the processor, handles specific application domains in the co-processor approach. Communication between the CPU and co-processor typically occurs via an interface or bus, allowing the CPU to concentrate on executing general-purpose instructions. Co-processors offer reusability across different systems and facilitate straightforward integration or removal from the processor. However, the drawback lies in the communication overhead incurred when transmitting data to the co-processor.

On the other hand, tightly coupled acceleration involves augmenting the processor with additional instructions and sharing the processor units, such as the register file and the ALU.

Another critical aspect to consider is the granularity of the accelerator's operation. Accelerators can either independently execute extensive segments of algorithms, referred to as **coarse-grain** acceleration, or handle smaller portions of algorithms, named as **fine-grain** acceleration.

In the context of this work, we employ fine-grain tightly coupled acceleration. We compare this acceleration to a weakly coupled co-processor to show the difference between the methods. This decision is justified for embedded systems with limited memory and processing requirements due to its ability to optimize resource utilization and minimize overhead. By integrating closely with the processor and executing small portions of algorithms efficiently, fine-grain acceleration ensures that computational tasks are handled precisely while conserving memory and processing power, which is particularly advantageous for resource-constrained embedded environments.

3. RELATED WORK

Section 3.1 presents proposals that evaluate LWC algorithms and how these analyzes are performed and commonly used in research environments. Section 3.2 present work adopting encryption extensions in ISA RISC-V and also new instructions to help speed up existing algorithms. Section 3.3 summarizes the related work and positions this work in relation to the literature.

3.1 Assessments on LWC algorithms

Part of the literature has addressed the behavior of LWC algorithms in edge devices [Regla and Festijo, 2022, Beg et al., 2019, Akbas, 2019, Elaguech et al., 2019, Kotel et al., 2016, Lara-Nino et al., 2016]. However, the proposed approaches present drawbacks such as: (*i*) a low number or no end or edge device evaluated; (*ii*) a low number of assessed LWC algorithms; and (*iii*) none covered all PPA (Power-Performance-Area) metrics. In this regard, relevant recent works are reviewed to understand how assessments were performed in the literature and their applications to this work.

Ledwaba *et al.* [Ledwaba et al., 2018] investigate three widely adopted standard algorithms (AES, ECDSA, and SHA) that target end devices in the Internet of Energy (IoE). Their analysis primarily focuses on a single version of each algorithm without addressing LWC options. Ledwaba *et al.* utilize ST Microelectronics boards with Arm cores and evaluate the following parameters: performance, power consumption, and memory footprint. These Authors conclude that symmetric cryptography is viable for use in endpoint devices, a finding that does not extend to asymmetric cryptography such as ECDSA.

Kane *et al.* [Kane et al., 2020] evaluate various combinations of three low-power microcontrollers (ATmega328, STM32F103C8T6, and ESP8266) and three cryptographic algorithms (AES, ChaCha, and Acorn). The Authors assess power consumption, energy usage, execution time, and memory footprint, demonstrating the trade-offs that emerge among processor and cipher selections concerning the evaluated metrics.

Thakor *et al.* [Thakor et al., 2021] present a comprehensive review of fifty-two lightweight cryptography algorithms or their variations. The Authors describe each cipher and examine the performance of both hardware and software implementations for most algorithms. Additionally, the paper evaluates the relative security of the ciphers and their susceptibility to different types of attacks. In conclusion, the study recommends further research on substitution-permutation methods involving S-Boxes to develop new ciphers that balance cost, performance, and security.

More recently, Elsadek *et al.* [Elsadek et al., 2022b, Elsadek et al., 2022a] performed a PPA assessment of ASIC implementations for some of the NIST LWC standardization candidates [NIST, 2022]. Mohajerani *et al.* [Mohajerani et al., 2021] also proposed a hardware benchmarking of these LWC algorithms.

While these Authors focus on evaluating the hardware implementation of a few candidates, this dissertation differs from them by assessing the software and hardware integration of the winning LWC algorithm for NIST standardization, in addition to adding modifications in the RISC-V ISA for efficient use of the same.

3.2 Cryptography Extension in RISC-V ISA

IoT end devices require optimized performance and excellent energy efficiency. One technique used to enhance runtime performance and benefit efficiency in LWC algorithms is the bit manipulation extension. Sud *et al.* [Sud et al., 2022] present a simulation environment for the open-source RISC-V ISA that incorporates the base RISC-V ISA as well as the "bit manipulation" instruction set extension. They demonstrate the implementation of lightweight block ciphers LEA, SIMON, and SPECK on the simulator and evaluate their performance with and without bit manipulation instructions. Results show an average 38% improvement in clock cycles for lightweight ciphers when utilizing bit manipulation instructions, as well as providing insights by comparing their performance with other architectures, including ARM processors as reported in existing literature.

Cheng *et al.* [Cheng et al., 2022] conducted an evaluation of nine out of ten submissions from the final round of the NIST LWC selection process. The Authors focused on the design, implementation, and evaluation of Instruction Set Extensions (ISEs) for these algorithms. The approach involved developing independent ISEs for each algorithm, establishing a uniform structure while using the most recent submission of each algorithm as a reference. RISC-V served as the base instruction set architecture, and the assessments were performed based on latency measured in clock cycles and area cost measured in FPGA LUT usage. Note that Ascon, the main algorithm assessed in this work, was among the evaluated algorithms. However, it is important to mention that the proposed extension for Ascon bears a significant resemblance to the SHA-2 extensions, implying that its area cost is non-trivial.

On the other hand, Tehrani *et al.* [Tehrani et al., 2020] introduce a specialized execution unit for RISC-V processors that achieves significant performance gains in lightweight 64-bit block ciphers, with potential speedups of over a hundred compared to the reference architecture. The execution unit leverages five adaptable instructions specifically designed for the VexRiscv architecture, offering improved agility and performance for lightweight cryptographic implementations. Detailed architecture and implementation of these instructions

are provided, demonstrating their impact on execution speed, code size, and the potential for reduced instruction count. Additionally, the authors highlight the importance of Round Key specific registers for enhanced performance and security in a protected version under development.

As the ChaCha Algorithm was initially designed to be efficient in software, implementing it in hardware is challenging since the ChaCha function computes the values of 32-bit registers in two consecutive operations and can involve up to 3 elements. This is a problem since acceleration like RISC-V operates with a maximum of 2 registers. Therefore, [Marshall et al., 2021] proposes an ISE to support simple operations where the instruction has a maximum of 2 operators to interact with the RISC-V processor. The ISE then stores operands and intermediate results, executes in 1 clock cycle, and guarantees constant timing to prevent latency-based attacks.

3.3 Related Work Summary

Table 3.1 compares the reviewed proposals, organized by color coding. Green proposals focus on evaluating and comparing LWC methods, though they may not necessarily propose new algorithm acceleration techniques. Proposals marked in blue focus on proposing novel techniques or combining existing methods to enhance RISC-V algorithms, providing results.

This MSc work aims to enhance AEAD in low-power embedded systems by implementing hardware acceleration for Ascon, AES-128 in CCM mode, and ChaCha20-Poly1305 on the Ibex RISC-V platform. By integrating *Zbkb*, *Zkne*, and *Xascon* instruction set extensions, this work seeks to improve cryptographic efficiency with minimal power dissipation overhead, addressing the critical need for secure and efficient data processing in resource-constrained environments. This approach leverages the advancements in lightweight cryptography and RISC-V architectural extensions, contributing to the security of IoT devices and similar embedded systems.

Table 3.1 – Related works on Cryptography Extension and Assessment.

Work	Algorithm	Target	Goals
Security and Performance in IoT: A Balancing Act [Kane et al., 2020]	AES, ChaCha and Acorn	IoT devices	Evaluate cryptographic performance of IoT devices, including power, performance, energy, memory footprint on ATmega328, STM32F103C8T6 and ESP8266 low-power microcontroller devices.
Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities [Thakor et al., 2021]	52 LWC algorithms submitted to NIST competition	IoT devices	Comparison in terms of implementation cost, hardware, and software performances and attack resistance properties.
Performance evaluation and design considerations of lightweight block cipher for low-cost embedded devices [Kotel et al., 2016]	AES, PRESENT, Simon, Speck	Low-cost embedded Devices	Improve the software performance of lightweight block ciphers suited for low-cost embedded systems.
Hardware and Energy Efficiency Evaluation of NIST Lightweight Cryptography Standardization Finalists [Elsadek et al., 2022b]	10 LWC final round submissions, including Ascon	IoT devices	Design and evaluate the hardware of 10 LWC NIST final round using ASIC synthesis.
RISC-V Extension for Lightweight Cryptography [Tehrani et al., 2020]	PRESENT, GIFT, PRINCE, Midori, Twine, Skinny	RISC-V (VexRiscv)	Develop an execution unit for the RISC-V processor able to run the most common lightweight 64-bit block ciphers (**custom extensions**)
Evaluating Cryptographic Extensions on a RISC-V Simulation Environment [Sud et al., 2022]	LEA, SIMON, and SPECK	RISC-V with B extension (bitmanip)	Simulation environment comparing performance with and without bit manipulation instructions, with an average improvement of 38% in the total number of clock cycles to run lightweight ciphers with the B extension
RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography [Cheng et al., 2022]	10 LWC final round submissions, including Ascon	RISC-V with *Zbkb* extension (Rocket core, with 5-stage pipeline)	Implementation and evaluation of the ten LWC finalists using *Zbkb*
A lightweight ISE for ChaCha on RISC-V [Marshall et al., 2021]	ChaCha20	RISC-V for IoT edge devices	Propose a lightweight ISE to support ChaCha on RISC-V architectures
This MSc work	Ascon, AES-128 in CCM mode and ChaCha20-Poly1305	Ibex RISC-V with *Zbkb*, *Zkne* and Xascon extensions	Hardware Acceleration of Authenticated Encryption with Associated Data via RISC-V Instruction Set Extensions in Low Power Embedded Systems

4. AEAD HARDWARE ACCELERATION

This Chapter presents methods to accelerate AEAD algorithms.

