PUCRS

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

CARLOS GABRIEL DE ARAUJO GEWEHR

# HARDWARE ACCELERATION FOR POST-QUANTUM CRYPTOGRAPHY IN RESOURCE CONSTRAINED EMBEDDED SYSTEMS WITH RISC-V ISES

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**

# HARDWARE ACCELERATION FOR POST-QUANTUM CRYPTOGRAPHY IN RESOURCE CONSTRAINED EMBEDDED SYSTEMS WITH RISC-V ISES

**CARLOS GABRIEL DE ARAUJO GEWEHR**

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

**Porto Alegre**
**2024**

# Ficha Catalográfica

G396h Gewehr, Carlos Gabriel de Araujo

Hardware Acceleration for Post-Quantum Cryptography in Resource Constrained Embedded Systems with RISC-V ISEs / Carlos Gabriel de Araujo Gewehr. – 2024.

96.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. Post-quantum cryptography. 2. Crystals-Kyber. 3. Embedded Systems. 4. Instruction Set Extensions. 5. RISC-V. I. Moraes, Fernando Gehm. II. 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

**CARLOS GABRIEL DE ARAUJO GEWEHR**

# HARDWARE ACCELERATION FOR POST-QUANTUM CRYPTOGRAPHY IN RESOURCE CONSTRAINED EMBEDDED SYSTEMS WITH RISC-V ISES

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 February 29, 2024.

## COMMITTEE MEMBERS:

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

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS)

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

# ACELERAÇÃO EM HARDWARE DE CRIPTOGRAFIA PÓS-QUANTICA EM SISTEMAS EMBARCADOS DE BAIXO DESEMPENHO COM INSTRUÇÕES ESPECIALIZADAS EM UM PROCESSADOR RISC-V


## RESUMO

O iminente alvorecer da computação quântica apresenta ameaças a algoritmos criptográficos usados na atualidade para troca de chaves, como Diffie-Hellman, RSA e construções baseadas em Curvas Elípticas (ECC), por meio do algoritmo de Shor. Como resposta a estes desafios, o National Institute for Standard and Technology (NIST), órgão responsável pela padronização de algoritmos criptográficos nos Estados Unidos, iniciou uma competição para algoritmos de troca de chaves que apresentam resistência a ataques clássicos e quânticos. Em julho de 2022, o algoritmo Crystals-Kyber foi anunciado como vencedor dessa competição, sendo padronizado sob o nome ML-KEM. Implementações de tais algoritmos visando sistemas embarcados com poucos recursos computacionais são consideradas um problema em aberto, visto que há poucos trabalhos na literatura que exploram soluções os mesmos. Esse trabalho visa explorar aceleração em hardware por meio de instruções especializadas (ISEs) em um processador RISC-V de baixa complexidade em uma avaliação considerando desempenho, consumo de energia e memória, assim como custos em área, almejando obter uma implementação eficiente de um sistema criptográfico resistente a eventuais ataques ocasionados pela computação quântica, aderindo a padrões e algoritmos modernos. Além da exploração do algoritmo Kyber, também são avaliados algoritmos para funções resumo (*hash functions*) e criptografia simétrica autenticada (AEAD), sendo medidos os ganhos devido ao uso de ISEs para essas finalidades. Sucintamente, os ganhos medidos para funções resumo são de 32%, 38% e 16% em desempenho, gastos de energia e de memória, respectivamente. Para AEAD, os ganhos são de 58%, 61% e 35% em desempenho, gastos de energia e de memória, respectivamente. O custo em área é de 10% da área do processador Ibex base ou 4K portas lógicas equivalentes. A aceleração em hardware de primitivas simétricas (e.g. SHA-3) dentro do algoritmo Kyber implicam em ganhos de desempenho e energia de 32% cada. Combinando aceleração das primitivas simétricas e uma nova ISE proposta nesse trabalho chamada *XKyber*, ganhos adicionais de 46% and 44% em desempenho e gasto de energia são observados, além de uma redução no tamanho de código de 15%. O custos em área devido a ISE *XKyber* é novamente de 10% da área do processador base.



**Palavras-Chave:** Criptografia Pós-Quantica, Crystals-Kyber, Sistemas Embarcados, Baixa Potência, RISC-V, Ibex.

# HARDWARE ACCELERATION FOR POST-QUANTUM CRYPTOGRAPHY IN RESOURCE CONSTRAINED EMBEDDED SYSTEMS WITH RISC-V ISES


## ABSTRACT

The imminent rise of practical quantum computing threatens well-established cryptography algorithms for secret key exchange in use today, such as Diffie-Hellman, RSA and Elliptic Curve based schemes (ECC), via Shor's algorithm. To answer this challenge, the National Institute for Standard and Technology (NIST) has launched a competition for Key Encapsulation Mechanism (KEM) algorithms showing resistance to classical and quantum-based attacks. In July 2022, NIST announced that the Crystals-Kyber algorithm was chosen as the competition's winner, being standardized as ML-KEM. No works in literature sufficiently address the issue of efficient implementation of Kyber in resource-constrained embedded systems. This work aims to explore hardware acceleration through Instruction Set Extensions (ISEs) in a low-end 32-bit RISC-V core in a comprehensive evaluation comprising performance, energy consumption, memory footprint and die area costs, enabling an efficient implementation of a cryptosystem that can withstand attacks from the emergence of quantum computers and is compliant to current cryptographic standards and algorithm suites. In addition to Kyber, this work also explores several algorithms for authenticated encryption (AEAD) and hash functions at the 128 and 256 bit security levels, evaluating improvements due to the use of specialized instructions in each algorithm. In summary, the use of ISEs in hash functions provides gains of 32%, 38% and 16% in performance, energy consumption, and code size, respectively. Gains in authenticated encryption are of 58%, 61% and 35% in performance, energy consumption, and code size, respectively. Area costs are of at most 10% of the baseline Ibex processor with no ISEs, corresponding to 4K equivalent gates. Hardware acceleration of symmetric primitives (e.g. SHA-3) in Kyber show performance and energy gains of 32% each. Combining hardware acceleration via a novel *XKyber* ISE and of Kyber symmetric primitives, further gains of 46% and 44% in performance and energy consumption are observed, while also reducing code size by 15%. *XKyber* area costs are again of 10% of the baseline Ibex processor with no ISEs.



**Keywords:** Post-quantum cryptography, Crystals-Kyber, Embedded Systems, Low Power, RISC-V, Ibex.

# LIST OF FIGURES

# LIST OF TABLES

# CONTENTS

# 1. INTRODUCTION

The advancement of quantum computing has raised concerns about the long-term security of current key exchange mechanisms, such as Diffie-Hellman and Elliptic Curve Diffie-Hellman (ECDH), widely used in current cryptographic systems. These traditional methods are vulnerable to quantum-based attacks, which could compromise the confidentiality and integrity of sensitive information in the future. Consequently, deploying new key exchange algorithms based on mathematical foundations that are naturally resistant to such attacks is crucial, giving birth to a new field of research called Post-Quantum Cryptography (PQC). Lattice-based problems such as Learning with Errors (LWE) and its variants, particularly Module Learning with Errors (MLWE), emerged as potential solutions, showing resistance to classical and quantum-based attacks. Several algorithms have been proposed using the MLWE problem as the basis for its security guarantees.

To address this need, the National Institute of Standards and Technology (NIST) initiated a competition to standardize new quantum-resistant key exchange algorithms. After several submissions for Key Encapsulation Mechanisms (KEM), the winning algorithm was announced in July 2022 to be the MLWE-based scheme Crystals-Kyber [Avanzi et al., 2022]. As a result of this competition, Kyber is expected to be soon standardized under the name ML-KEM [NIST, 2023b].

As with most lattice-based schemes, Kyber introduces performance and memory usage overheads. These overheads become even more pertinent in embedded systems and Internet of Things (IoT) devices, which feature low-complexity processors and limited on-chip memory. It should be noted that such devices typically serve as the first point of data collection from sensor interfaces, which are then forwarded to upstream complex nodes. Upstream nodes possess greater computational power and are better equipped to secure data effectively. However, the same cannot be said for downstream, less complex nodes. The efficient provision of post-quantum secure communication in low-complexity network endpoints is paramount for the further advancement of the IoT.

Since the NIST PQC competition, recommended algorithm sets encompassing all cryptographic services usually needed for practical security are being updated to include quantum-safe algorithms. Of such algorithm sets, the most relevant is the Commercial National Security Algorithm Suite (CNSA) 2.0, maintained by the National Security Agency (NSA), listing publicly-known algorithms recommended for usage in US government systems [NSA, 2023]. Even though the 128-bit security level is widely regarded as enough for practical security, CNSA 2.0 demands the use of algorithms at the 256-bit security level for symmetric cryptography and hash functions. The challenges in providing post-quantum security are further aggravated by the need to support 256-bit level primitives if CNSA 2.0 compliance is to be pursued.

Parallel to the post-quantum competition, NIST has also promoted the lightweight cryptography (LWC) competition, aiming to standardize algorithms tailored to devices with constrained resources. The LWC competition accepted the submission of algorithms providing symmetric encryption and optionally hashing capabilities. Such algorithms should provide better performance and memory footprint than traditionally used algorithms at the 128-bit security level. In February 2023, the Ascon [Dobraunig et al., 2021] algorithm family was selected as the winner of the LWC competition. The intersection between Lightweight Cryptography and Post-Quantum Cryptography is still an open research problem.

Hardware acceleration can significantly enhance performance, memory, and energy efficiency in crucial operations of an algorithm. One way to achieve hardware acceleration is by implementing specialized instructions in a general-purpose processor, a practice known as Instruction Set Extensions (ISEs). Compared to memory-mapped accelerators, extending a base instruction set with specialized instructions for a given algorithm provides benefits relevant to low-power embedded systems, including: (*i*) resource sharing among generic and specialized components, such as the register file and RAM interfaces; (*ii*) trivial data transfer among generic and specialized components via the register file, avoiding costly memory accesses; (*iii*) no added system complexity to e.g. the bus and interrupt controllers.

## 1.1 Motivation

The primary motivation for this study is the scarcity of research in the literature concerning Kyber implementations and hardware acceleration via ISEs in low-resource embedded systems. As discussed in detail in Chapter 5, most previous efforts in Kyber implementations focus on software-only implementations that do not explore hardware acceleration or highly complex, performance-driven implementations that do not prioritize crucial metrics for IoT, such as energy consumption and memory footprint. Given the increasing deployment of IoT devices, addressing this issue is becoming more critical [García-Morchón et al., 2019]. Despite the clear need for post-quantum security solutions tailored to the IoT context, there is a significant gap in the literature regarding Kyber implementations in such systems.

Despite practical quantum computing not yet being feasible[1], it is still necessary to consider security against quantum adversaries in the present. Two hypothetical situations illustrate this need: quantum-based attacks could become prevalent in devices with a long expected lifespan while the device is still in operation. Additionally, in deployments with high maintenance costs, the financial burden of replacing a classically secure solution with a quantum-secure solution may be prohibitive. This highlights the practical requirement for implementing quantum-secure solutions before actual quantum-based attacks become possible.

[1] https://www.ibm.com/quantum/roadmap

## 1.2 Objectives

The strategic objective of this work is to offer an area- and energy-efficient approach for implementing post-quantum cryptography, considering both the 128-bit and 256-bit security levels in low-resource embedded systems such as IoT sensor nodes.

Specific goals include:

1. Provide an RTL implementation and simulation environment for a RISC-V processor [RISC-V Foundation, 2019], namely the Ibex processor [lowRISC, 2018], with specialized instructions for cryptography, encompassing the standardized *Zkne*, *Zknh* and *Zbkb* extensions [RISC-V Foundation, 2022], for AES encryption, SHA-2 hashing and general bit manipulation, respectively;
2. Make use of the specialized instructions previously implemented in an application-ready C library, integrating into the TinyCrypt library [Intel, 2017] hardware-accelerated implementations of the AES-128, AES-256, SHA-256, and SHA-512 algorithms;
3. Implement AES with the T-Table technique for higher performance at a memory footprint cost, keeping compatibility with the TinyCrypt AES API;
4. Integrate into the environment a RISC-V optimized SHA-3 implementation from [Saarinen, 2022], using the *Zbkb* extension;
5. Implement and integrate in software the non-standardized *XAscon* extension proposed in [Cheng et al., 2022], accelerating the Ascon algorithm.
6. Integrate the hardware-accelerated symmetric primitives from the previous steps in the reference Crystals-Kyber implementation from the NIST competition third round, considering (based on the symmetric primitives used) the standard Keccak-based version; Kyber-90s, using AES and SHA-2, and a novel Ascon-based version.
7. Propose a novel *XKyber* extension for providing hardware acceleration specific to the Crystals-Kyber algorithm, implement it in the Ibex processor, and integrate it in the Kyber C implementation.
8. Evaluate quantitatively the effectiveness of the resulting pure-software and hardware accelerated with relevant extensions implementations in several metrics of interest to resource-constrained embedded systems, such as performance, code size, and memory footprint for the software components, as well as die area usage and energy consumption for the hardware components.

## 1.3 Methodology

The work described in this dissertation was conducted as follows. Initially, the Ibex processor, an existing open-source implementation of the RISC-V ISA, was extended with specialized instructions that conform to the scalar cryptography extension of the RISC-V architecture [RISC-V Foundation, 2022]. Those instructions accelerate the SHA-2, SHA-3, and AES (encryption only) algorithms and are defined in the *Zknh* (SHA-2), *Zbkb* (SHA-3) and *Zkne* (AES encryption) extensions. In addition to using standardized extensions, a non-standardized extension [Cheng et al., 2022] for the Ascon algorithm was also implemented and evaluated. The SHA-2, SHA-3, AES and Ascon primitives will first be evaluated by themselves (i.e., not integrated into a higher-level component), in order to explore and compare their use for hashing and encryption. The evaluation measured performance and memory footprint, while energy consumption and die area usage of the accelerated solution were measured and compared against the non-extended version of Ibex via post-synthesis simulations. The methodology for obtaining concrete results is described in greater detail in Section 2.3.

These primitives were integrated into different Kyber versions, taking advantage of hardware acceleration from the newly added specialized instructions. Kyber versions differ in their choice of symmetric primitives. The standard Keccak-based Kyber uses primitives from the SHA-3 standard, and is accelerated using the *Zbkb* extension; Kyber-90s uses primitives from the SHA-2 and AES standards, and is accelerated using the *Zknh* and *Zkne* extensions; Kyber-Ascon uses Ascon, accelerated using the *XAscon* extension. Keccak-based Kyber and Kyber-90s are proposed in the original NIST competition, while Kyber-Ascon is a novel version from this work. All 3 versions are evaluated both with and without hardware acceleration via ISEs, such that the evaluation can bring understanding concerning both software-only and hardware-accelerated implementations.