- Section 4.1 provides information about how to accelerate the Ascon algorithm using an approved ISE, *Zbkb*, and provides insights into its functioning. The ChaCha20-Poly1305 1305 is also accelerated using the *Zbkb* ISE extension.
- Section 4.2 introduces a novel method for accelerating the Ascon algorithm, employing specialized instructions and implementation details.
- Section 4.3 provides an overview of the T-table technique for accelerating the AES algorithm by using the *Zkne* ISE.
- Section 4.4 presents an Ascon core tailored for loosely coupled acceleration to compare this method against ISE acceleration techniques.

4.1 RISC-V ZBKB

The *Zbkb* extension represents a subset of instructions tailored for scalar cryptography on RISC-V architecture, offering bit manipulation capabilities to increase the performance of operations at the bit level. The inclusion of *Zbkb* instructions is expected to enhance the performance of certain cryptographic algorithms, including SHA-3, SHA-2, and Ascon, particularly when executed on a 32-bit processor. These algorithms operate on 64-bit data words, while the processor operates on 32 bits. This section provides an overview of key aspects of the Ascon algorithm and explain how it benefits from the *Zbkb* extension.

4.1.1 Zbkb Extension and Relationship with Ascon

The main components of Ascon algorithm are the two 320-bit permutations p^a and p^b. Permutations iteratively apply an SPN-based (substitution-permutation-network) round transformation p that in turn, consists of three steps p_C, p_S, p_L:

$$p = p_L \circ p_S \circ p_C$$

p^a and p^b differ only in the number of rounds, where p^a has 12 rounds and is used on initialization and finalization, while p^b has 6 rounds and is used in associated data and plaintext states. The number of rounds a and the number of rounds b are tunable security parameters if you want to use Ascon-128a or Ascon-Hash.

For the round transformations, the 320-bit state S is split into five 64-bit registers words x_i, $S = x0 \parallel x1 \parallel x2 \parallel x3 \parallel x4$, as shown in Figure 4.1. This internal state is used to store the input data and perform the permutation. This arrangement of the data is used to facilitate the operations. Figure 4.1 presents the high-level view of each of the three steps.

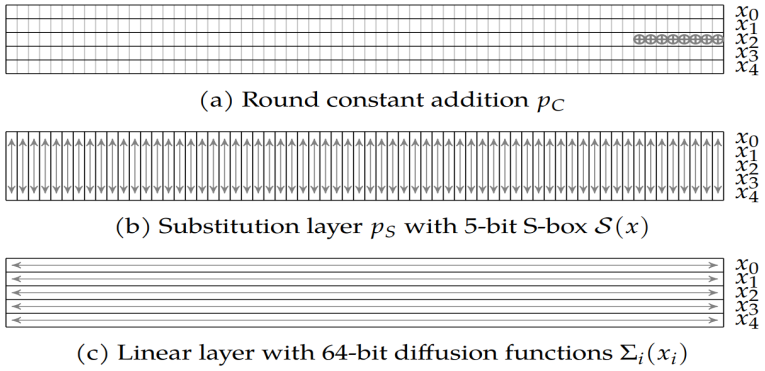


Figure 4.1 – Register words of the 320-bit state S and operations p_C, p_S, and p_L [Dobraunig et al., 2021].

As all Ascon algorithm operations work with 64-bit words, 32-bit processors cannot use just one register to store each word and operate on just one register, so each word is divided into two registers as shown in the Figure 4.2. With this separation of data into two registers, operations end up being more costly, since we need more instructions to manage the data between them. The bit manipulation extension makes this work easier by saving instructions to operate on the data.

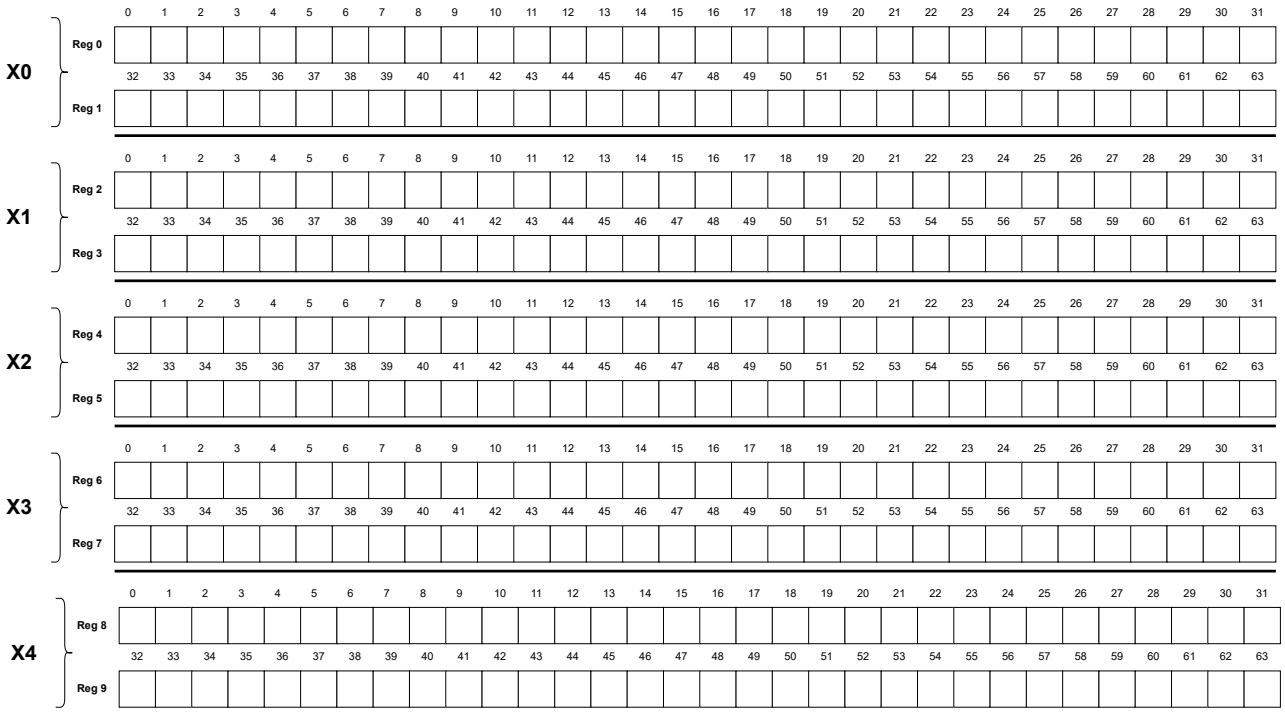


Figure 4.2 – Ascon State on 32-bit Register Processors.

The first step is the p_C, which is the function that adds a round constant to the state. This step is straightforward, as shown in Figure 4.1(a). It gets the word $x2$ of the state and performs a xor with a constant c_r based on the indices r and i. Indices r and i start from zero, and the algorithm uses $r = i$ for p^a and $r = i + a - b$ for p^b. Table 4.1 presents the constant c_r for the combinations.

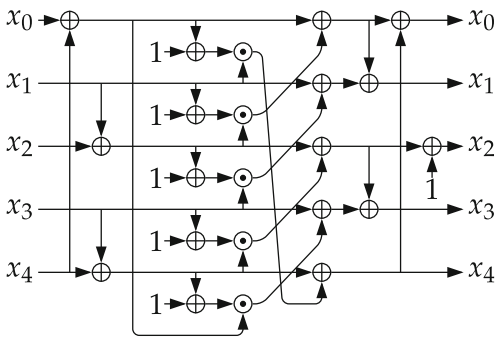
Table 4.1 – Constant c_r table.

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			00000000000000000f0	6	2	0	0000000000000000096
1			00000000000000000e1	7	3	1	0000000000000000087
2			00000000000000000d2	8	4	2	0000000000000000078
3			00000000000000000c3	9	5	3	0000000000000000069
4	0		00000000000000000b4	10	6	4	000000000000000005a
5	1		00000000000000000a5	11	7	5	000000000000000004b

The second step is the substitution layer p_S. The five words (x0...x4) from the state S are updated by the 5-bit S-box as defined in Figure 4.3. In this Figure, x corresponds to one of 32 possible values that a given bit slice across the five registers can assume, and $S(x)$ is the output of the lookup table (LUT). This LUT is implemented by the circuit depicted in Figure 4.4(a), using bit slicing. This method is faster than accessing a LUT in memory, as it comprises only logical gates and reduces memory access, consequently reducing memory leakages.

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

Figure 4.3 – Ascon 5-bit S-box S as a lookup table [Dobraunig et al., 2021].



(a) Ascon's 5-bit S-box $\mathcal{S}(x)$

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

(b) Ascon's linear layer with 64-bit functions $\Sigma_i(x_i)$

Figure 4.4 – Ascon substitution layer and linear diffusion layer [Dobraunig et al., 2021].

Code 4.1 presents the assembly code to execute the S-box implemented in the bitsliced form (Figure 4.4(a)) – adapted from: https://github.com/ascon/ascon-c/blob/main/crypto_aead/ascon128v12/asm_bi32_rv32b/ascon.S. In this example, a macro named *sbox*

was created to execute the logic and is called twice (lines 22-23). On lines 6 and 16, two instructions from the *Zbkb* extension, `orn` and `andn`, are used. In the assembly without the *Zbkb* extension, instead of `orn`, which performs the `or` operation and inverts the result in a single instruction, we would have to call a `xori x4, x4, -1`, followed by a `or x4, x4, x3`, spending one more instruction than in the *Zbkb* implementation. The same occurs for the `andn` instruction. This demonstrates how the *Zbkb* extension can optimize operations by reducing the amount of required instructions.

Lines 22 and 23 execute the low part and high part respectively of each internal state word, as shown in Figure 4.2 the internal state word is divided into two registers, e.g: x0 is stored in reg0 and reg1. Therefore, this macro is called 5 times, to execute throughout the internal state of Ascon.