In addition to hardware acceleration of the symmetric primitives, a novel extension called *XKyber* was also proposed. This extension provides specialized instructions specific to the internal operations in the Kyber algorithm, such as polynomial coefficients multiplication and compression, enabling further hardware acceleration beyond the symmetric primitives. Each internal operation accelerated by *XKyber* was analyzed, comparing their software-only and hardware-accelerated implementations under the same criteria and methodology previously described.

## 1.4 Document organization

This manuscript is organized as follows.

- Chapter 2 presents fundamental concepts concerning hardware acceleration and the RISC-V architecture, as well as the Ibex processor used in this work and the experimental setup around it;
- Chapter 3 presents work performed concerning the hardware acceleration of hash functions in the SHA-3, SHA-2 and Ascon standards;
- Chapter 4 presents work performed concerning the hardware acceleration of symmetric AEAD (Authenticated Encryption with Associated Data) cryptography with the AES and Ascon algorithms;
- Chapter 5 presents work concerning asymmetric cryptography with the Crystals-Kyber algorithm.
- Chapter 6 finishes this manuscript with conclusions derived from previous chapters and highlights future work.

Chapters 3, 4, and 5 constitute the core of this Dissertation. Each chapter has been structured to be self-contained, encompassing a review of the literature, a detailed account of the research conducted, the development process, results, and conclusions specific to the chapter's subject. To adhere to the criterion of self-sufficiency for each chapter, instances of content repetition are inevitable. These possible overlaps are explicitly mentioned in the text.

# 2. RISC-V AND HARDWARE ACCELERATION

This chapter provides an introduction to fundamental concepts critical to the understanding of this work. Section 2.1 introduces the RISC-V instruction set, an open instruction set architecture, and the Ibex processor. Section 2.2 provides an overview of implementing hardware acceleration of post-quantum cryptography in contexts with limited computational resources, such as processing power and memory, for example, in low-cost IoT devices. Finally, Section 2.3 illustrates the experimental setup used in the evaluations.

## 2.1 RISC-V Architecture and Ibex

The RISC-V architecture [RISC-V Foundation, 2019] emerged in the context of embedded systems as a viable alternative to the long-standing ARM processors, with multiple companies such as SiFive[1], Codasip[2] and Andes Technology[3] providing commercially successful IP to a broad range of applications. As an open instruction set, no company owns the RISC-V standard, and as such, no license fees are required to implement and commercialize RISC-V IPs. Considering design choices made in previous RISC architectures such as MIPS and SPARC, RISC-V defines a minimal set of instructions to be implemented, such that the development of software-side tools such as compilers can be performed without assumptions concerning the underlying micro-architecture that implements the instructions defined in the RISC-V specification.

Note that the RISC-V standard makes an explicit effort as to not make any definitions that may favor a certain implementation flavor: RISC-V processors can range from simple low-complexity microcontroller-oriented implementations to highly complex multi-core superscalar performance-optimized processors. Despite great differences in possible micro-architectures, they all must implement the same underlying base instruction set. In fact, the range of possible applications of RISC-V is treated as a core point in the development of the RISC-V specification in itself. It is important that RISC-V is not seen only as an "ARM replacement", but capable of enabling a wide range of solutions in different application domains [Waterman, 2016].

Supporting the design philosophy for concern separation between instruction set definition and micro-architecture implementation, the RISC-V specification defines several optional extensions for functionality that exceeds the fundamental already defined instruction set. This allows for the parallel development of compilers and hardware, both starting from a well-defined common ground. Additionally, this discourages ad-hoc vendor-specific

[1] https://www.sifive.com/risc-v-core-ip
[2] https://codasip.com/products/codasip-risc-v-processors/
[3] http://www.andestech.com/en/products-solutions/andescore-processors/

implementations, such that compatibility between different software and hardware vendors is maintained, while micro-architecture design space is not constrained to restrict implementations to specific characteristics. An example of such a standardized extension is the M extension for native multiplication instructions. Software may be compiled for a target in which the M extension has been implemented, for which MUL and MULH instructions can be issued by the assembler, or for a target without the M extension, where multiplication is performed via compiler-generated software routines. Note that this is transparent to the software implementation.

Due to RISC-V being an open instruction set, several open-source implementations have recently been made popular. One implementation is the Ibex processor [lowRISC, 2018], presented in Figure 2.1. The version of Ibex used in this work implements the RV32IMC instruction set, meaning an integer (I) 32-bit base ISA with the M (multiplication) and C (compressed instructions) extensions.

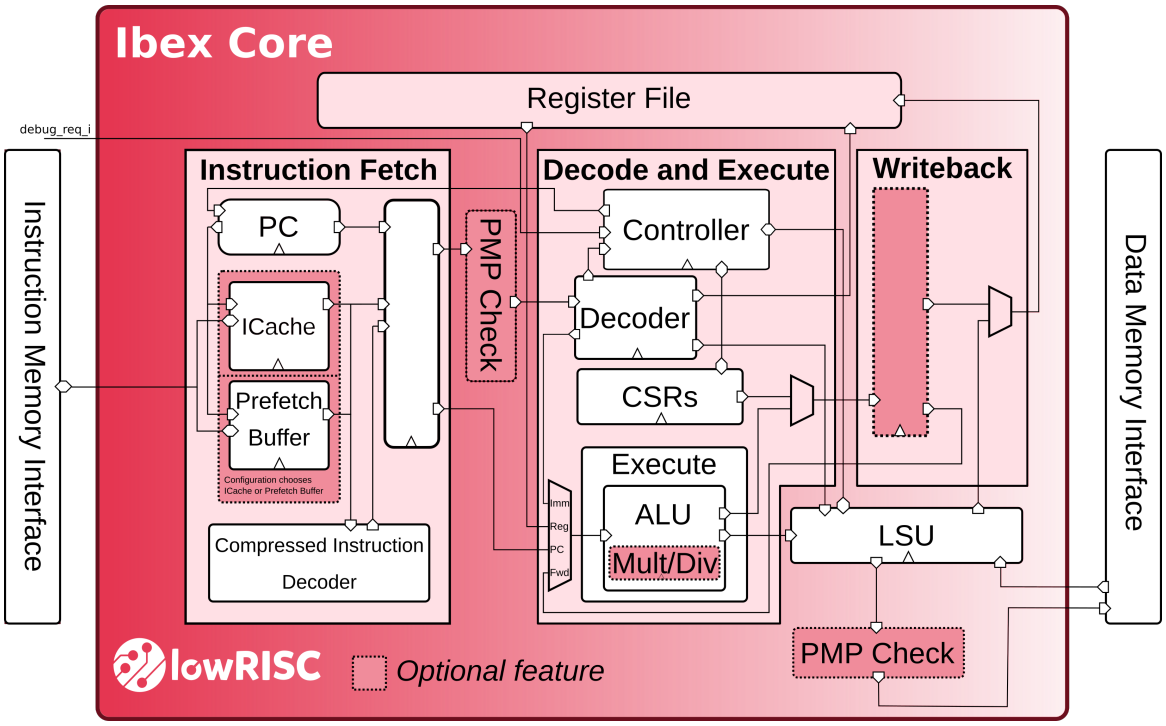


Figure 2.1 – Ibex processor [lowRISC, 2018].

The primary design objective of the Ibex processor is low complexity, reflected in its two-stage pipeline architecture. In the first stage, the processor fetches instructions from the main memory, while the second stage handles instruction decoding and execution, completing most instructions in a single cycle. This simplified pipeline design enables a more straightforward implementation, reducing power consumption and die area - highly desirable characteristics in IoT environments.

## 2.2 Hardware Acceleration Fundamentals

Algorithms may be subject to constraints on various metrics during execution. These requirements could specify that the algorithm must execute within a set time limit, use a predetermined amount of memory or stack space, or maintain energy consumption under a certain threshold. Typically, algorithms are abstractly defined, and concrete implementations are developed case-by-case to ensure context-specific constraints are met. For instance, in scenarios where performance is a higher priority than code size, certain values that are difficult to compute at runtime may be precomputed and stored in memory in a look-up table. Conversely, a code-density-focused implementation may compute these values in real time, despite reduced performance compared to the first hypothetical implementation.

Note that an abstract algorithm is made concrete via its implementation through a given programming language, by which any number of techniques can be used to fine-tune the implementation towards a given design goal. Independently of the implementation itself, the context in which the execution of the algorithm will take place is static: no matter what data structures and optimized libraries are used in the implementation, the underlying execution context will always be the same. System characteristics such as memory access timings and processor instructions that the compiler can pick cannot be influenced by the programmer that writes the code, which implies a boundary to software-only optimizations.

Hardware acceleration of an algorithm enables the manipulation of the execution context characteristics for a given algorithm, providing greater flexibility to the implementing entity to explore the design space further. This approach can overcome the software-only optimization limitations and allow for a wider range of options available to a designer to alter system characteristics, which in turn can lead to improved metrics of interest.

Hardware accelerators can be classified into two dimensions: **coarse-grain** or **fine-grain** and **tightly-coupled** or **loosely-coupled** [Dally et al., 2020]. Coarse-grain accelerators execute large portions of the algorithm within themselves, while fine-grain accelerators execute only small parts (often the most critical) of the algorithm in question. Tightly-coupled accelerators are very closely integrated into the processor, which executes the non-hardware-accelerated parts of the algorithm in software, typically behaving as a separate functional unit of the main processor. In contrast, loosely-coupled accelerators are placed further away from the processor, typically behaving as a peripheral on a system bus.

An example of fine-grain loosely-coupled accelerators is presented in [Fritzmann et al., 2019]. The authors propose 2 accelerators for hash and Number Theoretic Transform (NTT) operations coupled to a main processor via an AHB bus. Such operations are commonly seen in PQC algorithms. The system architecture is shown in Figure 2.2. Note how the Hash and NTT accelerators (in purple) are in parallel in the bus hierarchy with the UART and GPIO peripherals. The accelerators take in their inputs directly from the main processor

via data memory bus writes, characterizing it as a loosely-coupled accelerator. Furthermore, due to the accelerators effectively accelerating only parts of a larger algorithm, this work presents fine-grain acceleration.

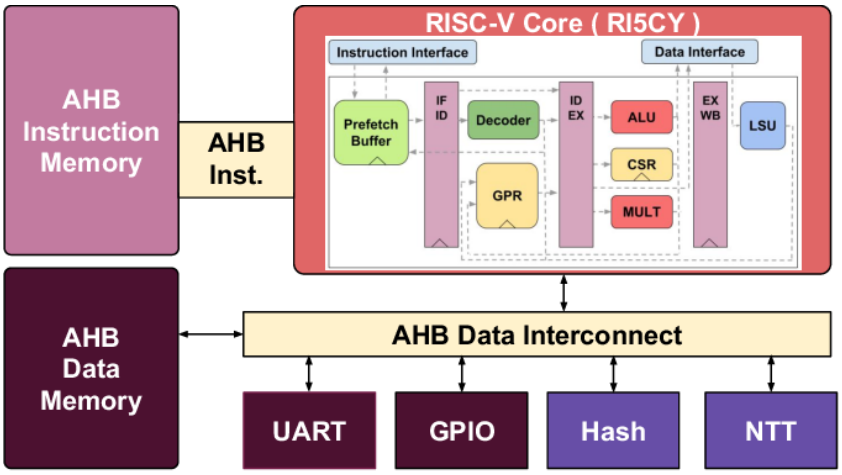


Figure 2.2 – Fine-grain loosely-coupled accelerators [Fritzmann et al., 2019].

A further development of [Fritzmann et al., 2019] is presented in [Fritzmann et al., 2020], where the authors integrate the aforementioned accelerators inside the general processor. The resulting architecture is presented in Figure 2.3. In this work, the accelerators take in their inputs from the main processors main register file (GPR block) or floating-point register file (FPR block), characterizing it as tightly-coupled accelerators. Each accelerator only accelerates specific portions of a larger algorithm. Accelerators cannot communicate between themselves directly, only through the main processor's register file, and as such, they cannot be seen as one big accelerator. This fact makes this work to be an example of fine-grain tightly-coupled accelerators.

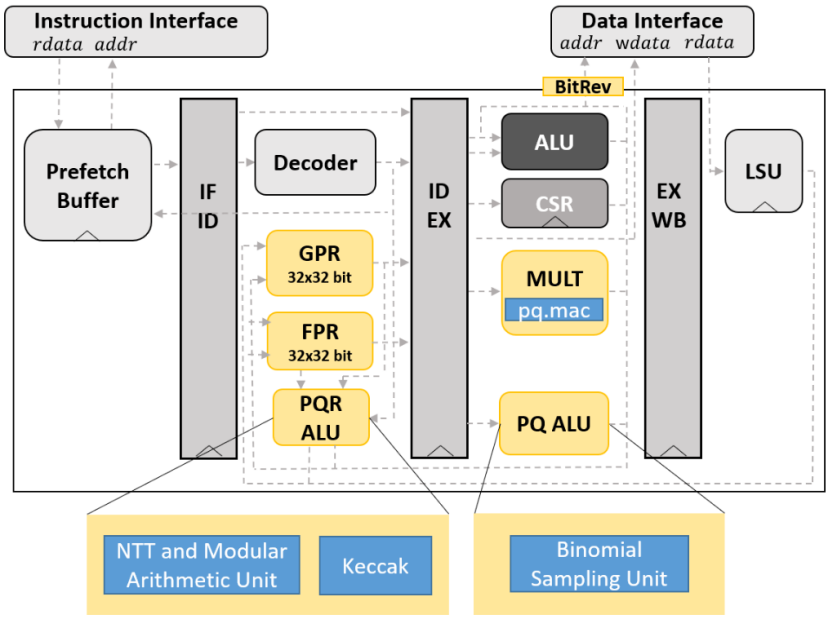


Figure 2.3 – Fine-grain tightly-coupled accelerators [Fritzmann et al., 2020].

[Banerjee et al., 2019] show a proposal for a generic accelerator supporting several lattice-based PQC algorithms. Their design can be visualized in Figure 2.4. Note how the design contains several accelerators grouped within the top hierarchical level. Accelerators can communicate directly between themselves and intermediary values can be stored in internal RAMs, independently of the main memory and processor. Due to these facts, the design can be seen as one comprehensive accelerator, instead of merely a set of different accelerators. A PQC algorithm can then have all its computationally-intensive tasks computed entirely within the accelerator, while software running in the main processor performs only trivial tasks. This proposal is characterized as a coarse-grain accelerator. Like [Fritzmann et al., 2019], data inputs come through memory bus writes issued from the main processor, making this a loosely-coupled accelerator.