Code 4.1 – Assembly of Ascon 5-bit S-box using the RISC-V assembly with Zbkb.

```
1  ## S-BOX MACRO DEFINITION FOR ZBKB INSTRUCTIONS
2  .macro sbox x0, x1, x2, x3, x4, t0, t1, t2
3      xor \t1, \x0, \x4
4      xor \t2, \x3, \x4
5      xor \t0, \x1, \x2
6      orn \x4, \x3, \x4
7      xor \x4, \x4, \t0
8      xor \x3, \x3, \x1
9      or \x3, \x3, \t0
10     xor \x3, \x3, \t1
11     xor \x2, \x2, \t1
12     or \x2, \x2, \x1
13     xor \x2, \x2, \t2
14     or \x0, \x0, \t2
15     xor \t0, \t0, \x0
16     andn \x1, \x1, \t1
17     xor \x1, \x1, \t2
18 .endm
19
20 ## MACRO USAGE
21     # s-box
22     sbox x0l, x1l, x2l, x3l, x4l, t0l, t0h, t1h
23     sbox x0h, x1h, x2h, x3h, x4h, t0h, x0l, t1h
```

The Linear Diffusion Layer constitutes the third step, providing diffusion p_L within each 64-bit word by applying a linear function, as shown in Figure 4.4(b). This function executes right rotation on the bits.

For the Linear Diffusion Layer function p_L, a right-rotation operation must be performed on 64-bit words. Since the processor uses 32-bit registers to store data, the rotation would require several instructions to load the data into registers and then carry out the rotations, executing XOR operations between the registers to complete this task. This process is optimized using the instructions `pack` , `packh` , `zip` and `unzip` . These instructions split the 64-bit word into two 32-bit registers, with one register storing the odd bits and the other storing the even bits, Figure 4.5 presents the functioning of the zip, unzip and pack func-

tions that will be used for a better understanding of what happens in each one, this example uses 8-bit registers for simplicity. This process is detailed in Code 4.2. Each rotation operation on the 64-bit word corresponds to 2 rotation operations in this setup. This optimization effectively reduces the computational overhead and improves the algorithm's performance.

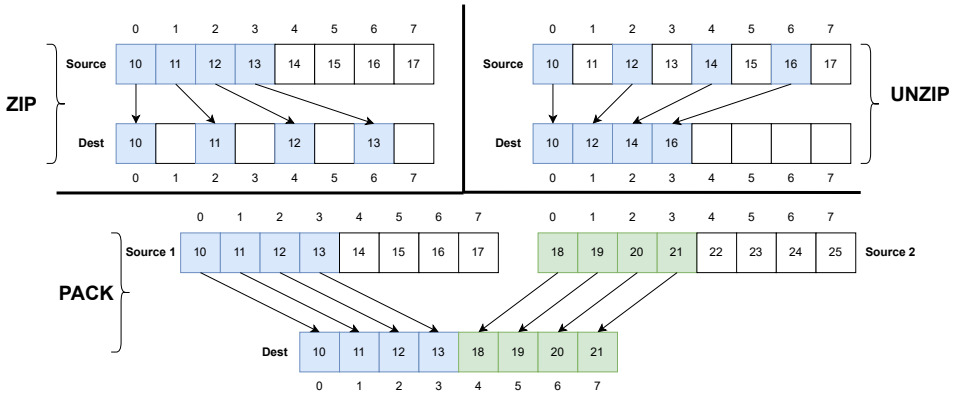


Figure 4.5 – Zip, Unzip and Pack instructions (Source: the Author).

Code 4.2 – Code with *Zbkb* instructions to rotate words larger than the processor word (e.g., 64-bit words in 32-bit processor). This example assumes a 16-bit word in an 8-bit processor.

```
// 16-bit word, stored in 8-bit registers, R1 and R2:
  R1: [0  1   2   3   4   5   6   7]
  R2: [8  9  10  11  12  13  14  15]

 // Unzip in R1 and R2 registers:
  R1:  [0   2   4   6   1   3   5   7]
  R2:  [8  10  12  14   9  11  13  15]

 // pack instruction using R1 and R2 - even bits
 R3:  [0   2   4   6   8  10  12  14]

 //  rotate right 4 bits right in R1 and R2
 R1:  [1   3   5   7   0   2   4   6]
 R2:  [9  11  13  15   8  10  12  14]

 // pack instruction using R1 and R2 - odd bits
 R4: [1   3   5   7   9  11  13  15]

 // rotate right 1 bit right in R3 and R4
 R3:   [14   0   2   4   6   8  10  12]
 R4:   [15   1   3   5   7   9  11  13]

 // pack instruction using R3 and R4
 R1:   [14   0   2   4  15   1   3   5]

 //  rotate right 4 bits right in R3 and R4
 R3:   [6   8  10  12   14   0   2   4]
 R4:   [7   9  11  13   15   1   3   5]

 // pack instruction using R3 and R4
 R2:  [6   8  10  12   7   9  11  13]

 // zip in R1 and R2
 R1:  [14  15   0   1   2   3   4   5]
 R2:  [6   7   8   9  10  11  12  13]

  // merging R1 and R2 we obtain a 2-bit rotate right
  14  15  0  1  2  3  4  5  6  7  8  9 10  11  12  13
```

4.2 RISC-V ISE XAscon

The XAscon ISE was introduced by [Cheng et al., 2022]. This ISE introduces two new instructions, as depicted in Figure 4.6. One instruction is dedicated to computing the high part of each of the five 64-bit word registers, while the other instruction handles the computation of the low part. Consequently, each sigma function can be computed in just two instructions, a significant reduction from the 16 instructions required in a RISC-V base implementation.

31 30	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
00	imm	rs2	rs1	111	rd	0101011	ascon.sigm.lo
01	imm	rs2	rs1	111	rd	0101011	ascon.sigm.hi

Figure 4.6 – Xascon Instruction definition.

As seen in Section 4.1.1, normally the substitution layer is implemented in a bit-sliced using logical ANDs, XORs and NOTs, whereas the permutation layer performs an operation of the form $x = x \oplus (x >> n) \oplus (x >> m)$ in each 64-bit word of the state, using Bit Interleaved, separating these 64-bits into two 32-bit registers, one with the odd position bits and the other with the even position bits. We show in the results section that this technique actually accelerates Ascon's linear functions, but it is worth highlighting that this gain occurs at the expense of conversions between the Bit Interleaved representation and the normal one.

As discussed in [Campos et al., 2020], an optimized implementation of the S-BOX necessitates 17 native RV32GC instructions. However, with the Zbkb extension, this count can be reduced to 15 instructions. Nevertheless, the extension proposes that with the inclusion of two custom instructions—one for the lower and another for the higher part of a 64-bit state word—a more substantial speedup can be achieved. Additionally, rotation values can be specified using immediate values. Consequently, the instruction count for the full permutation layer can be reduced from 80 (i.e., 16 per word) to just 10 instructions.

Equation (4.1) and Equation (4.2) display the lookup tables that need to be defined to facilitate the implementation of these instructions (see (Figure 4.4(b)). The position that will be utilized in the rotation is conveyed through the Immediate register, denoted as *imm*, with RTO_0 designated for the low part and RTO_1 designated for the high part.

$$ROT_0 = \{19, 61, 1, 10, 7\} \tag{4.1}$$

$$ROT_1 = \{28, 39, 6, 17, 41\} \tag{4.2}$$

Code 4.3 and Code 4.4 depict the steps that each of the instructions must follow during implementation. The operations remain consistent for both instructions, with the selection of the output being determined through a multiplexer to choose between directing the output to the high part or the low part.

Code 4.3 – ascon.sigma.lo rd, rs1, rs2, imm.

```
// Xascon sigma High instruction datapath
 x_hi = GPR[rs2]
 x_lo = GPR[rs1]
 x = x_hi || x_lo
 r = x ^ ( x >> ROT_0[ imm ] ) ^ ( x >> ROT_1[ imm ] )
 GPR[rd] = r_{31...0}
```

Code 4.4 – ascon.sigma.hi rd, rs1, rs2, imm.

```
 // Xascon sigma High instruction datapath
 x_hi = GPR[rs2]
 x_lo = GPR[rs1]
 x = x_hi || x_lo
 r = x ^ ( x >> ROT_0[ imm ] ) ^ ( x >> ROT_1[ Imm ] )
 GPR[rd] = r_{63..32}
```

4.3 RISC-V ZKNE

AES (Advanced Encryption Standard) [NIST, 2001a] is a block cipher standardized for NIST, and is widely used in various applications. Using just a block cipher algorithm like AES with ECB does not guarantee confidentiality, this means that with the same data input, we always generate the same output, which is why there are others modes of operation, which prevent attackers from inferring plaintext information by analyzing ciphertext patterns. Some of the operating modes that are used in AES are Electronic Code Book (ECB), Cipher Block Chaining (CBC), PCBC Mode (Propagating cipher-block chaining), CFB Mode (Cipher feedback), OFB Mode (Output feedback) and CTR (Counter) Mode.

For a fair comparison between Ascon and AES, it is necessary to use the AES-CCM [NIST, 2007] variation to generate authentication tags, providing authentication and confidentiality. AES employs both CBC and CTR in CMM as shown in Figure 2.5. The *Zkne*

extension of the RISC-V cryptography extension version 1.0.1 is used to accelerate AES functions. For this purpose, it is necessary to add 6 new instructions, as shown in Table 2.4

Internally, AES performs four operations by round: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. During the SubBytes operation we can use an optimization technique, where each S-BOX value is pre-calculated and stored in a Look-Up Table (LUT). The SBOX LUT comprises 256 bytes and is equivalent in size to 64 RV32I instructions.

An additional observation pertains to the potential for pre-computation not only of the SubBytes operation but also of the ShiftRows and MixColumns operations. These operations involve shifts and multiplications by constants, respectively. This pre-computation technique is referred to as the T-Table approach [Marshall et al., 2020]. The T-Table methodology entails the pre-computation of S-box substitution values for all conceivable 8-bit inputs. Subsequently, it generates tables (T-Tables) for the MixColumns operation, characterized by matrix multiplication within the AES algorithm. During the encryption process, the precomputed values from the T-Tables are utilized, thereby avoiding the need for recurrent computations.

The SBOX LUT comprises 256 bytes and is equivalent in size to 64 RV32I instructions, with a 1 to 1 byte mapping, while the T-Table maps from 1 to 4 bytes simplifying an entire round of AES to 16 LUT lookups and 16 XOR operations bit by bit. Equation (4.3) represents how this operation works.

$$T_0[x] = \begin{bmatrix} 02_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \\ 03_{(16)} \otimes Sbox(x) \end{bmatrix} \quad T_1[x] = \begin{bmatrix} 03_{(16)} \otimes Sbox(x) \\ 02_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} 01_{(16)} \otimes Sbox(x) \\ 03_{(16)} \otimes Sbox(x) \\ 02_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \end{bmatrix} \quad T_3[x] = \begin{bmatrix} 01_{(16)} \otimes Sbox(x) \\ 01_{(16)} \otimes Sbox(x) \\ 03_{(16)} \otimes Sbox(x) \\ 02_{(16)} \otimes Sbox(x) \end{bmatrix} \tag{4.3}$$

To implement this acceleration it is necessary to use the *Zkne* extension of the RISC-V cryptography package version 1.0.1 and use the *aes32esmi* and *aes32esi* instructions. The *aes32esmi* instruction can be interpreted as calculating T-Table entries in hardware and then XORing the current input with previous T-Table entries for the same output column. The *aes32esi* only performs a single SBOX and XOR lookup, used in calculating round keys and the last round of encryption, without MixColumns.

4.4 Ascon Crypto-Core

We can implement co-processors to perform a specific task so that processing is more efficient and the main processor can continue processing other data. The evaluated core is a version described in SystemVerilog made available by the Ascon team [Ascon, 2022b]. Figure 4.7 presents the interface of this Core and Table 4.2 contains a description of the interface signals.

To integrate this module with the RISC-V processor, we developed an AXI-Lite bus to send the data and configure the Ascon Core. At the RISC-V side, we added an AXI master interface and at Ascon core side a slave interface. We also reserved addresses ranges, so that the processor can access the accelerator as a memory-mapped I/O device.

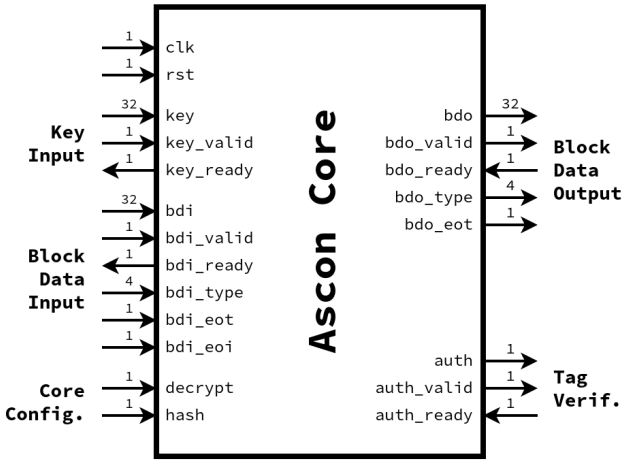


Figure 4.7 – Ascon Core Interface (Source: [Ascon, 2022b]).

Incorporating loosely coupled hardware accelerators with processors presents challenges and trade-offs. These challenges are related to managing the increased complexity associated with coordinating communication between the CPU and accelerators, handling data transfers, ensuring correct operation, and keeping compatibility between software and hardware. Additionally, modifications to the architecture may be necessary to accommodate the memory map and ensure seamless integration.

However, despite these challenges, hardware accelerators, holds the potential for significant efficiency gains. The CPU can focus on other tasks by offloading specific tasks to optimized hardware accelerators, improving overall system performance. Moreover, hardware accelerators can operate parallel with the CPU, enabling higher levels of parallelism and enhancing system throughput.

Despite the challenges to overcome, the potential efficiency gains offered by hardware accelerators make them a compelling option for enhancing the performance of hardware architectures in real-world applications.

Table 4.2 – Ascon Core interface signals

Name	Description
clk	Clock signal.
rst	Reset signal. Note: Synchronous active high.
key	Key data input.
key_valid	Key data is valid.
key_ready	Ascon core is ready to receive a new key.
bdi_data	Block data input (BDI).
bdi_valid	BDI data is valid.
bdi_ready	Ascon core is ready to receive data.
bdi_eot	The current BDI block is the last block of its type.
bdi_eoi	The current BDI block is the last block of input other than the tag segment.
bdi_type	Type of BDI data. See rtl/config_core.sv.
decrypt	0=Encryption, 1=Decryption.
hash	0=Encryption/Decryption, 1=Hash.
bdo_data	Block data output (BDO).
bdo_valid	BDO data is valid.
bdo_ready	Test bench is ready to receive data.
bdo_type	Type of BDO data. See rtl/config_core.sv.
auth	1=Authentication success, 0=Authentication failure.
auth_valid	Authentication output is valid.
auth_ready	Test bench is ready to accept authentication result.

4.5 Implementation

This section presents the modifications made to the IBEX environment to include ISEs. Section 4.5.1 presents the changes to add the *Zbkb* extension, Section 4.5.2 presents the changes to add *Xascon* and Section 4.5.3 presents the changes to add the Ascon core loosely coupled to the processor.

4.5.1 Changes for *Zbkb*

To include the *Zbkb* extension, modifications were made in the Decoder and Alu blocks, in addition to the system flags mentioned in Table 2.5. These changes were implemented within the Ibex framework to include the new instructions.

The first block modified was the **ibex_decoder**, where the processor interprets the instructions. Figure 4.8 presents the binary code for the *zip rd*, *rs* instruction. The object code of this instruction is embodied in bits {31,25}, {24,20}, {14,12}, and {6,0}. The instruc-

tion is only decoded if the system flag is set up to include these instructions. Otherwise, the "*illegal_insn*" flag is be set to 1, signaling an error in the decoding process.

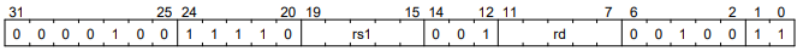


Figure 4.8 – zip instruction specification

Another block requiring modification is **ibex_alu**, responsible for implementing the logic for the instructions. For example, when the decoder identifies the instruction as ALU_ZIP, the ALU block places all odd bits in the lower halves of the word and even bits in the upper halves. Modifications for the remaining instructions of the *Zbkb* extension are required, such as `unzip`, `brev8`, and `packh`.

The **ibex_pkg** file is responsible for establishing definitions and parameters, as well as creating the system flags to manage the hardware during synthesis. In this file, the flag **RV32BCrypto** was added to group the instructions for the *Zbkb* that were implemented, along with some instructions from *zbb* that were already implemented.

Through this approach, the hardware is synthesized with the required instructions, occupying the smallest possible area. This facilitates the evaluation of the trade-off between speedup and area utilization.

To setup correctly the use of these instructions, We need to notify the GCC which extensions should be used. Then we should pass all extensions split by an underscore like *rv32imc_zicsr_Zbkb*, to compile and run without errors.

4.5.2 Changes for Xascon

A few changes were needed to add the Xascon extension. First, we modify the decoder block so that the processor understands that it is a valid instruction, as shown in Figure 4.6, which presents the binary code of the instruction. Bits {14,12} are used to identify which is the Xascon extension and bits {31,25} identify which of the 10 instructions should be decoded.

Figure 4.9 illustrates the ID/EX pipeline stage datapath featuring the XAscon functional unit. This block only adds the logic of the sigma functions which was explained in Section 4.2. This unit operates entirely in a combinational manner, with all XAscon instructions executing within a single clock cycle. As sigma functions involve rotations by static values, this implementation is very efficient on hardware. Unlike other units that possess well-defined datapaths, the XAscon unit is essentially comprised of XOR operations, responsible for computing sigma functions. A multiplexer (MUX) is employed to select either the high or low part of the sigma function as required.