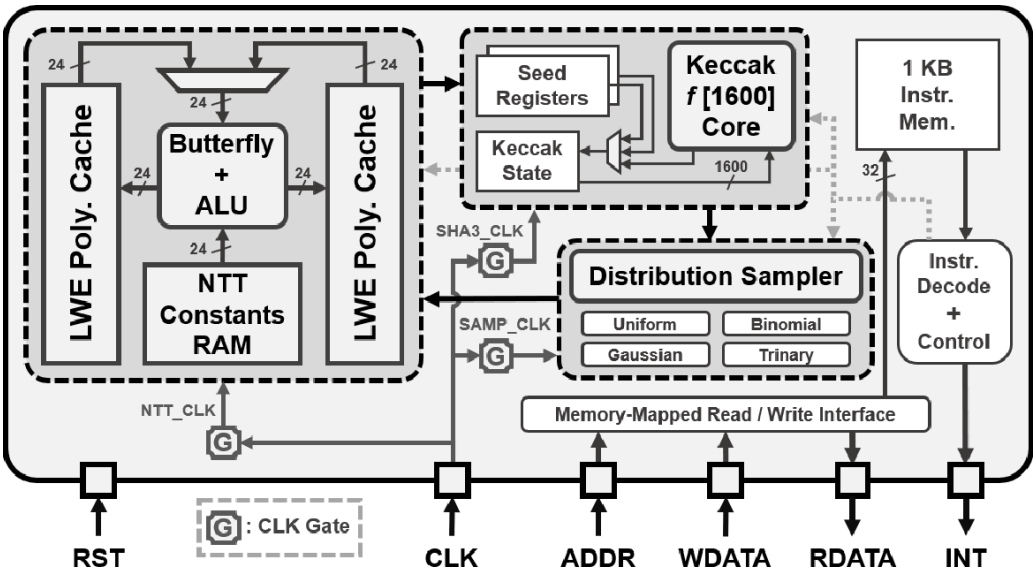


Figure 2.4 – Coarse-grain loosely-coupled accelerator [Banerjee et al., 2019].

Finally, a coarse-grain tightly-coupled accelerator can be seen in [Zgheib et al., 2021], where the Authors implement an AES accelerator inside an Ibex RISC-V processor, shown in Figure 2.5. This requires small modifications to the instruction decoding logic, to which a new instruction that enables and configures the AES accelerator is added, as well as the register file control, such that the register file can be interfaced either to the base Ibex processor or the AES accelerator. As the AES accelerator performs the entire algorithm in itself, the accelerator is regarded as a coarse-grain accelerator. Due to its close integration with the Ibex processor, it is a tightly-coupled accelerator.

It should be noted that an accelerator that is tightly coupled to a processor can result in a more efficient overall implementation by allowing for further exploration of resource sharing with existing elements of the base processor. The AES implementation described in [Zgheib et al., 2021] exemplifies this, as operations with data to and from the main memory are handled by the processor itself, resulting in significant resource sharing. However, there

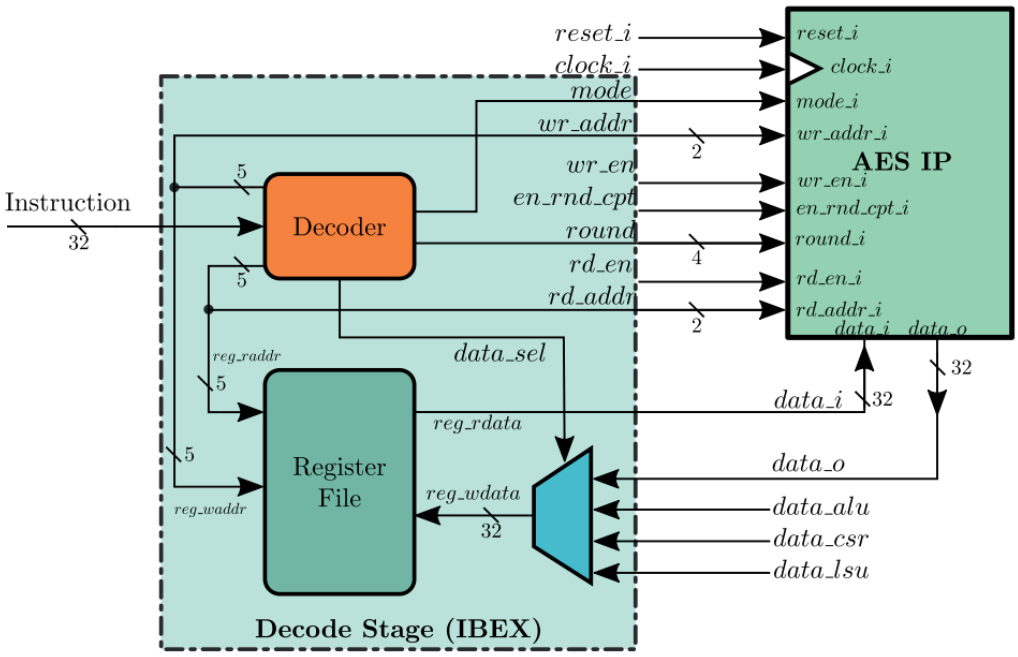


Figure 2.5 – Coarse-grain tightly-coupled AES accelerator [Zgheib et al., 2021].

is still a non-ideal aspect in moving data between the main processor register file and the internal registers in the AES accelerator.

To enable further exploration of resource sharing in tightly-coupled hardware acceleration, two approaches are found in the literature: **Instruction Set Extensions** (ISEs) and **co-processors**. Both approaches involve adding new specialized instructions to the base processor, which allows for the performance and efficiency gains of hardware acceleration while retaining the flexibility expected from a software implementation.

Co-processors can be understood as a collection of accelerators integrated between themselves, encapsulating related functionality in a self-contained module that is interfaced to the base processor via the instruction decoding logic and register file ports, similar to the AES accelerator example being explored. Typically, a co-processor will have its own register file and arithmetic elements. Note that resource sharing is explored not between the co-processor and the base processor but between functional units within the same co-processor, which have in common a register file and arithmetic elements. To move data to and from the co-processor, specialized instructions to move data words between register files must be implemented, which may present a non-trivial overhead to overall application performance (but still much less costly than moving data to and from main memory). Examples of co-processor implementations in a PQC context can be found in [Xin et al., 2020] and [Lee et al., 2022], explored in detail in Section 5.2.

ISA extensions also provide specialized instructions, but are implemented directly in the base processor, removing the overhead of moving data between register files and the need for a separate register file itself. Note that the register file accounts for the greatest share of the die area of all elements in a processor, making ISA extensions the method of choice for hardware acceleration in implementations where chip area is a concern. In

addition to exploring resource sharing with respect to the register file, further opportunities for resource sharing are found in the arithmetic elements of the processor and associated control logic, leading to even further area gains.

In conclusion, Instruction Set Extensions (ISEs) are of special interest to the IoT context, due to the minimal area overhead associated with providing hardware acceleration through this technique. Due to this characteristic, it was chosen as the approach presented in this work.

## 2.3 Experimental Setup and Methodology

All implementations in this work are validated with test vectors published by NIST or generated by the reference implementation of not yet standardized algorithms (Ascon and Kyber). Implementations are made to withstand remotely exploitable timing side-channel attacks by showing a constant-time execution. This is additionally enforced by running simulations with the Ibex data independent timing CSR being set to 1. Protection against higher effort attacks that require physical access to the device under attack such as power analysis or fault injection attacks are outside of the scope of this work. It should be noted that power analysis attacks could be mitigated by using the Ibex dummy instruction insertion feature[4].

The evaluation setup for all experiments performed in this work uses the Ibex Simple System[5], a minimal implementation of a system around the Ibex processor, including the essential elements necessary for running software on Ibex. Besides the Ibex processor itself, it contains a 2-port RAM, connected to the Ibex instruction and data memory interfaces; a programmable timer; and a control peripheral that can stop the simulation and write ASCII output from the processor into a file in the host machine for the simulation session, all via software running in the Ibex processor. Some modifications were performed to ease integration into the overall evaluation environment.

The previous RTL compilation and simulation flow employed the open-source FuseSoC and Verilator tools. These tools were replaced with a traditional flow using Make and Cadence XCelium. This flow aggregates the software and hardware build processes, such that executing arbitrary software in a given Ibex variant (with any ISE explored in this work) is made effortless, controlled by a set of Make targets and command-line supplied variables. The software build process employs GCC version 12.2.0 with the `-Os` and *-fstack-usage* flags, generating binaries optimized for size and outputting stack usage files for each compilation unit, used in the automated stack analysis flow discussed at the end of this section. For the non-standard *XAscon* and *XKyber* extensions, assembly using pre-built toolchains

[4] https://ibex-core.readthedocs.io/en/latest/03_reference/security.html#dummy-instruction-insertion
[5] https://github.com/lowRISC/ibex/tree/master/examples/simple_system

is impossible because the toolchain does not have knowledge of the instructions in these extensions. A custom *binutils* build supporting *XAscon* and *XKyber* opcodes is necessary.

The Ibex processor contains a set of performance counters[6] that can provide insights into the run-time behavior of software being executed. In addition to the standard Ibex counters, 6 new counters were added, such that loads and stores at word, half-word, and byte granularity can be differentiated. A profiler tool was implemented to accurately characterize the software being executed, which can capture the state of the performance counters during the simulation without software intervention. The profiler works at C function granularity, computing the difference between counter values at function entry and exit for a given list of functions of interest. In addition to performance counter outputs, the tool provides the simulation time for each entry and exit event.

Function entry and exit events are determined by monitoring the Program Counter (PC) register, which contains the address of the instruction being executed in the processor. A function entry event is seen when the PC register takes in the starting address of one of the C functions of interest. Each starting address is known via the ELF executable generated by the build process using the GNU *nm* tool. This process is entirely automated in the build environment; the user must only provide a list of function names, for which a file containing the starting address for each function is generated and read by the profiler at the beginning of the RTL simulation. Conversely, function exit is determined by checking the PC register for the address immediately after the `JAL` instruction that leads to a function entry event. Figure 2.6 shows sample output from the profiler tool.

```
[Profiler] Fetched end of symbol <pqcrystals_kyber1024_90s_ref_keypair> addr <102ef4> at <3209821.00 ns> <8082>
====================
Performance Counters
====================
Cycles: 1585938
NONE: 0
Instructions Retired: 1063497
LSU Busy: 206117
Fetch Wait: 80630
Loads: 133961
Stores: 72152
Jumps: 58789
Conditional Branches: 49713
Taken Conditional Branches: 49713
Compressed Instructions: 415681
Multiplier Busy: 126976
Divider Busy: 216
Memory store word: 21055
Memory store half word: 41986
Memory store byte: 9111
Memory load word: 58552
Memory load half word: 61434
Memory load byte: 13975
====================
```

Figure 2.6 – Profiler sample output.

Memory footprint is evaluated considering code size, static data, and maximum stack usage. Code size and static data are known through section sizes obtained via a custom script that parses the sections header of disassembled object code generated by the *objdump* tool, with *-fhSD* flags. Code is compiled with the *-fdata-sections* and *-ffunction-*

[6] https://ibex-core.readthedocs.io/en/latest/03_reference/performance_counters.html

*sections* GCC flags, creating individual sections for each function and data symbol instead of lumping them together in a single *.text* or *.data* section. Maximum stack usage considers the stack usage information output by GCC with the *-fstack-usage* flag for each function in a compilation unit. Stack usage information for each function is used to compute the worst-case path through the call graph, with stack usage as the cost metric. This is automated by an existing script[7], modified for use with RISC-V software.

Hardware-oriented metrics are evaluated using netlists generated by logic synthesis with Cadence Genus version 2112. A high-density 8-track cell library in 28 nm technology from STMicroelectronics is used. Synthesis considers a PVT (Process, Voltage, Temperature) corner of a slow process, 0.75 V at 125 C. Netlist simulations are performed using the aforementioned simulation infrastructure. Using the simulation time output function of the profiler, the average power of the Ibex core during a specific function call can be estimated using the collected switching activity of the synthesized netlist, considering the time window determined by the profiler tool. Multiplying the average power during this time window by the window size yields the energy consumed by the core during the function call. Power analysis with Cadence Joules considers a PVT corner of the nominal process, 0.9 V at 25 C.

To obtain a more comprehensive energy cost evaluation, memory accesses are considered in addition to the processor core costs. The CACTI 7 [Balasubramonian et al., 2017] tool is used to estimate the energy spent by memory reads and writes. The tool is configured to provide estimates for a 16 KByte memory with a read-only port and a second read/write port, considering low-power bitcells at the 28 nm technology node. The tool outputs energy cost values of 63.362 fJ/bit for reads and 41.436 fJ/bit for writes. The number of bits read from and written to the memory is determined via the performance counters output of the profiler tool. The relevant counters are the total amount of instructions, and the number of compressed instructions fetched during the function call, as well as amounts of loads and stores, differentiated by word, half-word, and byte granularity.

[7] https://github.com/ttsiodras/checkStackUsage/tree/master

# 3. HARDWARE ACCELERATION OF HASH FUNCTIONS USING ISES

This chapter presents the hardware acceleration using ISEs for hash functions using the SHA-2, SHA-3, and Ascon algorithm families. Section 3.1 introduces hash functions and the inner workings of the SHA-2, SHA-3, and Ascon-Hash/Ascon-XOF algorithms. Section 3.2 reviews previous work in the literature. Section 3.3 details the implementation of hardware acceleration using ISEs in the Ibex processor. Section 3.4 evaluates the implementations, following the methodology detailed in Section 2.3. Finally, Section 3.5 presents conclusions and final remarks.

Part of this chapter was published in the following conference:

> **Improving the Efficiency of Cryptography Algorithms on Resource-Constrained Embedded Systems via RISC-V Instruction Set Extensions**
> Carlos Gewehr, Fernando Gehm Moraes
> In: SBCCI, 2023

## 3.1 Introduction to Hash Functions

A hash function is a mapping from an input of arbitrary size to a fixed-size output. These functions provide an efficient means to verify a given piece of data. Comparing the data directly to an expected value can be challenging, particularly as the size of the data grows. Instead, the output of a hash function, which is of a fixed and smaller size than the original data, can be compared to a pre-computed hash value. This eliminates the need for many costly memory read operations in the data validation process, as the reference value for comparison is much smaller than the data itself.

Mapping arbitrary-sized inputs to a fixed-sized output presents an obvious issue:if there are many more possible inputs than possible outputs, then necessarily there must be an output value mapped to more than one input value. Having two known inputs associated with the same output value is known as a hash collision. This presents an issue in the data validation scenario presented above: if a collision occurs, invalid data can be mistakenly seen as valid if it produces the same hash as valid data. A malicious entity may exploit this by intentionally hashing invalid data that produces the same hash output as valid data. Therefore, the hash function must be carefully designed to make this as difficult as possible.