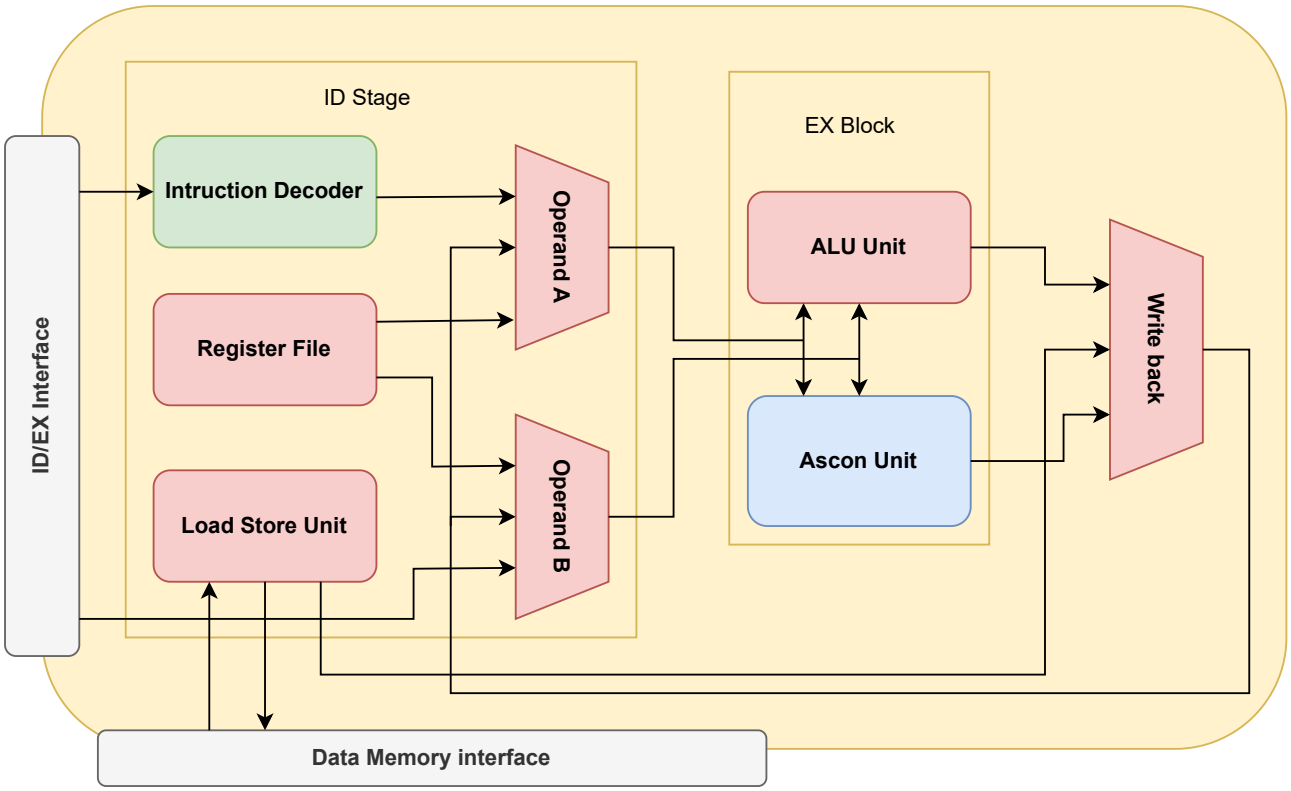


Figure 4.9 – Ibex ID/EX pipeline stage datapath with XAscon functional unit (Source: the Author).

Finally, we need to change GCC to include these instructions by modifying (riscv.md) in the GCC source code, defining the new instruction pattern that matches the machine code that ISA expects, as they are not standardized and the compiler cannot process without these modifications.

4.5.3 Changes for Ascon core

The initial phase involved the development of the AXI bus, resulting in the creation of a module named **bus_axi.sv**, which was designed according to the AXI bus specification [Arm, 2023]. A wrapper was designed to facilitate integration with the Ascon encryption core. This wrapper encompasses the instantiation of the Ascon core, the incorporation of memories for the AXI bus connectivity, and the establishment of control logic that manages data reception and core operation.

Subsequently, on the RISC-V processor side, it became imperative to embed the AXI bus module within the **ibex_simple_system** block. Additionally, modifications were applied to the **ibex_load_store_unit.sv** file to enable the processor to communicate with the bus, enabling I/O transactions. With these changes, it was also necessary to change the part of the memories that use the bus, changing the **prim_generic_ram_2p.sv** block.

5. RESULTS

This Chapter presents the results obtained in this work.

- Section 5.1 provides a brief comparison between Ascon and other lightweight encryption algorithms, aiming to assess the positioning of this new standard to existing LWC algorithms.
- Section 5.2 evaluates the performance of the Ascon algorithm. This evaluation includes performance metrics of various implementations, including pure software implementations, to facilitate an assessment of acceleration for each implementation.
- Section 5.3 compares AEAD and Ascon algorithms, specifically AES-128 in CCM mode and ChaCha20-Poly1305 in the IBEX environment. This section also delves into the analysis of the central operations of each algorithm to scrutinize the acceleration results for both the entire algorithm and only the main kernels.
- Section 5.4 presents the results of the Ascon core separetly, once it is an implementation external to the processor.
- Section 5.5 presents final remarks related to the results obtained in this Chapter.

The results presented in Section 5.2 and Section 5.3 were obtained for two distinct scenarios typically found in real-world applications, as proposed by [Cardoso dos Santos et al., 2020]. These scenarios involve different packet sizes, with the first scenario representing Zigbee and the second representing IPV6. This approach allows for an analysis of algorithm behavior concerning both shorter and longer messages, aiming to identify any operational variations based on instructions or among different algorithms.

Compilation was conducted using GCC version 12.2.0 with the -Os flag in an IBEX RISC-V Environment.

5.1 Assessment of LWCs

Table 5.1 presents the performance of several LWC algorithms, utilizing the same implementation as detailed in [Moura et al., 2023]. It is worth noting that this comparison is not entirely fair, as the Ascon algorithm is an AEAD algorithm, whereas the other algorithms listed may not necessarily provide authentication alongside encryption. Nevertheless, this table provides insight into how Ascon positions itself alongside other LWC algorithms in terms of performance.

All results were obtained under the same scenario: one full-block cipher operation. In this scenario, 128-bit input data were encrypted and subsequently decrypted. This approach allows for an estimate of the processing cost associated with a complete block for each LWC algorithm. These implementations can be found in https://github.com/nicolasMoura25/cryptography-algorithms.

Table 5.1 – Lightweight performance figures on IBEX base (128-bit message)

Performance Counters	Ascon	ARIA	GOST	HIGHT	IDEA	NOEKEON	SEED	SIMON
Clock cycles	**32,878**	59,975	87,868	67,846	47,585	28,039	32,871	41,252
Instructions retired	**21,994**	42,026	59,642	50,972	29,037	18,803	22,700	31,999
Instructions fetched (KB)	**12.89**	114.75	172.83	153.45	78.06	51.28	63.87	102.76
Loads from memory (KB)	**5.05**	7.61	5.01	5.78	2.81	2.10	2.93	2.90
Stores to memory (KB)	**8.49**	5.10	3.53	3.45	2.46	2.52	1.42	1.82

For the sake of perspective, we observe that the Ascon stands out in performance. Despite the additional overhead of encrypting the associated data and the plaintext, Ascon was faster than most algorithms, only behind NOEKEON, with similar performance to SEED. This is a relevant result, considering that Ascon is an AEAD algorithm, requiring encryption of the associated data and the plaintext, unlike the other LWC algorithms.

5.2 Ascon Performance Evaluation

Table 5.2 assess the performance obtained for each of the analyzed extensions and the performance results achieved through the software execution (average values of the Table row “clock cycles”):

- *Zbkb*: 15.9% better performance (15.9% and 15.9%) than the software;
- *Xascon*: 52.9% better performance (52.55% and 53.28%) than the software;
- *Zbkb* + *Xascon*: 59.71% better performance (59.11% and 60.31%) than the software implementation.

Hardware acceleration leads to a reduction in the number of retired and fetched instructions. The count of store-to-memory operations remains unaltered since all implementations’ input data and operations remain the same. However, memory loads increased in the *Zbkb* implementation due to adopting bit-interleaved representation. Notably, this increase in memory loads is not replicated in the *Zbkb*+*XAscon* implementation. This is because the increase in loads is specific to the layer replacement phase, which is executed by *XAscon* when both methods are combined. Therefore, when *Zbkb* and *XAscon* are utilized

Table 5.2 – Ascon performance figures.

Performance Counters	**Zigbee packet (A = 25 bytes, P = 86 bytes)**				**IPv6 packet (A = 40 bytes, P = 1224 bytes)**			
	Software	*Zbkb*	Xascon	*Zbkb*+Xascon	Software	*Zbkb*	Xascon	*Zbkb*+Xascon
Clock cycles	**14,385**	**12,097**	**6,825**	**5,881**	**128,471**	**107,921**	**60,011**	**50,979**
Instructions retired	13,734	8,939	6,174	5,230	123,381	79,887	54,921	45,889
Instructions fetched (KB)	53.086	34.326	23.555	19.867	478.639	308.428	211.217	175.936
Loads from memory (KB)	0.222	0.448	0.222	0.222	1.342	3.409	1.342	1.342
Stores to memory (KB)	0.190	0.190	0.190	0.190	1.295	1.295	1.295	1.295

together, the increase in memory loads attributable to bit-interleaved representation is not duplicated.

5.3 Assessment of Authenticated Encryption with Associated Data Algorithms

Table 5.3 compares AES-128 in CCM mode, Ascon, and ChaCha20-Poly1305 algorithms executed purely in software within the IBEX RISC-V environment. Ascon emerges as the faster algorithm in both scenarios. However, it is noteworthy that ChaCha20-Poly1305 exhibits a minimal difference in the IPV6 scenario. This suggests that ChaCha20-Poly1305 may possess a more efficient architecture for processing large messages than small ones. In the Zigbee scenario, Ascon in pure software demonstrates a speed advantage of approximately 15 times over AES-CCM. Conversely, compared to ChaCha20-Poly1305, Ascon is only around 1.5 times faster. Ascon presents significantly fewer loads and stores than the other two algorithms, directly impacting its performance.

Table 5.3 – AEAD performance figures – software execution.

Performance Counters	**Zigbee packet (A = 25 bytes, P = 86 bytes)**			**IPv6 packet (A = 40 bytes, P = 1224 bytes)**		
	AES-128 CCM	Ascon	ChaCha20-Poly1305	AES-128 CCM	Ascon	ChaCha20-Poly1305
Clock cycles	**218,582**	**14,385**	**21,913**	**2,171,290**	**128,471**	**129,660**
Instructions retired	133,834	13,734	16,615	1,325,726	123,381	100,666
Instructions fetched (KB)	309.541	53.086	54.049	4,112.371	478.639	330.445
Loads from memory (KB)	50.142	0.222	4.392	493.120	1.342	20.497
Stores to memory (KB)	30.818	0.190	4.204	303.554	1.295	16.341

Table 5.4 presents the results related to the hardware acceleration, offering insight into which algorithm attained the best performance and which experienced the most significant gains. Notably, the ChaCha20-Poly1305 algorithm achieved the lowest speedup, with the *Zbkb* extension yielding only marginal performance improvements (5.2%). While AES had an improvement of 95.1% and Ascon 60.3%. But it is worth noting that AES had more room for improvement, and with the acceleration, the final result was very close to the Ascon result purely in software.