Note that hash collisions are inevitable, but are very unlikely to be found at random. For a hash function with output length $n$, the likelihood of finding an input value for a specific output at random is $2^{-n/2}$. Hash functions used in cryptography are designed such that a

collision for an input of choice cannot efficiently be obtained by a bad actor, which would break the data validation scheme being discussed. If the hash function does not have a predictable structure that can be used to map an output to an input, the only way to produce a collision is to perform a costly brute-force attack guessing random input values, which is impractical, and thus, the hash function can be regarded as fit for cryptographic applications.

Trusted hash functions that have been extensively researched are defined in SHA-2 [NIST, 2015a] and SHA-3 [NIST, 2015b] standards. Thus far, no attacks significantly more efficient than a brute-force search have been reported for the algorithms defined in these standards, and neither have any collisions for these algorithms been found at random.

### 3.1.1 SHA-2

The SHA-2 standard [NIST, 2015a] defines two hash functions of special interest: SHA-256 and SHA-512. Both functions are quite similar, at a user level mainly differ in their output sizes. SHA-256 has an output of 256 bits, while SHA-512 has a 512-bit-sized output.

The SHA-256 algorithm accepts an input of any size and divides it into message blocks of 512 bits each. If the input size is not a multiple of 512, padding is applied to the last message block. Each message block is compressed by a function that reduces 512 input bits to 256 bits. The compression output of the current message block is added to the compression output of the previous block at each compression operation. This running value is also referred to as the SHA-256 state and serves as input to the next message block compression function. After all message blocks have been compressed, the final SHA-256 state is taken as the hash function's output.

The SHA-256 algorithm can be seen in pseudocode in Algorithm 3.1. Note that all variables are 32-bit unsigned integers, and the addition operation is performed mod $2^{32}$, ignoring overflows. The sigma functions $\Sigma_0$, $\Sigma_1$, $\sigma_0$, $\sigma_1$, are defined in Figure 3.1, where *ROTRn* and *SHRn* denote rotation right shift and logical right shift by *n* bits, respectively.

$$
\begin{aligned}
\sum_0^{\{256\}}(x) &= ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\sum_1^{\{256\}}(x) &= ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\
\sigma_0^{\{256\}}(x) &= ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x) \\
\sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}
$$

Figure 3.1 – SHA-256 sigma functions [NIST, 2015a].

SHA-512 follows the same structure as SHA-256, with a few minor differences: internal variables are of 64 bits in size, message blocks are 1024 bits wide, and the inner-most loop runs in 80 iterations instead of 64. Consequently, the compression function compresses

**Algorithm 3.1** SHA-256 Hash Function

$Hash_0 \leftarrow 6A09E667_{16}$
$Hash_1 \leftarrow BB67AE85_{16}$
$Hash_2 \leftarrow 3C6EF372_{16}$
$Hash_3 \leftarrow A54FF53A_{16}$
$Hash_4 \leftarrow 510E527F_{16}$
$Hash_5 \leftarrow 9B05688C_{16}$
$Hash_6 \leftarrow 1F83D9AB_{16}$
$Hash_7 \leftarrow 5BE0CD19_{16}$
**for** each MessageBlock **do**
  **for** $i$ from 0 to 15 **do**
    $W_i \leftarrow MessageBlock_i$
  **end for**
  **for** $i$ from 16 to 63 **do**
    $W_i \leftarrow \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}$
  **end for**
  $\{A_{-1}, B_{-1}, C_{-1}, D_{-1}, E_{-1}, F_{-1}, G_{-1}, H_{-1}\} \leftarrow Hash_{\{0,7\}}$
  **for** $i$ from 0 to 63 **do**
    $T_{1i} \leftarrow H_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i$
    $T_{2i} \leftarrow \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1})$
    $A_i \leftarrow T_{1i} + T_{2i}$
    $B_i \leftarrow A_{i-1}$
    $C_i \leftarrow B_{i-1}$
    $D_i \leftarrow C_{i-1}$
    $E_i \leftarrow D_{i-1} + T_{1i}$
    $F_i \leftarrow E_{i-1}$
    $G_i \leftarrow F_{i-1}$
    $H_i \leftarrow G_{i-1}$
  **end for**
  $Hash_{\{0,7\}} \leftarrow \{A_{63}, B_{63}, C_{63}, D_{63}, E_{63}, F_{63}, G_{63}, H_{63}\} + Hash_{\{0,7\}}$
**end for**
**return** Hash

1024 bits to 512 bits, resulting in a 512-bit sized output. SHA-512 sigma functions also differ from SHA-256. The SHA-512 sigma functions are shown in Figure 3.2.

$$\begin{aligned}
\sum_0^{\{512\}}(x) &= ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \\
\sum_1^{\{512\}}(x) &= ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \\
\sigma_0^{\{512\}}(x) &= ROTR^{1}(x) \oplus ROTR^{8}(x) \oplus SHR^{7}(x) \\
\sigma_1^{\{512\}}(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^{6}(x)
\end{aligned}$$

Figure 3.2 – SHA-512 sigma functions [NIST, 2015a]

Due to being a NIST standard, seeing vast real-world deployments in addition to its integration into Kyber in its Kyber-90s version, the SHA-2 family of hash functions has been selected for evaluation in this work.

### 3.1.2 SHA-3

The SHA-3 standard defines several hash algorithms with varying output sizes, all based on the Keccak-p[1600] permutation and the so-called sponge construction. The width ($w$) of the Keccak permutation is determined by the sum of two parameters, rate ($r$) and capacity ($c$), such that $w = r + c$. The values of ($r$) and ($c$) are determined for each algorithm defined in SHA-3. The usage of these parameters for each algorithm in SHA-3 can be visualized in Table 3.1[1].

Table 3.1 – SHA-3 parameters (XOF: Extendable Output Function).

| Hash function / XOF | $r$ | $c$ | Output Length (bits) | Security Level (bits) |
|---|---|---|---|---|
| SHA3-224 | 1152 | 448 | 224 | 112 |
| SHA3-256 | 1088 | 512 | 256 | 128 |
| SHA3-384 | 832 | 768 | 384 | 192 |
| SHA3-512 | 576 | 1024 | 512 | 256 |
| SHAKE-128 | 1344 | 256 | - | 128 |
| SHAKE-256 | 1088 | 512 | - | 256 |

The Keccak-p[1600] permutation, (as defined in [NIST, 2015b], refered to as the Keccak permutation in the remainder of this manuscript) operates on a 1600 bits wide state in 24 iterations. The usage of the permutation macro $f$ in the sponge construction can be seen in Figure 3.3, where the input bit string is denoted by $N$, the output bit string denoted by $Z$ (truncated at $d$ bits), and the underlying permutation denoted by $f$. At each call to the permutation macro, $r$ bits from the input bit string are XOR-ed with the aligned output from the previous call to the permutation, until all bits from the $N$ input are fed into the sponge.

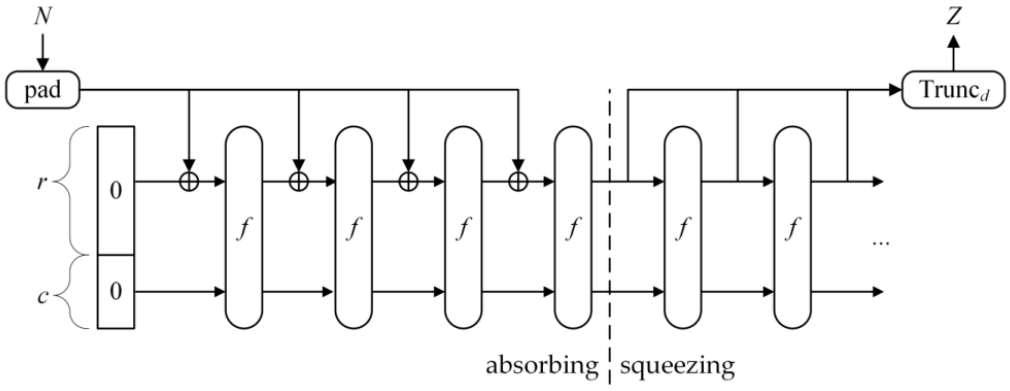


Figure 3.3 – SHA-3 sponge construction [NIST, 2015b].

[1] https://keccak.team/keccak_specs_summary.html

Once the input has been entirely processed, the next calls to the permutation function are used to obtain the output. Note that multiple calls to the permutation in the squeeze phase are possible. This is used to implement Extendable Output Functions (XOFs), which can be interpreted as hash functions with arbitrarily sized output (but finite security levels).

Internally, the Keccak permutation works on the 1600 bit state as a 5x5x25 cube, shown in Figure 3.4. In the Keccak state cube representation, bits grouped in the *x* dimension are rows, in the *y* dimension are columns and in the *z* dimension are lanes. Bits grouped in the *xy* dimensions are slices, in the *yz* dimensions as sheets and in the *xz* dimensions are planes. The entire permutation consists of 5 steps, performed in the following order: $\theta$ (Theta), $\rho$ (Rho), $\pi$ (Pi), $\chi$ (Chi) and $\iota$ (Iota).

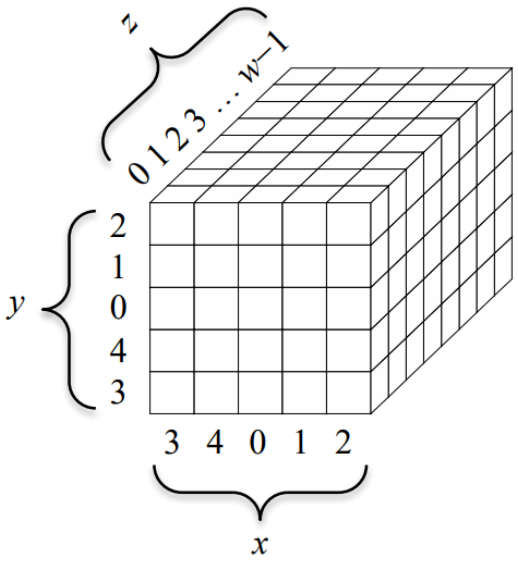


Figure 3.4 – Keccak permutation internal state [NIST, 2015b].

In the $\theta$ step, each column's parity is computed (represented by $\Sigma$ in Figure 3.5), producing 25 intermediary column parity values. These values are used to update each bit in the state, such that bit in the state after the $\theta$ operation is the result of the 3-way XOR between its previous value and 2 column parities. The two specific columns whose parities will influence the value of a bit in state after the $\theta$ operation are determined by its *x* and *z* coordinates, considering the columns given by $(x', z') = (x - 1 \ mod \ 5, z)$ and $(x'', z'') = (x + 1 \ mod \ 5, z - 1 \ mod \ 25)$. Note that all bits in the same column will be XOR-ed with the parities of the same two columns.

In the $\rho$ step each lane is rotated by a static amount, as shown in Figure 3.6. The $\rho$ rotation amounts (mod 64, the lane size) are shown in Figure 3.7.

The $\pi$ step shuffles bits in a slice. The manner in which bits are shuffled in shown in Figure 3.8.

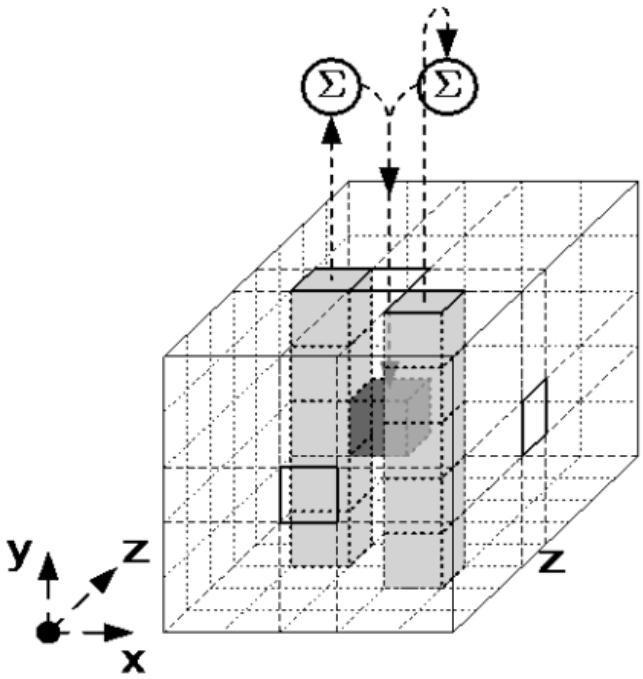


Figure 3.5 – Keccak $\theta$ (Theta) step [NIST, 2015b].

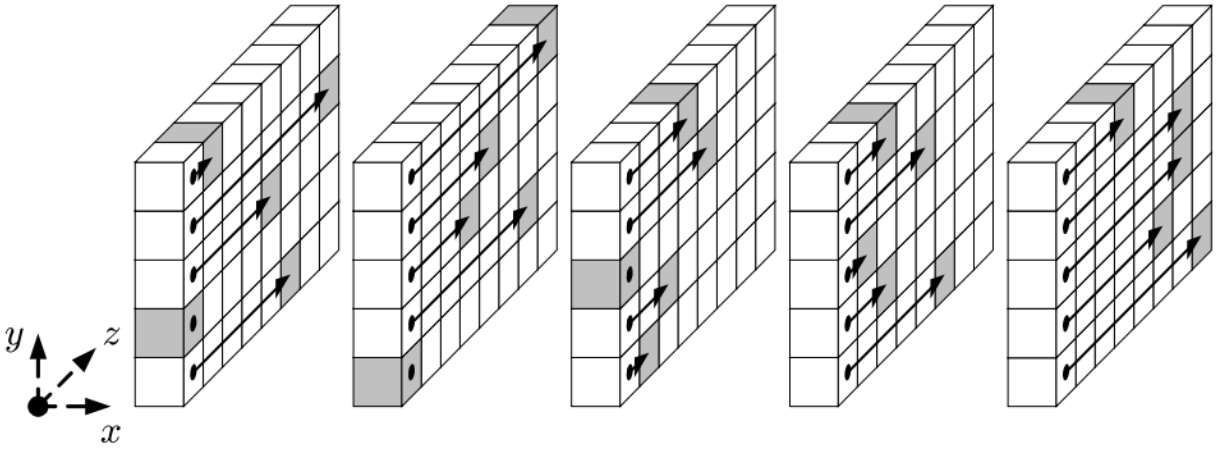


Figure 3.6 – Keccak $\rho$ (Rho) step [NIST, 2015b].

| | $x=3$ | $x=4$ | $x=0$ | $x=1$ | $x=2$ |
|---|---|---|---|---|---|
| $y=2$ | 153 | 231 | 3 | 10 | 171 |
| $y=1$ | 55 | 276 | 36 | 300 | 6 |
| $y=0$ | 28 | 91 | 0 | 1 | 190 |
| $y=4$ | 120 | 78 | 210 | 66 | 253 |
| $y=3$ | 21 | 136 | 105 | 45 | 15 |

Figure 3.7 – Keccak $\rho$ (Rho) step rotation amounts (mod 64) [NIST, 2015b].