For a more detailed analysis of the AES-128 in CCM mode and Ascon algorithms, Table 5.5 and Table 5.6 present results for the main operations of the algorithms, focusing

Table 5.4 – AEAD performance figures – hardware acceleration.

Performance Counters	**Zigbee packet (A = 25 bytes, P = 86 bytes)**			**IPv6 packet (A = 40 bytes, P = 1224 bytes)**		
	AES-128 CCM	Ascon	ChaCha20-Poly1305	AES-128 CCM	Ascon	ChaCha20-Poly1305
Clock cycles	**11,119**	**5,881**	**20,953**	**107,176**	**50,979**	**122,940**
Instructions retired	7,414	5,230	14,695	73,752	45,889	87,226
Instructions fetched (KB)	25.568	19.867	47.250	258.230	175.936	282.865
Loads from memory (KB)	3.698	0.222	4.392	34.993	1.342	720.497
Stores to memory (KB)	0.818	0.190	4.204	5.429	1.295	16.341

on the AES-128 block cipher and the permutation operation (*pb*) for Ascon. From these results, the AES-128 in CCM mode with the *Zkne* extension exhibits superior performance when executed on an extended IBEX core for Associated data, but lose for Xascon+*Zbkb* for plaintext, for long messages like IPv6 the plaintext there are more impact. It's important to note that this observation is aligned from the findings in Table 5.4, primarily due to the inclusion of data I/O operations and the relative optimization levels of the AES-128 software implementation in CCM mode versus the Ascon baseline implementation. Another point is that while AES in CCM mode relies on memory operations for temporary values, Ascon keeps these values within the register file, potentially contributing to its overall efficiency.

Table 5.5 – Associated Data & Plaintext Performance (smaller is better).

		Ascon				**AES**
Metric		**SW**	*Zbkb*	**Xascon**	*Zbkb*+**Xascon**	**AES-128 Zkne**
AD	Cycles/byte	89.875	75.375	37.375	34.375	**18.563**
	Instr./byte	88.125	56.875	35.625	32.625	**14.5625**
P	Cycles/byte	89.875	75.375	37.375	**34.375**	37.125
	Instr./byte	88.125	56.875	35.625	32.625	**29.125**

Table 5.6 provides insight into the CPI (Cycles Per Instruction) required to execute the algorithms. AES-128 exhibits a higher CPI because it needs to load the round key at each iteration. In contrast, Ascon avoids memory operations, except in the *Zbkb* case. We can notice that the implementation with just the ISE *Zbkb* causes the number of memory accesses to increase, going from a CPI of 1.02 to 1.32.

Table 5.6 – Core operation profiling – (Clock cycles, Instr. retired), CPI.

	Ascon				**AES**
Step	**SW**	*Zbkb*	**Xascon**	*Zbkb*+**Xascon**	**AES-128 Zkne**
SBOX	(204, 204), 1	(180, 180), 1	(204, 204), 1	(180, 180), 1	-
Linear	(480, 480), 1	(365, 238), 1.53	(60, 60), 1	(60, 60), 1	-
Other	(35, 21), 1.66	(58, 37), 1.57	(35, 21), 1.67	(35, 21), 1.67	-
Total	(719, 705), 1.02	(603, 455), 1.32	(299, 285), 1.05	(275, 261), 1.05	(297, 233), 1.27

Table 5.7 presents the synthesis data of the baseline Ibex processor and its extensions, targeting a high-density 8-track cell library for a 28nm FDSOI process from ST Microelectronics. These results were obtained using the GENUS Version 2112 tool, em-

ploying the PLE flow, and considering a worst-case PVT corner timing of a slow process (0.75 V at 125℃). All netlists have a slack time of 0 ns for fair comparison. The analysis indicates that the *Zbkb* extension exhibits the lowest overhead when considering only area overhead. However, when both area and performance are considered, AES-128 in CCM mode demonstrates a smaller overhead compared to the accelerated implementations of Ascon and ChaCha20-Poly1305.

Table 5.7 – Synthesis results for baseline and extended Ibex cores.

Synthesis results	**Baseline**	*Zbkb*	**Xascon**	**Xascon+***Zbkb*	**Zkne**
Cell Area (μm^2)	11,238	11,307	12,210	12,148	11,709
Net Area (μm^2)	6,992	5,142	8,181	5,957	6,261
Total Area (μm^2)	18,230	16,449	20,391	18,105	17,970
Cell Instance Count	**10,289**	**11,769**	**11,010**	**12,256**	**11,458**
Equiv. NAND2 gates	**34,433**	**34,642**	**37,408**	**37,219**	**35,873**

In conclusion, the hardware acceleration for AES-128 CCC mode and Ascon resulted in performance improvements of 95.1% and 60.3% (clock cycles) at the cost of 4.2% and 8.1% (equivalent NAND2 gates). Such results demonstrate the benefits of using ISEs to accelerate these algorithms.

5.3.1 Memory Usage Evaluation

Figure 5.1 presents the memory usage for AES-128 in CCM mode and Ascon. The GCC *fstack-usage* flag is used to obtain stack usage and GNU *nm* for functions and static data sizes. The total memory used for the AES-128 dropped from 2,437 to 1,563 bytes (35.9%) when using ISE *Zkn*. Ascon had its smallest reduction when combining the *Zbkb*+*Xascon*, from 1,694 bytes to 1,198 (29.3%). It is worth mentioning that these values are the sum of Max stack usage + Code size + Static data size. Table 5.8 presents raw results for memory consumption.

Table 5.8 – Raw results for memory consumption

Memory results	**AES-128 SW**	**AES-128 Zkne**	**Ascon SW**	**Ascon** *Zbkb*	**Ascon Xascon**	**Xascon+***Zbkb*
Max Stack Usage (Bytes)	337	257	96	96	96	96
Code Size (Bytes)	1800	1296	1598	1434	1310	1102
Static Data Size (Bytes)	300	10	0	25	0	0

The values for the ChaCha20-Poly1305 were not presented due to the complexity of obtaining precise measurements. In general terms, the memory usage for ChaCha20-Poly1305 is reported to be more than twice the size of AES-128 CCM. Additionally, the comparison highlights that AES benefits more than Ascon in terms of memory usage when accelerated.

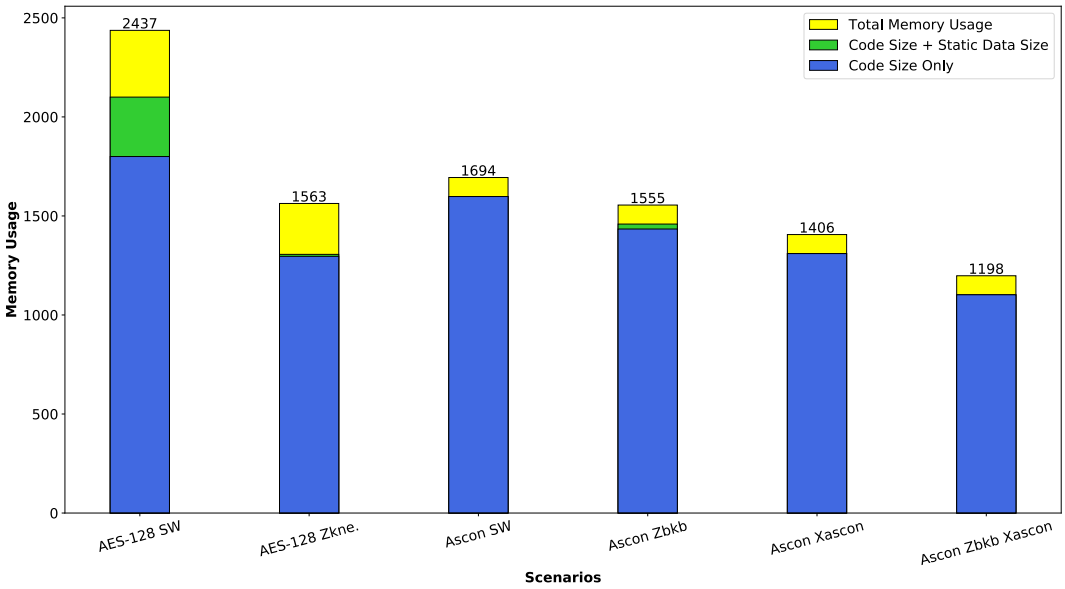


Figure 5.1 – Memory usage for AES-128 in CCM mode and Ascon.

5.3.2 Energy Consumption Evaluation

Energy consumption is evaluated using the CACTI tool [Balasubramonian et al., 2017] for memory, and gate-level simulations for logic. These results are obtained from a nominal PVT corner of a typical process (0.9 V at 25℃). For a 16KB dual-port SRAM with low-power bit cells in 28 nm technology, CACTI reports energy consumption of 63.362 fJ/bit for reads and 41.436 fJ/bit for writes.

Energy consumption results are depicted in Figure 5.2 – the software AES-128-CCM result is omitted due to its high energy cost. Table 5.9 presents the raw results for energy consumption. The energy consumption in RISC-V systems is predominantly influenced by the core rather than memory and I/O operations. With ISEs, the most efficient implementation is Ascon with *Zbkb* and *XAscon* for both scenarios.

Table 5.9 – Raw results for energy consumption for IPv6 scenario.

Memory results	**AES-128 SW**	**AES-128 Zkne**	**Ascon SW**	**Ascon** *Zbkb*	**Ascon Xascon**	**Xascon+***Zbkb*	**ChaCha20 SW**	**ChaCha20** *Zbkb*
Data IO Energy (*nJ*)	324.950	20.006	1.136	2.209	1.136	1.136	16.186	16.186
Instruction IO Energy (*nJ*)	2134.574	134.037	248.443	160.093	109.635	91.321	171.521	146.824
Core Energy (*nJ*)	10348.913	592.391	1063.259	674.785	453.604	359.809	791.294	648.511
Total Energy (*nJ*)	**12808.436**	**746.434**	**1312.838**	**837.087**	**564.374**	**452.266**	**979.002**	**811.521**

The AES algorithm reduced the total energy from 12,808.436 nJ to 746.434 nJ using the ISE Zkne (94.2%). While Ascon reduced 65.6% when comparing Ascon in software with Ascon *Zbkb*+Xascon.