The $\chi$ step combines 2 lanes in the same row via bitwise AND and XOR operations. The result is placed in a lane different than the 2 input lanes, such that lanes are shuffled at the same time new values for each lane are computed, as shown in Figure 3.9.

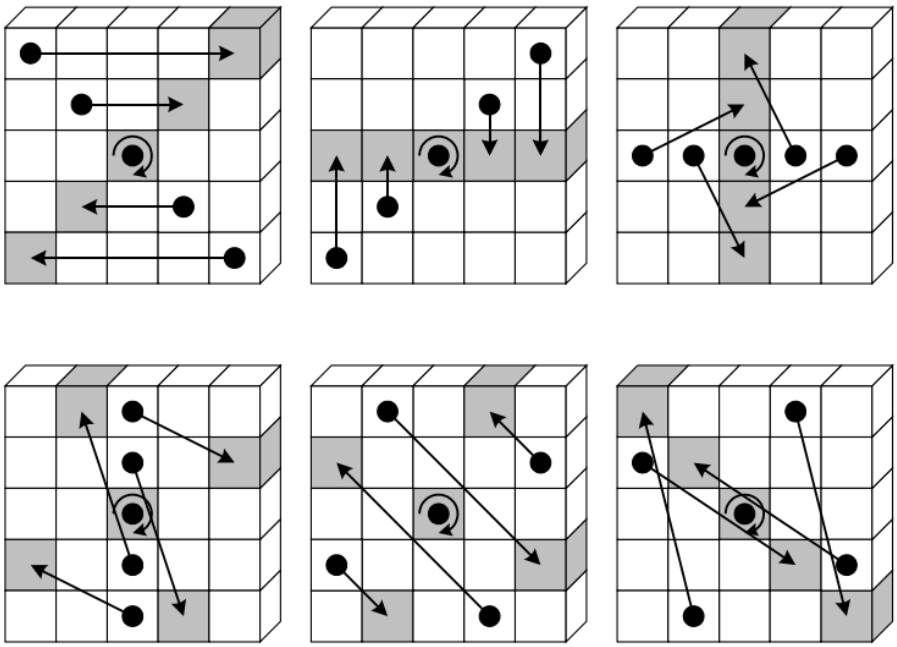

Figure 3.8 – Keccak $\pi$ (Pi) step [NIST, 2015b].

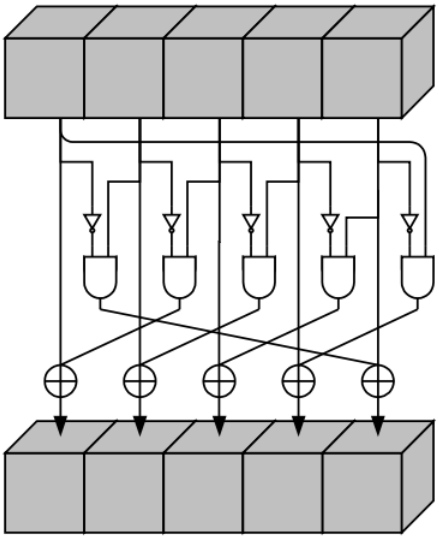

Figure 3.9 – Keccak $\chi$ (Chi) step [NIST, 2015b].

Finally, the $\iota$ (Iota) step updates the center lane with a round constant. Each round constant is determined via 7 steps of an 8 bit Linear Feedback Shift Register (LFSR), shown in Algorithm 3.2. The output bit in each step of the shift register is collected into an intermediary 7 bit word. This 7 bit word is expanded to a 64 bit word (lane size) by placing the $j$'th bit of the 7 bit word into the $2^j - 1$ bit of the 64 bit word. The remaining bits of the 64 bit words are filled with zeros. The 64 bit word is then XOR-ed into the center lane of the state.

Due to being a NIST standard, seeing increasing deployments in real-world applications such as its integration into Kyber in its standard Keccak-based version, the SHA-3 family of hash functions and XOFs have been selected for evaluation in this work.

**Algorithm 3.2** Keccak $\iota$ (Iota) shift register NIST [2015b]

**Require:** *R is the state of the LFSR*

$R \leftarrow 0||R$
$R[0] \leftarrow R[0] \oplus R[8]$
$R[4] \leftarrow R[4] \oplus R[8]$
$R[5] \leftarrow R[5] \oplus R[8]$
$R[6] \leftarrow R[6] \oplus R[8]$
**return** $R[1:8]$

### 3.1.3 Ascon-Hash and Ascon-XOF

Like SHA-3, Ascon-Hash and Ascon-XOF also work based on the sponge construction, using a different permutation. The permutation is the same the Ascon for AEAD, as presented in Section 4.1.2. The differences between Ascon for AEAD and Ascon-Hash/Ascon-XOF are in the workings of the sponge construction. While Ascon-Hash and Ascon-XOF work in the traditional manner like SHA-3, with data input occurring at the absorb phase and data output at the squeeze phase, AEAD Ascon has simultaneous input and output when absorbing plaintext. The AEAD Ascon sponge is covered in greater detail in Section 4.1.2.

Figure 3.10 presents the Ascon (hash) sponge. In normal sponge operation, at each stage of the Ascon sponge a permutation ($p^a$ or $p^b$) of the internal state is performed. For Ascon-Hash and Ascon-XOF $p^a$ and $p^b$ are the same 12 round permutation, with both algorithm differing only in the IV values and output size, which is fixed at 256 bits for Ascon-Hash or arbitrarily long for Ascon-XOF. This work considers the Ascon variant with the sponge rate ($M_i$ and $H_i$ widths) of 64 bits.

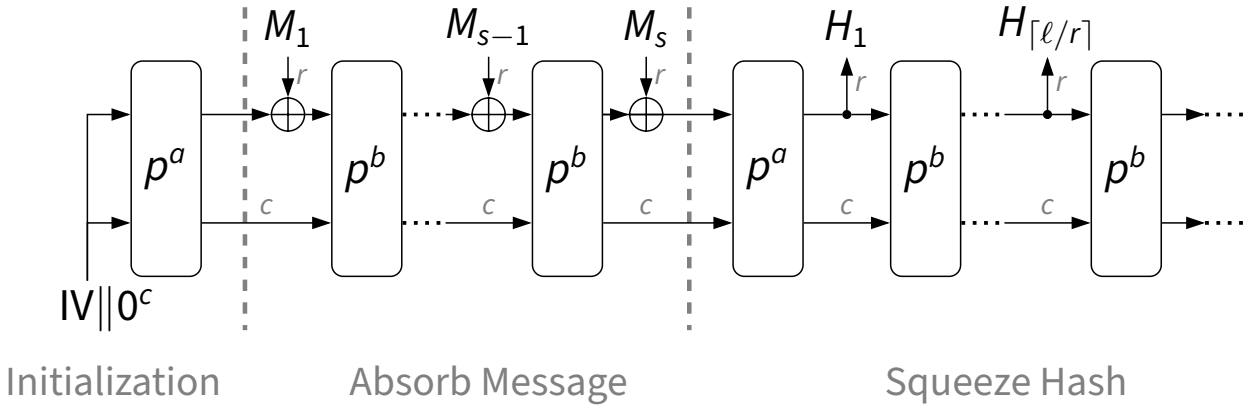


Figure 3.10 – Ascon-Hash and Ascon-XOF sponge [Dobraunig et al., 2021].

Internally, the Ascon permutation has a state of 320 bits, split into five 64-bit variables $x_0, ..., x_4$, shown in Figure 3.11. The permutation comprises 3 operations performed in the following order: the addition of a round constant into the $x_2$ state variable; the SBOX operation; and the linear diffusion operation.

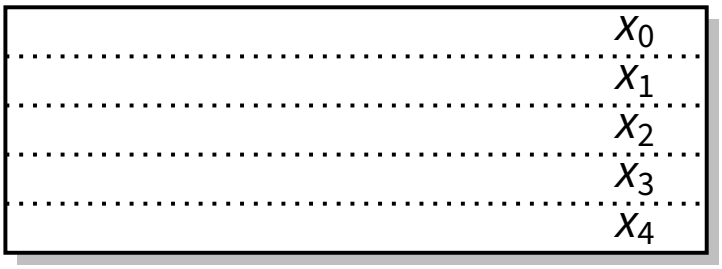


Figure 3.11 – Ascon internal state [Dobraunig et al., 2021].

The round constant addition operation is trivial, with the constant *Cr* being determined based on the current round number $r$. The round constant of the current round can be determined by the subtraction of the previous round constant by 15. The initial *C0* value varies between the $p^a$ and $p^b$ permutations. The *Cr* value for each round of each permutation is shown in Figure 3.12. The *Cr* constant is zero-extended to the left and XOR-ed into the $x_2$ state variable.

| $p^{12}$ | $p^8$ | $p^6$ | Constant $c_r$ | $p^{12}$ | $p^8$ | $p^6$ | Constant $c_r$ |
|---|---|---|---|---|---|---|---|
| 0 | | | 00000000000000f0 | 6 | 2 | 0 | 0000000000000096 |
| 1 | | | 00000000000000e1 | 7 | 3 | 1 | 0000000000000087 |
| 2 | | | 00000000000000d2 | 8 | 4 | 2 | 0000000000000078 |
| 3 | | | 00000000000000c3 | 9 | 5 | 3 | 0000000000000069 |
| 4 | 0 | | 00000000000000b4 | 10 | 6 | 4 | 000000000000005a |
| 5 | 1 | | 00000000000000a5 | 11 | 7 | 5 | 000000000000004b |

Figure 3.12 – Ascon round constants [Dobraunig et al., 2021].

The SBOX operation consists of a series of bitwise XOR and AND operations performed between each word in the state. The SBOX operation is depicted in Figure 3.13 and its interaction with the Ascon state in Figure 3.14. Note that each bit in a state variable in the SBOX operation depends only on itself and the same relative indexed bit in the remaining state variables. In this manner, the SBOX operation can be computed for any number of bits at a time, usually the machine word size.

Finally, the linear diffusion operates individually on each word of the state. Similarly to the SHA-2 sigma functions presented in Section 3.1.1, each word is determined based on the bitwise XOR of rotations of itself. Each Ascon sigma function is presented in Figure 3.15 and its interaction with the Ascon state in Figure 3.16.

Due to its novelty and upcoming standardization process as a result of the NIST LWC competition, showing good performance at a low memory footprint when compared with previous NIST standards, the Ascon family of algorithms has been chosen for evaluation in this work.

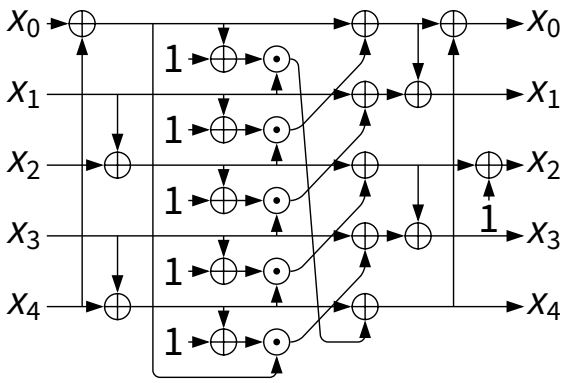


Figure 3.13 – Ascon 5-bit SBOX S(x) [Dobraunig et al., 2021].

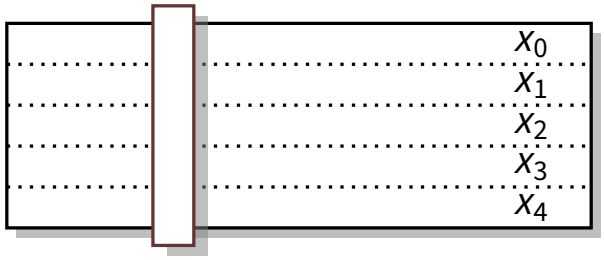


Figure 3.14 – Ascon SBOX applied to the state [Dobraunig et al., 2021].

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

Figure 3.15 – Ascon linear diffusion operation [Dobraunig et al., 2021].

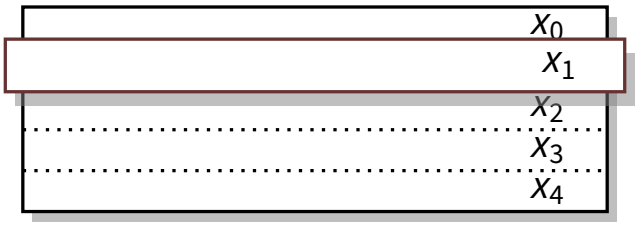


Figure 3.16 – Ascon linear diffusion applied to the state [Dobraunig et al., 2021].

## 3.2 Related Work

Software evaluations concerning the entire RISC-V cryptography ISE are reported in [Nişancı et al., 2022; Marshall et al., 2021]. In [Nişancı et al., 2022], SHA-256 and SHA-512 clock cycle count gains of 43% and 40%, as well as a reduction in program memory usage by 33% and 32% are reported, respectively. Despite reporting performance in terms of clock cycles instead of instructions retired, the authors of [Nişancı et al., 2022] do not

sufficiently describe their hardware platform, simply stating their processor core to be "a 32-bit 5-stage pipelined RISC-V processor that only supports base integer instructions". Crucial facts concerning the complexity of the RISC-V core used in their evaluation such as operand forwarding, branch prediction, and multiplier and shifter topology are omitted. This makes it difficult to compare results, seeing as the complexity of their RISC-V core cannot be correlated to the Ibex core we use. Similar results are reported in the earlier [Marshall et al., 2021] work, evaluated in the context of the SCARV [University of Bristol, 2018] core.

[Bertoni et al., 2012] presents several optimization techniques for the implementation of the Keccak permutation. [Stoffelen, 2019] evaluates the SHA-3 optimization techniques presented in [Bertoni et al., 2012] in an assembly implementation focused on performance rather than on code size. 13774 clock cycles are reported for the Keccak permutation, benchmarked in a SiFive E31 core.

[Campos et al., 2020] provide another similar evaluation of the Keccak permutation, porting the implementation for [Stoffelen, 2019] to C. An improvement of 3% due only to using C instead of assembly is reported, considering the same E31 core. Additionally, the authors evaluate the Keccak permutation considering the bit manipulation (*B*) extension performed on the Spike simulator. Results of 12402 and 14633 clock cycles are reported, with and without the bit manipulation extension, respectively.

[Saarinen, 2022] provides an implementation of the hash functions and XOFs on the SHA-3 standard. This implementation is more appropriate to the resource-constrained context being explored, as it does not make use of loop-unrolling such as in previous implementations in [Stoffelen, 2019; Campos et al., 2020], while still being well optimized in light of the techniques presented in [Bertoni et al., 2012]. It can also be easily configured as to use the *Zbkb* extension.

[Campos et al., 2020] also consider Ascon in their evaluation, considering an inlined ASM implementation of the Ascon permutation, and one permutation using the bit-interleaving technique, described in details in Section 3.1.2, both using loop unrolling.

[Cheng et al., 2022] propose ISEs for several LWC algorithms, among them Ascon. In the case of Ascon, they propose a new custom extension called *XAscon* and evaluate it considering software-only and *Zbkb*-accelerated implementations. The evaluation for the *XAscon* extensions makes use of both *XAscon* and *Zbkb*. The *XAscon* + *Zbkb* implementation shows 2.28x and 1.18x performance gains compared to software-only and *Zbkb*-only implementations, respectively. Authors perform their work in the context of the Rocket core [Chips Alliance, 2023], with 5 pipeline stages and floating point instructions, more complex than expected for an IoT context. Hardware-specific metrics are evaluated in FPGAs but not ASIC implementations. The *XAscon* + *Zbkb* and *Zbkb*-only extended Rocket RV32GC processors come at an area cost of 1.282x and 1.140×, respectively, measured in FPGA LUTs used (Xilinx Kintex-7 xc7k160tfbg676).

From the works reviewed in this section, it is clear that ISEs in the context of low-complexity embedded systems are an under-explored field of study. Most works concern themselves with high-complexity processors, where area costs due to cryptography ISEs are of little relevance to the overall area cost of the processor; or conduct their evaluations using FPGA or open-source EDA tools, not showing accurate values representative of factual ASIC implementations, using commercial EDA tools. Additionally, this work considers a comprehensive evaluation of memory overheads and runtime behaviour in terms of clock cycles, instructions executed and memory accesses for its software aspects.

## 3.3 Implementation

This section presents the efforts made towards the efficient software-only and hardware accelerated implementation of SHA-2, SHA-3 and Ascon-Hash in the Ibex processor.

### 3.3.1 SHA-2 Implementation

Hardware acceleration of the SHA-256 and SHA-512 hash functions is provided through 10 instructions defined in the *Zknh* extension, allowing for efficiently computing the 4 sigma functions in each algorithm. For SHA-256, 4 instructions are defined: `sha256sum0`, `sha256sum1`, `sha256sig0` and `sha256sig1`. These instructions directly implement the $\Sigma_0^{256}$, $\Sigma_1^{256}$, $\sigma_0^{256}$ and $\sigma_1^{256}$ functions defined in the SHA-2 standard [NIST, 2015a], restated below.

$$\Sigma_0^{256}(x) = ROTR_2(x) \oplus ROTR_{13}(x) \oplus ROTR_{22}(x)$$

$$\Sigma_1^{256}(x) = ROTR_6(x) \oplus ROTR_{11}(x) \oplus ROTR_{25}(x)$$

$$\sigma_0^{256}(x) = ROTR_7(x) \oplus ROTR_{18}(x) \oplus SHR_3(x)$$

$$\sigma_1^{256}(x) = ROTR_{17}(x) \oplus ROTR_{19}(x) \oplus SHR_{10}(x)$$

Similar functions are defined for SHA-512:

$$\Sigma_0^{512}(x) = ROTR_{28}(x) \oplus ROTR_{34}(x) \oplus ROTR_{39}(x)$$

$$\Sigma_1^{512}(x) = ROTR_{14}(x) \oplus ROTR_{18}(x) \oplus ROTR_{41}(x)$$

$$\sigma_0^{512}(x) = ROTR_1(x) \oplus ROTR_8(x) \oplus SHR_7(x)$$

$$\sigma_1^{512}(x) = ROTR_{19}(x) \oplus ROTR_{61}(x) \oplus SHR_6(x))$$

The availability of *Zknh* instructions allows for an implementation that is not only faster, but more compact and energy-efficient. Implementing these 4 functions in plain RV32I is very costly due to the usage of rotation shifts, for which no RV32I instruction can directly compute. Computing a rotation shift by $n$ bits in RV32I instructions requires a logical shift by $n$ bits in the forward direction; a logical shift by $32 - n$ bits in the reverse direction; and OR-ing both these intermediary shifts together. This requires the use of additional temporary registers, which may lead to the compiler storing intermediary values in the stack due to the lack of registers available to the register allocator.

Since SHA-512 variables are defined to be 64 bits in size, such functions cannot be directly implemented as a single instruction yielding a 32-bit result, as is the case for SHA-256. Note that the temporary registers issue is aggravated due to the increased variable size. Six instructions are defined to accelerate SHA-512 sigma functions: `sha512sig0h`, `sha512sig0l`, `sha512sig1h`, `sha512sig1l`, `sha512sum0r` and `sha512sum1r`. With *Zknh* instructions the functions $\sigma_0^{512}$ and $\sigma_1^{512}$ are computed with 2 different instructions, for the high and low parts of the result separately, while $\Sigma_0^{512}$ and $\Sigma_1^{512}$ are computed via 2 executions of the same instruction, but once with inverted operands, implying a rotation by 32 bits.

*Zknh* instructions are implemented in their functional unit, shown in Figure 3.17. The SHA-2 unit is entirely combinational, with all *Zknh* instructions executing in a single cycle. The sigma functions discussed in Section 3.1.1 lend themselves to an efficient hardware implementation, seeing as shifts and rotations by static amounts have zero cost. The SHA-2 unit can be seen simply as a sea of XOR gates, computing each sigma function, and a MUX, choosing the high/low part of a sigma function of interest.

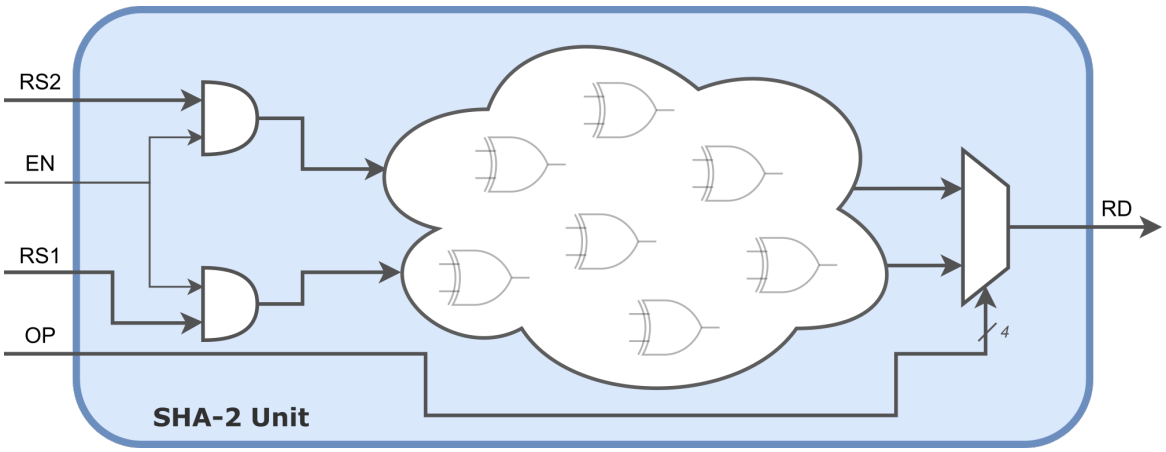


Figure 3.17 – SHA-2 Unit [Gewehr and Moraes, 2023].

The SHA-2 (and AES) functional unit integration in the Ibex processor's ID/EX pipeline stage can be seen in Figure 3.18. New elements added to the processor are colored in blue, existing elements modified to accommodate the new functional units are colored in green, while unmodified elements are colored in yellow.

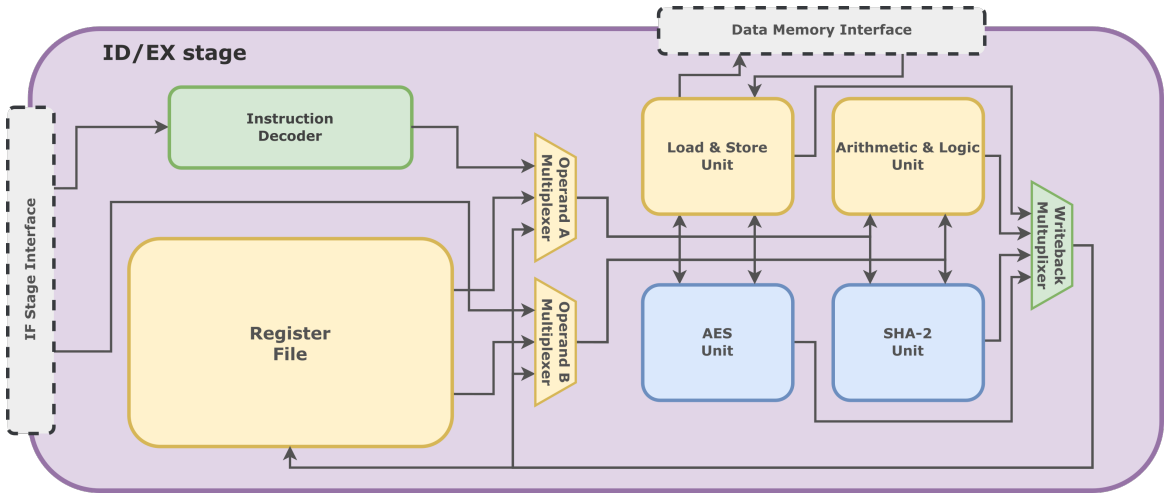


Figure 3.18 – Ibex ID/EX pipeline stage datapath with AES and SHA-2 functional units.

Figure 3.18 also illustrates the benefits of implementing hardware acceleration as ISEs over loosely-coupled accelerators in embedded contexts. Note that only small changes in the instruction decoding and writeback logic are required to integrate functional units into the Ibex processor, while resource sharing with existing processor elements is extensively explored, namely in the register file and load/store unit.

### 3.3.2 SHA-3 Implementation

SHA-3 relies on operations between different lanes. This presents challenges in implementing SHA-3 in targets with 32-bit words such as Ibex. Each lane in Keccak is defined to have 64 bits, implying in the need to have 2 32-bit registers storing a single lane. This makes computing the lane rotation operations from the $\rho$ (Rho) step difficult. The bit-interleaving technique can be used to aid the computation of 64-bit rotations. Instead of splitting a 64-bit variable between high and low registers (as defined in the RISC-V ABI), a register stores the even-indexed bits of the variable, while another stores its odd-indexed bits. A rotation by $2a$ bits in a 64-bit variable becomes two $a$ bit rotations, one in each 32-bit register. Note that this is transparent to bitwise operations between two lanes, as the relative position between bits in two lanes remain the same, regardless if lanes are in ABI or bit-interleaved representation.

Moreover, recall that the Keccak state is 1600 bits long, requiring 50 32-bit registers to store it within the register file. Consequently, to compute the Keccak permutation in Ibex, multiple passes of the state stored in memory must be performed at each round. The implementation from [Saarinen, 2022] performs 3 passes over the state per round: In the first pass the parities of each column are computed, which is used to determine the two parity bit that will be XOR-ed into each column. For each column, the XOR between the two

column parities is computed (called the $\theta$-effect in [Bertoni et al., 2012]). In the second pass the $\theta$ (Theta), $\rho$ (Rho) and $\pi$ (Pi) operations are computed by applying the $\theta$-effects to a lane (Theta), rotating the lane by the appropriate amount (Rho) and storing the result in the state array in memory in its appropriate position (Pi). The final third pass computes the $\chi$ (Chi) operation plane-wise, considering all lanes in each row.

Hardware acceleration within Keccak is provided via instructions from the *Zbkb* extension. The lane rotation operation, namely within $\rho$ (Rho) can be accelerated by `zip`, `unzip` and `pack` instructions, described in Algorithms 3.3 to 3.5, for the efficient translation between the ABI and bit-interleaved representation, as described in Algorithms 3.6 and 3.7. *Zbkb* also provides the `rori` instruction, directly implementing a rotation, otherwise implemented with two shifts and one `or` RV32I instructions. Another source of hardware acceleration is in the bitwise logical operations in the $\chi$ (Chi) step. For each lane in the state a NOT operation followed by an AND and an XOR must be performed. In plain RV32I assembly, this is implemented requiring 3 separate instructions (`XORI`, `AND` and `XOR`). The *Zbkb* extension provides the `ANDN` instruction, which negates the first operand before performing a bitwise AND between two operands, allowing to merge 2 operations in the $\chi$ (Chi) step into one.

**Algorithm 3.3** Zip instruction

**Require:** *RS*1 *and RD are registers in the register file*
  **for** *i* from 0 to 15 **do**
    $RD[2 * i] \leftarrow RS1[i]$
    $RD[2 * i + 1] \leftarrow RS1[i + 16]$
  **end for**
  **return** RD

**Algorithm 3.4** Unzip instruction

**Require:** *RS*1 *and RD are registers in the register file*
  **for** *i* from 0 to 15 **do**
    $RD[i] \leftarrow RS1[2 * i]$
    $RD[i + 16] \leftarrow RS1[2 * i + 1]$
  **end for**
  **return** RD

**Algorithm 3.5** Pack instruction

**Require:** *RS*1*, RS*2 *and RD are registers in the register file*
  $RD[15 : 0] \leftarrow RS1[15 : 0]$
  $RD[31 : 16] \leftarrow RS2[15 : 0]$
  **return** RD

The *Zbkb* extension is implemented inside the regular ALU of the Ibex processor. No internal structure can be clearly established as to enable visualization of its internals. This allows for better integration via resource-sharing with existing integer arithmetic elements within the ALU. All *Zbkb* operations are executed in a single clock cycle, with the exception of the `ror` and `rori` rotation instructions, which execute in 2 cycles.

**Algorithm 3.6** To bit-interleaved representation

**Require:** *Rl, Rh store the low and high parts of a 64-bit variable, Rt is a temporary register*
**Ensure:** *Ro, Re store the odd and even bits of a 64-bit variable in bit-interleaved form*
$Rt \leftarrow Unzip(Rl)$
$Ro \leftarrow Unzip(Rh)$
$Re \leftarrow Pack(Rt, Ro)$
$Rt \leftarrow Rt \gg 16$
$Ro \leftarrow Ro \gg 16$
$Ro \leftarrow Pack(Rt, Ro)$
**return** $Ro, Re$

**Algorithm 3.7** From bit-interleaved representation

**Require:** *Ro, Re store the odd and even bits of a 64-bit variable in bit-interleaved form, Rt is a temporary register*
**Ensure:** *Rl, Rh store the low and high parts of a 64-bit variable*
$Rt \leftarrow Pack(Re, Ro)$
$Re \leftarrow Re \gg 16$
$Ro \leftarrow Ro \gg 16$
$Rh \leftarrow Pack(Re, Ro)$
$Rl \leftarrow Zip(Rt)$
$Ro \leftarrow Zip(Ro)$
**return** $Rl, Rh$

### 3.3.3 Ascon Implementation

Ascon is designed to naturally allow for efficient software implementations. The bit-interleaving technique previously shown in Section 3.3.2 in the context of the Keccak permutation can also be applied in Ascon for speeding up rotation in the linear diffusion operation of the Ascon permutation. A published paper stemming from this work [Gewehr et al., 2024] demonstrates that the *XAscon* ISE outperforms the *Zbkb* extension at a similar area overhead, discouraging the use of bit-interleaving when implementing Ascon.