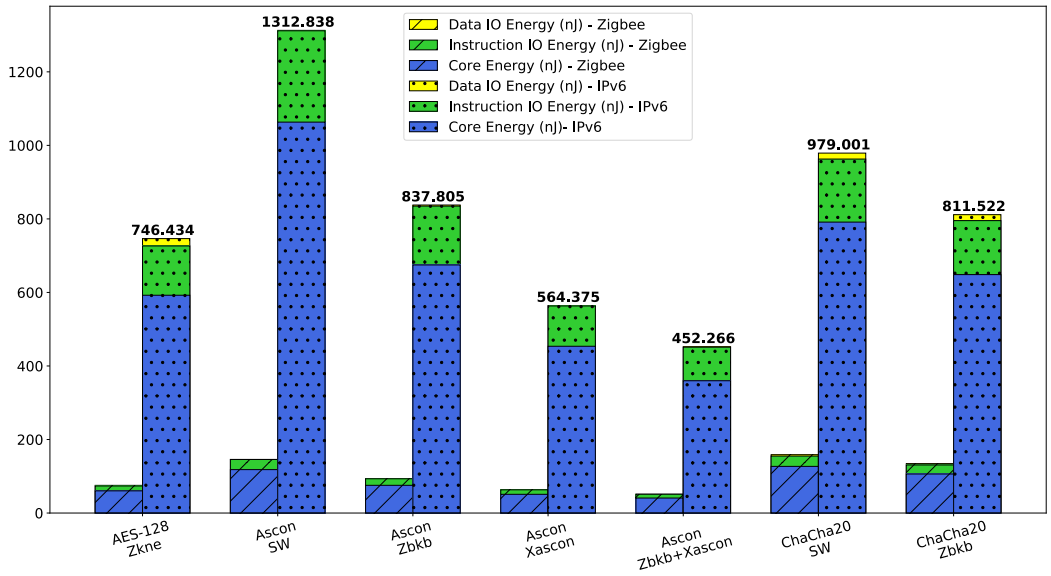


Figure 5.2 – Energy Consumption Evaluation - Zigbee - left, IPv6 - right

5.4 Ascon core

The results for the Ascon core are unavailable due to an issue with the hardware provided by the Ascon team. The provided model only works for a single input message size, leading to failure when simulating Zigbee and IPV6 scenarios. Since the objective of the work did not involve implementing or modifying the Ascon core, this aspect is included as future work. It is important to note that the AXI bus is functional, and all modifications are available in the RTL code. This means it is only necessary to integrate a working Ascon core and configure it via the bus to be functional.

Table 5.10 presents synthesis results related to the Ascon core. The equivalent number of NAND2 gates for this core is 5,963, while the overhead for IBEX with *Zbkb* and *XAscon* is 2,786 (Table 5.7). The smaller area overhead with tightly coupled acceleration is due to the resource sharing of the algorithm with the processor units.

From Table 5.4, IBEX with *Zbkb* and *XAscon* takes 41.6 clock cycles per byte to encrypt or decrypt a message. On the order side, the Ascon core took around 140 clock cycles to complete the cryptography process over a 192-byte message (1.4 clock cycles per byte). The performance overhead of acceleration using ISEs is due to the Von Neumann process (fetch-decode-execute instructions). On the other hand, bus transactions may hinder the performance of a dedicated core, penalizing its performance.

Table 5.10 – Synthesis results for Ascon Core

Synthesis results	**Ascon Core**
Cell Area (μm^2)	1,946
Net Area (μm^2)	1,207
Total Area (μm^2)	3,153
Cell Instance Count	1,796
Equiv. NAND2 gates	5,963

5.5 Final Remarks

In summary, comparing software-only and accelerated implementations depends on how the algorithm was designed. For example, in the case of ChaCha20-Poly1305, the implementation did not justify the addition of the *Zbkb* extension, as the performance increased by only 5%. On the other hand, Ascon exhibited a significant increase in performance by 2.45x when accelerated. Memory and energy consumption also benefit from this acceleration. The maximum area cost for implementing Ascon with *Zbkb* + *XAscon* was 8.1%. Table 5.11 summarizes the results.

Table 5.11 – Summary of the results related to the evaluated AEAD algorithms (IPV6 scenario).

Performance Figure	Implementation	AES-128 CCM Mode	Ascon-128	ChaCha20-Poly1305
Perfomance (clock cycles)	Software	2,171,290	128,471	129,660
	Hardware (best result)	107,176	50,979	122,940
	Gain	**95.1%**	**60.3%**	**5.2%**
Energy (nJ)	Software	12,808	1,313	979
	Hardware (best result)	746	452	811
	Gain	**94,2%**	**65.6%**	**17.2%**
Memory (bytes)	Software	2,437	1,694	NA
	Hardware (best result)	1,563	1,198	NA
	Gain	**35.9%**	**29.3%**	**NA**
Area (NAND2 gates)	Software	34,432	34,432	34,432
	Hardware (best result)	35,873	37,219	34,641
	Gain	**-4,2%**	**-8,1%**	**-0,6%**

6. CONCLUSION AND FUTURE WORKS

This MSc thesis conducted a PPA (Power, Performance, and Area) evaluation of three AEAD algorithms – Ascon, AES-128 in CCM mode, and ChaCha20-Poly1305, considering the base software execution and Instruction Set Extensions (ISEs) in the low-complexity RISC-V Ibex core. One of this work's contributions is the publicly available repository with all implementations: https://github.com/cggewehr/RISCV-crypto.

Despite efforts to ensure fairness in the evaluations, numerous factors impact the results. This was evident when comparing the performance of all encryption algorithms in two different scenarios: Zigbee and IPV6. In this comparison, Ascon emerged as the superior algorithm, requiring only 14,385 cycles for the Zigbee scenario and 128,471 cycles for the IPV6 scenario. However, when focusing solely on the main operations (such as the p^b permutation and the AES-128 block cipher), AES-128 in CCM mode demonstrated better acceleration, requiring 18,563 cycles for computing Associated data and 37,125 cycles for plaintext, while Ascon needed 34,375 cycles for both processes in its most accelerated version. Thus, it is expected that for larger messages, i.e., larger plaintext, the AES-128 in CCM mode may be faster than the Ascon algorithm.

Results showed a performance gain of 95.1%, 60.3%, and 5.2%, along with an increase in the energy efficiency of 94.2%, 65.6%, and 17.2%, for AES, Ascon, and ChaCha20-Poly1305, respectively. Area overheads were observed to be up to 9%. Additionally, improvements in memory usage (considering code size, static data size, and stack usage) of 35.9% (AES in CCM mode) and 29.3% (Ascon) were measured.

In conclusion, adopting an extension to accelerate an algorithm can yield considerable advantages, particularly when a device dedicates substantial time to data encryption. It is important to note that not all algorithms exhibit impressive performance enhancements, such as AES and Ascon; however, the justification for acceleration may lie in the resultant improvements in energy efficiency. For example, the ChaCha20-Poly1305 algorithm displayed a modest performance increment of 5.2%, but with a 17.2% enhancement in energy efficiency. The silicon area overhead, quantified at 8.1% in a worst-case scenario, constitutes a negligible trade-off when weighed against the performance and energy efficiency benefits. This suggests that devices with limited resources that need to encrypt data benefit significantly from such extensions.

The results of this work were published in:

Hardware Acceleration of Authenticated Encryption with Associated Data via RISC-V Instruction Set Extensions in Low Power Embedded Systems.
Gewehr, Carlos; Moura, Nicolas; Luza, Lucas; Bernardon, Eduardo; Calazans, Ney; Garibotti, Rafael; MORAES, Fernando Gehm.
In: LASCAS, 2023.

Assessment of Lightweight Cryptography Algorithms on ARM Cortex-M Processors.
Moura, Nicolas; Lucena, Joaquim ; Pereira, Eduardo; Calazans, Ney; Ost, Luciano; Garibotti, Rafael; Moraes, Fernando Gehm.
In: SBCCI, 2023. – https://ieeexplore.ieee.org/document/10261962

6.1 Future Work

The following points highlight the key areas of interest for future research:

1. Assembly implementations optimization. Develop assembly implementations of AES-128 in CCM mode and ChaCha20-Poly1305 optimized for code size. This optimization would allow for a fairer comparison with Ascon. While AES-128 in CCM mode demonstrated better performance in its main operations, this improvement did not translate into the results compared to the pure software model. This difference may be attributed to differences in the optimization levels of the software implementations. Optimizing the assembly implementations would provide a more accurate comparison between the algorithms.

2. Development of Ascon hardware core. Develop an Ascon core that adheres to the available standards. This core should be thoroughly tested to ensure all necessary functionalities are implemented correctly. Additionally, it should be designed to connect to RISC-V via the AXI bus that has already been created. Establishing a reliable Ascon core would enhance the feasibility and accuracy of future evaluations and comparisons.

REFERENCES

[Akbas, 2019] Akbas, A. (2019). Comparative Analysis of Lightweight Cryptography Algorithms on Resource Constrained Microcontrollers. In *International Informatics and Software Engineering Conference (UBMYK)*, pages 1–4. https://doi.org/10.1109/UBMYK48245.2019.8965525.

[Alliance, 2023] Alliance, C. (2023). Rocket Chip Generator. https://github.com/chipsalliance/rocket-chip, November 2023.

[Arm, 2023] Arm (2023). AMBA AXI Protocol Specification. https://developer.arm.com/documentation/ihi0022/latest/, November 2023.

[Ascon, 2022a] Ascon (2022a). Ascon128v12 reference. https://github.com/ascon/ascon-c/tree/main/crypto_aead/ascon128v12/ref, November 2023.