In this way, hardware acceleration is implemented through the *XAscon* ISE, originally proposed in [Cheng et al., 2022]. The *XAscon* ISE provides 10 new instructions, 5 computing the high part of each 64-bit internal state variable $x_0, ..., x_4$, and 5 computing the low part of each 64-bit internal state variable. Thus, each sigma function can be computed in only 2 instructions instead of 16 RV32I instructions in a software-only implementation.

*XAscon* instructions are also implemented in their own functional unit, shown in Figure 3.19. The Ascon unit is entirely combinational, with all *XAscon* instructions executing in a single cycle. The Ascon sigma functions lend themselves to an efficient hardware implementation, seeing as rotations by static amounts have zero logic cost. The Ascon unit does not have a well-defined internal structure like the AES unit and can be seen simply as a sea of XOR gates, computing each sigma function, and a MUX, choosing the high/low part of a

sigma function of interest. AND gates are again used to gate inputs, minimizing switching activity when the Ascon unit is not being used. The Ascon unit is integrated into the Ibex ID/EX pipeline state in the same manner as the AES unit.

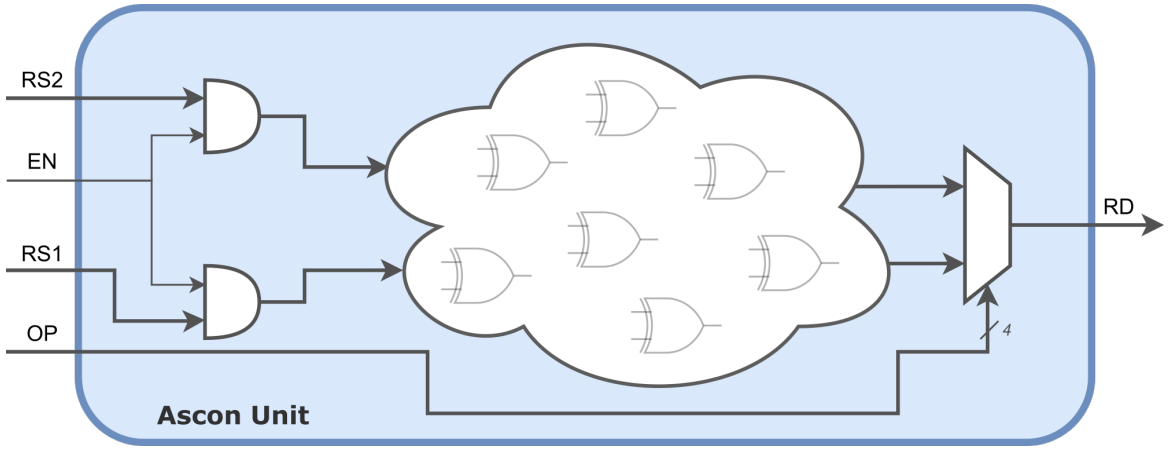


Figure 3.19 – Ascon Unit.

## 3.4 Experimental Evaluation

In this section a comparative evaluation of the SHA-256, SHA-512 (from the SHA-2 standard [NIST, 2015a]), SHA3-256, SHA3-512 (from the SHA-3 standard [NIST, 2015b]) and Ascon-Hash (from the LWC competition winning submission [Dobraunig et al., 2021]) is presented. The evaluation considers the hashing of the maximum payload size of Zigbee (86 bytes) and IPv6 (1224 bytes) packets, with and without hardware acceleration from the *Zknh* extension, for the SHA-2 algorithms; the *Zbkb* extension for the SHA-3 algorithms; and *XAscon* for the Ascon-Hash algorithm. The evaluation comprises a performance and runtime behavior comparison, energy consumption, memory footprint, and die area cost analyses.

### 3.4.1 Performance Evaluation and Profiling

Tables 3.2 and 3.3 show the performance counters for each algorithm being evaluated, with white columns showing results for the baseline software implementation and blue columns showing results for the hardware accelerated implementations. The best-performing algorithm is SHA-256, considering both software-only and hardware-accelerated implementations. The most significant performance gain is in Ascon-Hash, seeing a 49% reduction in clock cycles.

Table 3.2 – Hash functions profiling for Zigbee packet max payload size input (86 bytes).

| Counters | Zigbee packet max payload size (86 bytes) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | SHA-256 | | SHA-512 | | SHA3-256 | | SHA3-512 | | Ascon-Hash | |
| **Clock Cycles** | 14,731 | 10,511 | 19,657 | 13,679 | 33,027 | 25,185 | 63,708 | 48,024 | 25,280 | 12,980 |
| **Instructions** | 12,416 | 8,312 | 16,567 | 10,947 | 24,228 | 15,226 | 47,202 | 29,198 | 23,731 | 11,281 |
| **LSU Busy** | 1,360 | 1,352 | 2,355 | 1,843 | 6,750 | 6,746 | 13,072 | 13,064 | 696 | 846 |
| **Fetch Wait** | 464 | 356 | 328 | 482 | 941 | 1,057 | 1,510 | 1,742 | 370 | 370 |
| **Loads** | 885 | 881 | 1,470 | 1,167 | 3,961 | 3,959 | 7,647 | 7,643 | 373 | 448 |
| **Stores** | 475 | 471 | 885 | 676 | 2,789 | 2,787 | 5,425 | 5,421 | 323 | 398 |
| **Jumps** | 147 | 147 | 99 | 99 | 523 | 323 | 953 | 553 | 146 | 146 |
| **Branches** | 344 | 344 | 308 | 308 | 585 | 585 | 971 | 971 | 325 | 325 |
| **Taken Branches** | 316 | 316 | 283 | 283 | 524 | 524 | 851 | 851 | 231 | 231 |
| **Compressed Inst.** | 4,067 | 4,318 | 5,746 | 4,982 | 11,846 | 3,527 | 23,085 | 6,447 | 2,893 | 1,348 |
| **Multiplier Busy** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 12 |
| **Divider Busy** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Word** | 670 | 666 | 1,255 | 952 | 3,755 | 3,753 | 7,409 | 7,405 | 367 | 442 |
| **Load Half Word** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Byte** | 214 | 214 | 214 | 214 | 206 | 206 | 238 | 238 | 6 | 6 |
| **Store Word** | 348 | 344 | 726 | 517 | 2,669 | 2,667 | 5,273 | 5,269 | 291 | 366 |
| **Store Half Word** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Store Byte** | 126 | 126 | 158 | 158 | 120 | 120 | 152 | 152 | 32 | 32 |

Table 3.3 – Hash functions profiling for IPv6 packet max payload size input (1224 bytes).

| Counters | IPv6 packet max payload size (1224 bytes) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | SHA-256 | | SHA-512 | | SHA3-256 | | SHA3-512 | | Ascon-Hash | |
| **Cycles** | 148,049 | 106,926 | 196,576 | 135,256 | 331,010 | 252,590 | 573,098 | 431,942 | 254,540 | 124,980 |
| **Instructions** | 124,048 | 83,008 | 164,109 | 107,909 | 242,346 | 152,326 | 424,346 | 262,310 | 243,366 | 112,226 |
| **LSU Busy** | 13,648 | 13,568 | 22,590 | 17,470 | 67,336 | 67,296 | 117,240 | 117,168 | 5,814 | 7,394 |
| **Fetch Wait** | 5,282 | 5,279 | 5,754 | 5,754 | 10,038 | 11,198 | 14,142 | 16,230 | 2,497 | 2,497 |
| **Loads** | 9,317 | 9,277 | 14,951 | 11,921 | 39,973 | 39,953 | 69,013 | 68,977 | 3,072 | 3,862 |
| **Stores** | 4,331 | 4,291 | 7,639 | 5,549 | 27,363 | 27,343 | 48,227 | 48,191 | 2,742 | 3,532 |
| **Jumps** | 1,299 | 1,299 | 819 | 819 | 5,243 | 3,243 | 8,459 | 4,859 | 554 | 554 |
| **Branches** | 3,772 | 3,772 | 3,304 | 3,304 | 6,047 | 6,047 | 8,911 | 8,911 | 2,309 | 2,309 |
| **Taken Branches** | 3,690 | 3,690 | 3,252 | 3,252 | 5,455 | 5,455 | 7,847 | 7,847 | 1,938 | 1,938 |
| **Compressed Inst.** | 41,073 | 43,559 | 56,847 | 49,198 | 118,111 | 34,921 | 207,575 | 57,833 | 28,448 | 12,174 |
| **Multiplier Busy** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Divider Busy** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Word** | 6,812 | 6,772 | 12,446 | 9,416 | 37,491 | 37,471 | 66,499 | 66,463 | 3,072 | 3,862 |
| **Load Half Word** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Byte** | 2,504 | 2,504 | 2,504 | 2,504 | 2,482 | 2,482 | 2,514 | 2,514 | 0 | 0 |
| **Store Word** | 3,052 | 3,012 | 6,328 | 4,238 | 26,105 | 26,085 | 46,937 | 46,901 | 2,710 | 3,500 |
| **Store Half Word** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Store Byte** | 1,278 | 1,278 | 1,310 | 1,310 | 1,258 | 1,258 | 1,290 | 1,290 | 32 | 32 |

Table 3.4 shows relevant information concerning the core operation of each algorithm being evaluated, namely the SHA-256 and SHA-512 compression functions, the Keccak permutation and the Ascon permutation with 12 rounds. The data throughput (concerning only the core operation) for each algorithm in cycles per byte is also shown, not considering the cost of moving the operand data to/from memory.

Table 3.4 – Core operation profiling for each algorithm.

| Core Operation Metric | SHA-256 | | SHA-512 | | SHA3-256 | | SHA3-512 | | Ascon-Hash | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Clock Cycles** | 6,251 | 4,181 | 17,232 | 11,101 | 30,197 | 22,355 | 30,197 | 22,355 | 1,558 | 738 |
| **Instructions Retired** | 5,621 | 3,569 | 15,243 | 9,623 | 22,715 | 13,713 | 22,715 | 13,713 | 1,501 | 671 |
| **Cycles per byte** | 97.671 | 65.329 | 134.625 | 86.726 | 222.037 | 164.375 | 419.403 | 310.486 | 194.75 | 92.25 |

### 3.4.2 Memory Footprint Evaluation

The total memory footprint of the algorithms being evaluated is shown in Figure 3.20. Memory usage due to static data, namely the round constants for the SHA-2 and SHA-3 algorithms, are shown in yellow. Code size is shown in green, while the maximum stack usage is shown in blue. The baseline software implementations are shown with bars hatched with circles, while hardware accelerated implementations using instructions from the *Zknh*, *Zbkb*, and *XAscon* extensions are shown in bars hatched with diagonal stripes.

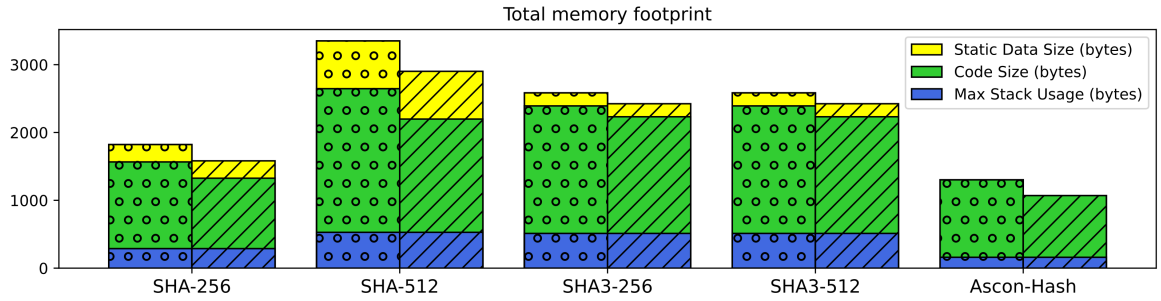


Figure 3.20 – Memory footprint of hash algorithms.

The memory footprint of all algorithms is dominated by code size, with the smallest memory footprint being of the Ascon-Hash algorithm. The use of ISEs only influences memory usage due to code size, not stack usage of static data. The static data (round constants) make a small but significant impact on the total memory footprint, namely in the SHA-512 algorithm and its 80 64-bit round constants. The Ascon round constants are determined at runtime and do not need to be pre-computed as with the other algorithms.

### 3.4.3 Energy Consumption Evaluation

The energy consumption of each algorithm for the Zigbee and IPv6 scenarios being explored are shown in Figure 3.21. Energy spent due to data and instructions memory operations are shown, respectively, in yellow and green. Energy spent due to data processing inside the Ibex processor is shown in blue. The baseline software implementations are shown with bars hatched with circles, while hardware accelerated implementations using *Zknh*, *Zbkb* and *XAscon* instructions are shown in bars hatched with diagonal stripes.

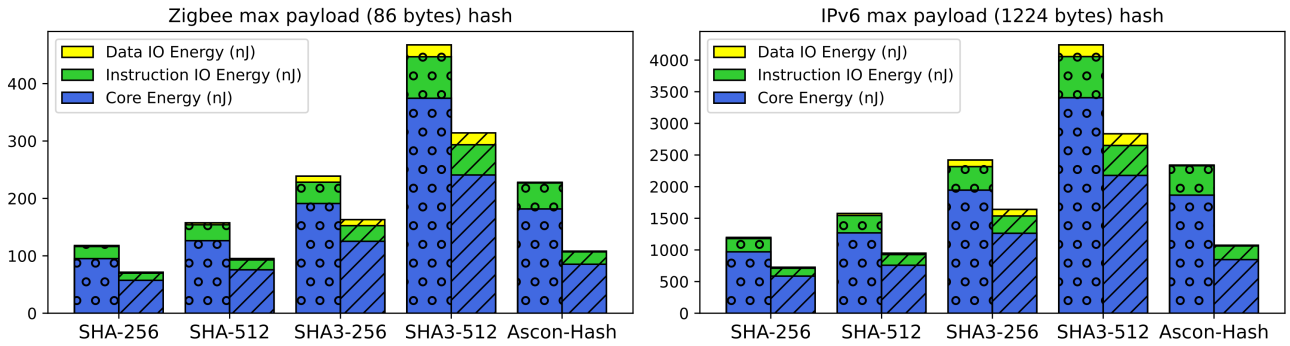


Figure 3.21 – Energy consumption of hash algorithms.