[Ascon, 2022b] Ascon (2022b). NIST LWC Hardware Reference Design of Ascon v1.2. https://github.com/ascon/ascon-hardware, November 2023.

[Balasubramonian et al., 2017] Balasubramonian, R., Kahng, A. B., Muralimanohar, N., Shafiee, A., and Srinivas, V. (2017). CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization*, 14(2):1–25. https://doi.org/10.1145/3085572.

[Beg et al., 2019] Beg, A., Al-Kharobi, T., and Al-Nasser, A. (2019). Performance Evaluation and Review of Lightweight Cryptography in an Internet-of-Things Environment. In *International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–6. https://doi.org/10.1109/CAIS.2019.8769509.

[Campos et al., 2020] Campos, F., Jellema, L., Lemmen, M., Müller, L., Sprenkels, A., and Viguier, B. (2020). Assembly or Optimized C for Lightweight Cryptography on RISC-V. In *International Conference on Cryptology and Network Security (CANS)*, pages 526–545. https://doi.org/10.1007/978-3-030-65411-5_26.

[Cardoso dos Santos et al., 2020] Cardoso dos Santos, L., Großschädl, J., and Biryukov, A. (2020). FELICS-AEAD: Benchmarking of Lightweight Authenticated Encryption Algorithms. In *Smart Card Research and Advanced Applications (CARDIS)*, pages 216–233. https://doi.org/10.1007/978-3-030-42068-0_13.

[Cheng et al., 2022] Cheng, H., Großschädl, J., Marshall, B., Page, D., and Pham, T. (2022). RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography. https://tches.iacr.org/index.php/TCHES/article/view/9951, November 2023.

[da Rocha et al., 2022] da Rocha, V., Moura, N., Gava, J., Bandeira, V., Ost, L., Reis, R., and Garibotti, R. (2022). Soft Error Reliability Assessment of Lightweight Cryptographic Algorithms for IoT Edge Devices. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 457–460. https://doi.org/10.1109/ISCAS48785.2022.9937998.

[Dally et al., 2020] Dally, W. J., Turakhia, Y., and Han, S. (2020). Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48—-57. https://doi.org/10.1145/3361682.

[Dobraunig et al., 2021] Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. (2021). Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *Journal of Cryptology*, 34(3):33:1–33:42. https://doi.org/10.1007/s00145-021-09398-9.

[Elaguech et al., 2019] Elaguech, A., Kchaou, A., El Hadj Youssef, W., Ben Othman, K., and Machhout, M. (2019). Performance Evaluation of Lightweight Block Ciphers in Softcore Processor. In *International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pages 101–105. https://doi.org/10.1109/STA.2019.8717266.

[Elsadek et al., 2022a] Elsadek, I., Aftabjahani, S., Gardner, D., MacLean, E., Wallrabenstein, J. R., and Tawfik, E. Y. (2022a). Energy Efficiency Enhancement of Parallelized Implementation of NIST Lightweight Cryptography Standardization Finalists. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 138–141. https://doi.org/10.1109/ISCAS48785.2022.9937755.

[Elsadek et al., 2022b] Elsadek, I., Aftabjahani, S., Gardner, D., MacLean, E., Wallrabenstein, J. R., and Tawfik, E. Y. (2022b). Hardware and Energy Efficiency Evaluation of NIST Lightweight Cryptography Standardization Finalists. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 133–137. https://doi.org/10.1109/ISCAS48785.2022.9937643.

[Gewehr and Moraes, 2023] Gewehr, C. and Moraes, F. (2023). Improving the Efficiency of Cryptography Algorithms on Resource-Constrained Embedded Systems via RISC-V Instruction Set Extensions. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/sbcci60457.2023.10261964.

[Guo et al., 2016] Guo, Z., Karimian, N., Tehranipoor, M. M., and Forte, D. (2016). Hardware Security Meets Biometrics for the Age of IoT. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1318–1321. https://doi.org/10.1109/ISCAS.2016.7527491.

[Ibex, 2023a] Ibex (2023a). Ibex: An embedded 32 bit RISC-V CPU core. https://ibex-core.readthedocs.io/en/latest/01_overview/index.html, November 2023.

[Ibex, 2023b] Ibex (2023b). IBEX Security Features. https://ibex-core.readthedocs.io/en/latest/03_reference/security.html, November 2023.

[Intel, 2017] Intel (2017). TinyCrypt Cryptographic Library. https://github.com/intel/tinycrypt, November 2023.

[Kane et al., 2020] Kane, L. E., Chen, J. J., Thomas, R., Liu, V., and Mckague, M. (2020). Security and Performance in IoT: A Balancing Act. *IEEE Access*, 8:121969–121986. https://doi.org/10.1109/ACCESS.2020.3007536.

[Kotel et al., 2016] Kotel, S., Sbiaa, F., Zeghid, M., Machhout, M., Baganne, A., and Tourki, R. (2016). Performance Evaluation and Design Considerations of Lightweight Block Cipher for Low-Cost Embedded Devices. In *IEEE/ACS International Conference of Computer Systems and Applications (AICCSA)*, pages 1–7. https://doi.org/10.1109/AICCSA.2016.7945695.

[Lara-Nino et al., 2016] Lara-Nino, C. A., Morales-Sandoval, M., and Diaz-Perez, A. (2016). An Evaluation of AES and PRESENT Ciphers for Lightweight Cryptography on Smartphones. In *IEEE International Conference on Electronics, Communications and Computers (CONIELECOMP)*, pages 87–93. https://doi.org/10.1109/CONIELECOMP.2016.7438557.

[Ledwaba et al., 2018] Ledwaba, L. P. I., Hancke, G. P., Venter, H. S., and Isaac, S. J. (2018). Performance Costs of Software Cryptography in Securing New-Generation Internet of Energy Endpoint Devices. *IEEE Access*, 6:9303–9323. https://doi.org/10.1109/ACCESS.2018.2793301.

[Marshall, 2021] Marshall, B. (2021). Scalar Cryptography v1.0.0 rc2. https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0-rc2-scalar, November 2023.

[Marshall et al., 2020] Marshall, B., Newell, G. R., Page, D., Saarinen, M.-J. O., and Wolf, C. (2020). The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):109–136. https://tches.iacr.org/index.php/TCHES/article/view/8729.

[Marshall et al., 2021] Marshall, B., Page, D., and Pham, T. H. (2021). A Lightweight ISE for ChaCha on RISC-V. Cryptology ePrint Archive, Paper 2021/1030. https://eprint.iacr.org/2021/1030, November 2023.

[Mohajerani et al., 2021] Mohajerani, K., Haeussler, R., Nagpal, R., Farahmand, F., Abdulgadir, A., Kaps, J.-P., and Gaj, K. (2021). Hardware Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 164–169. https://doi.org/10.23919/DATE51398.2021.9473930.

[Moura et al., 2023] Moura, N., Lucena, J., Pereira, E., Calazans, N., Ost, L., Moraes, F., and Garibotti, R. (2023). Assessment of Lightweight Cryptography Algorithms on ARM Cortex-M Processors. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. https://doi.org/10.1109/SBCCI60457.2023.10261962.

[Nir and Langley, 2015] Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. RFC 7539. https://www.rfc-editor.org/info/rfc7539, November 2023.

[NIST, 2001a] NIST (2001a). *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology. 38p., https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

[NIST, 2001b] NIST (2001b). *Recommendation for Block Cipher Modes of Operation*. National Institute of Standards and Technology. 66p., https://doi.org/10.6028/nist.sp.800-38a.

[NIST, 2007] NIST (2007). *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*. National Institute of Standards and Technology. 27p., http://dx.doi.org/10.6028/NIST.SP.800-38C.

[NIST, 2022] NIST (2022). Lightweight Cryptography: Project Overview. https://csrc.nist.gov/projects/lightweight-cryptography, November 2023.

[OpenHWGroup, 2023] OpenHWGroup (2023). CORE-V CV32E40P RISC-V. https://github.com/openhwgroup/cv32e40p, November 2023.

[Regla and Festijo, 2022] Regla, A. I. and Festijo, E. D. (2022). Performance Analysis of Light-weight Cryptographic Algorithms for Internet of Things (IoT) Applications: A Systematic Review. In *IEEE International Conference for Convergence in Technology (I2CT)*, pages 1–5. https://10.1109/I2CT54291.2022.9824108.

[Rescorla, 2018] Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://www.rfc-editor.org/info/rfc8446, November 2023.

[Rescorla et al., 2022] Rescorla, E., Tschofenig, H., and Modadugu, N. (2022). The Datagram Transport Layer Security (DTLS) Protocol Version 1.3, november 2023. RFC 9147. https://www.rfc-editor.org/info/rfc9147.

[RISC-V, 2019] RISC-V (2019). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. https://riscv.org/technical/specifications, May 2023.

[Saarinen, 2020] Saarinen, M.-J. O. (2020). A Lightweight ISA Extension for AES and SM4. https://arxiv.org/abs/2002.07041, November 2023.

[Sud et al., 2022] Sud, P., Neisarian, S., and Kavun, E. B. (2022). Evaluating Cryptographic Extensions On A RISC-V Simulation Environment. In *Euromicro Conference on Digital System Design (DSD)*, pages 548–555. https://10.1109/DSD57027.2022.00079.

[Tehrani et al., 2020] Tehrani, E., Graba, T., Merabet, A. S., and Danger, J.-L. (2020). RISC-V Extension for Lightweight Cryptography. In *Euromicro Conference on Digital System Design (DSD)*, pages 222–228. https://10.1109/DSD51259.2020.00045.

[Thakor et al., 2021] Thakor, V. A., Razzaque, M. A., and Khandaker, M. R. A. (2021). Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities. *IEEE Access*, 9:28177–28193. https://doi.org/10.1109/ACCESS.2021.3052867.

[Zurich and of Bologna, 2017] Zurich, E. and of Bologna, U. (2017). Ibex: An embedded 32-bit RISCV CPU core. https://ibex-core.readthedocs.io/en/latest/, November 2023.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br