Total energy consumption is dominated by core energy due to internal processing of data, rather than moving data to/from memory. Core energy is proportional to the amount of instructions executed, with the lowest consuming algorithm being SHA-256, which is also the best performing algorithm. As with performance, the algorithm that most benefits from hardware acceleration with ISEs is Ascon-Hash from an energy consumption standpoint.

### 3.4.4 Die Area Costs Evaluation

Table 3.5 shows synthesis results for the Ibex core and each ISE being explored. Seeing as *XAscon* can be used to accelerate both hashing with Ascon-Hash and AEAD with Ascon-128, it is compared to an Ibex core with *Zkne* and *Zknh* extensions, such that a fair comparison is made comprising hardware acceleration for both AEAD and hashing. *Zbkb* instructions (namely rotations and negated operand bitwise instructions) can be used to speed up generic code as well. The *Zkne* and *Zknh* combination shows a 11% increase in cell area while *Xascon* shows a 9% increase. *Zbkb* shows a smaller 1% cell area increase, but shows the greatest increase in cell instance count at 13%.

| **Synthesis results** | Ibex baseline | Ibex + Zknh + Zkne | Ibex + Zbkb | Ibex + XAscon |
|---|---|---|---|---|
| Cell Area ($\mu m^2$) | 11,238 | 12,447 | 11,307 | 12,210 |
| Net Area ($\mu m^2$) | 6,992 | 7,278 | 5,142 | 8,181 |
| Total Area ($\mu m^2$) | 18,230 | 19,726 | 16,449 | 20,391 |
| Cell Count (# instances) | 10,289 | 11,687 | 11,769 | 11,010 |
| Equivalent NAND2 gates | 34,433 | 38,132 | 34,642 | 37,408 |
| Slack @ 500 MHz | 0 | 0 | 0 | 0 |

Table 3.5 – Area comparison of ISEs for hardware acceleration of hash functions.

## 3.5 Conclusion and Final Remarks

Considering the analyses presented in Section 3.4, no algorithm presents itself as the unanimous better choice for an AEAD algorithm in resource constrained embedded systems. SHA-256 shows better performance and energy consumption both in software only and hardware accelerated implementations, but has higher memory usage that Ascon-Hash. Using ISEs, Ascon-Hash uses 512 bytes less memory than SHA-256 (32% difference) while performing at 2469 clock cycles worse (19% difference). Additionally, Ascon-Hash can also share code for the 12 round permutation with Ascon-128, making it the better choice for memory constrained systems.

At the 256 bit security level, SHA-512 severely outperforms SHA3-512, both with and without ISEs, at higher memory usage. The proportional performance difference (66% and 69%, with and without ISEs) is greater than memory footprint difference (23% and 17%, with and without ISEs). SHA3-512 should be favored over SHA-512 in memory constrained applications, especially in those in which both 128 and 256 must be provided, seeing as both SHA3-256 and SHA3-512 can be implemented with the exact same code, only differing in runtime parameters passed as arguments, implying in a null memory footprint overhead for providing both security levels.

In summary, hardware acceleration of hash functions via ISEs bring average improvements of 32% in performance, 38% in energy consumption (Zigbee packet max payload size case study) and 16% in code size, at an average area cost of 7%, considering acceleration for both hashing and AEAD for the *Zkne* and *Zknh* or *XAscon* cases or hash functions and generic code for the *Zbkb* case.

# 4. HARDWARE ACCELERATION OF SYMMETRIC CRYPTOGRAPHY USING ISES

This chapter presents efforts made towards hardware acceleration using ISEs of symmetric cryptography, using the AES and Ascon algorithms. Section 4.1 provides an introduction to symmetric cryptography and the inner workings of the AES and Ascon algorithms. Section 4.2 reviews previous relevant works in the literature. Section 4.3 details the implementation of hardware acceleration using ISEs in the Ibex processor. Section 4.4 shows the evaluation of the previously presented implementation, following the methodology detailed in Section 2.3. Finally, Section 4.5 presents conclusions and final remarks.

Part of this chapter was published in the following conferences:

**Improving the Efficiency of Cryptography Algorithms on Resource-Constrained Embedded Systems via RISC-V Instruction Set Extensions**
Carlos Gewehr, Fernando Gehm Moraes
In: SBCCI, 2023

**Hardware Acceleration of Authenticated Encryption with Associated Data via RISC-V Instruction Set Extensions in Low Power Embedded Systems**
Carlos Gewehr, Nicolas Moura, Lucas Luza, Eduardo Bernardon, Ney Calazans, Rafael Garibotti, Fernando Gehm Moraes
In: LASCAS, 2024

## 4.1 Introduction to Symmetric Cryptography

Ensuring the confidentiality of data transmitted by embedded systems is a major concern for secure communication. This means that the information sent and received should be protected from being exposed to entities that are not authorized to access it. To achieve this, two parties who wish to communicate securely can use a common secret to transform the data in a way that only those who know the secret can decrypt it, while to others, the data appears as random noise and is unreadable.

This process requires two functions: an encryption function, to be called by the sender; and a decryption function, to be called by the receiver. The encryption function requires two inputs, the data one wishes to transmit following the confidentiality principle, commonly referred to as the plaintext, and the secret shared between the two parties, called the key. The output of this function is called the ciphertext. Once the ciphertext has been computed, it can be securely sent to the receiving party.

At the receiver side, the plaintext is obtained from the ciphertext via the decryption function, which takes in the ciphertext itself and the shared key. In this scheme, the security guarantees come from the fact that the encryption function is constructed in a manner that no operation can be performed on the ciphertext, yielding the plaintext without knowledge of the key in a way that is significantly more efficient than randomly guessing key values (brute-force attack). Consequently, the ciphertext can be safely exposed (namely through an insecure channel) to potentially malicious third parties without fear of publishing confidential information, provided that the key is known only by the legitimate transmitter and receiver. This process is illustrated in Figure 4.1, where the plaintext, ciphertext and key are denoted by the variables $x$, $y$ and $k$, respectively.

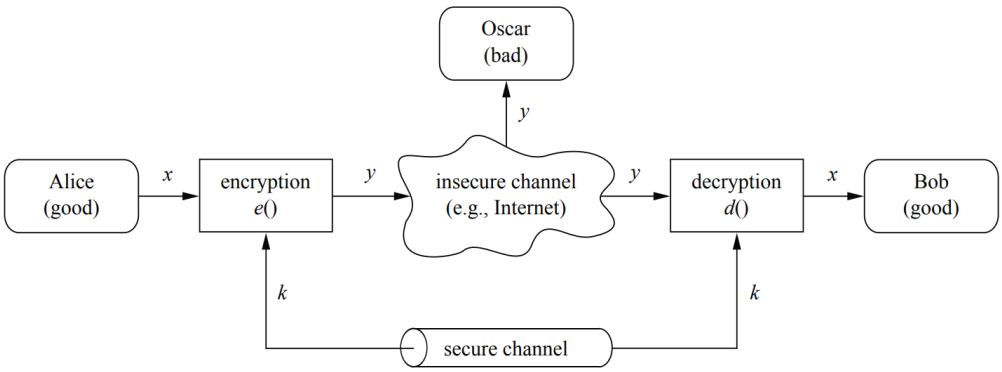


Figure 4.1 – Symmetric cryptography [Paar and Pelzl, 2009].

While confidentiality is a crucial aspect of secure communication, it alone does not ensure the integrity of the ciphertext when transmitted over an insecure channel. This means that, despite the encryption, there is no inherent guarantee that the transmitted data will remain unmodified during its transmission through potentially vulnerable networks. A ciphertext may be unintentionally modified due to transmission errors or deliberately tampered with by a malicious actor. As a result, an additional property desired in secure communication is integrity. The integrity property ensures that the receiver can determine if the ciphertext has been tampered with and disregard it if it does not match what the transmitter sent. To know whether the ciphertext obtained from a sender has been modified, redundancy is added to the ciphertext. This redundancy is called a Message Authentication Code (MAC) or Tag.

Authenticated Encryption with Associated Data (AEAD) ensures confidentiality and authenticity of sensitive data. AEAD algorithms seek to provide confidentiality and authentication for a given plaintext $P$ and authentication (but not confidentiality) of associated data $A$. A use case for AEAD is network packets, where headers ($A$) are visible to a router, but payloads are private. In addition to $A$ and $P$, the AEAD process $E$ (Equation (4.1)) expects a key $K$ and a unique nonce $N$. These additional values are used to set the initial state of the algorithm, with, for Ascon, a static initialization vector $IV$; and for AES in CCM mode, the message length, determining $CTR_0$. The outputs of the AEAD process are the ciphertext $C$ and an authentication tag $T$.

$$E(K, N, A, P) = (C, T) \tag{4.1}$$

### 4.1.1 AES

In the communication framework presented in Figure 4.1, the security assumptions rely on the premise the encryption and decryption functions are designed such that there is no computationally efficient means to obtain plaintext from a given ciphertext without knowledge of the key. The Advanced Encryption Standard (AES) [NIST, 2001a] defines a widely used algorithm for the encryption and decryption functions. The standardized algorithm has been initially published in 1998 [Daemen and Rijmen, 1998], and since then, no attacks significantly more efficient than a brute-force attack are known [Zodpe and Shaikh, 2021].

Figure 4.2 presents AES at a high level. AES is a block cipher, which accepts fixed-sized inputs and produces outputs of the same size. The block size (*Nb* 32-bit words in Figure 4.3) for AES is fixed at 128 bits. The security level of AES is defined for different levels of resistance to a brute-force attack, represented in terms of bits. For example, AES with 128 bits of security requires $2^{128}$ decryption operations with as many keys to obtain a plaintext that produces a given ciphertext. The key size (*Nk* 32-bit words) for AES is directly determined by the number of bits of security required for a specific instance of AES. Specifically, AES-128, AES-192, and AES-256 are defined to provide varying levels of security. These variants differ in the number of rounds *Nr* the core permutation of AES is performed over the plaintext: 10, 12, and 14 rounds are executed for AES-128, AES-192, and AES-256, respectively, as shown in Figure 4.3.

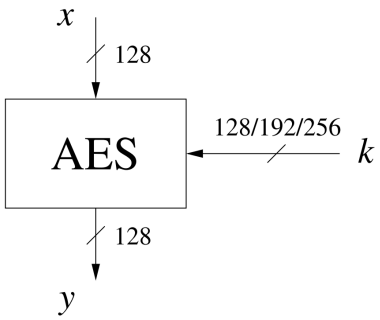


Figure 4.2 – AES encryption [Paar and Pelzl, 2009].

Internally, an AES round uses 4 operations in the following order: `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. The internal state is arranged as a 4x4 matrix of bytes.

The `SubBytes` step performs a linear transform on the GF($2^8$) inverse of each byte of the state, shown in Figure 4.4 and Figure 4.5. This operation applied on each byte is also known as a Substitution Box (SBOX).

| | Key Length *(Nk words)* | Block Size *(Nb words)* | Number of Rounds *(Nr)* |
|---|---|---|---|
| **AES-128** | 4 | 4 | 10 |
| **AES-192** | 6 | 4 | 12 |
| **AES-256** | 8 | 4 | 14 |

Figure 4.3 – AES parameters [Paar and Pelzl, 2009].

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 4.4 – AES SubBytes() linear transform (SBOX) [NIST, 2001a].

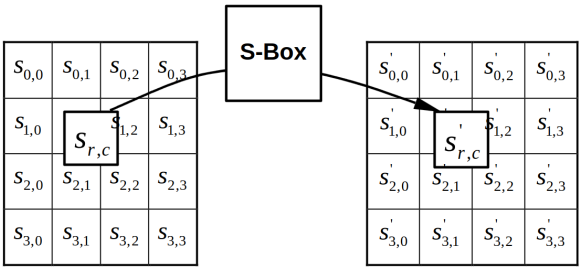


Figure 4.5 – AES SubBytes() SBOX applied to the state [NIST, 2001a].

The `ShiftRows` step shuffles the internal state, left-rotating the *i*'th row by *i* bytes to the left. This is shown in Figure 4.6.

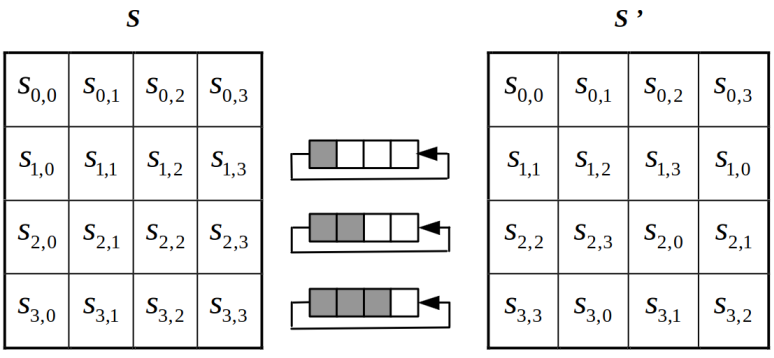


Figure 4.6 – AES ShiftRows() [NIST, 2001a].

The `MixColumns` step is performed column-wise, multiplying each column in the state with a constant matrix. This is shown in Figure 4.7.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < \boldsymbol{Nb}$$

Figure 4.7 – AES MixColumns() [NIST, 2001a].

Finally, `AddRoundKey` XORs words from the key schedule *W* into the state, shown in Figure 4.8. The key schedule is prepared before the round loop execution begins, as shown in Figure 4.9. Note that the key schedule can be computed only once for any number of AES block cipher operations with the same key (but any input plaintext).

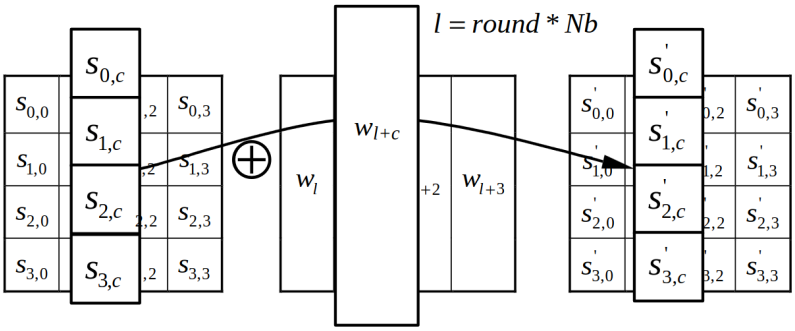


Figure 4.8 – AES AddRoundKey() [NIST, 2001a].

The use of a block cipher like AES with a secret key does not inherently guarantee confidentiality. Specifically, encrypting identical blocks of plaintext with the same key results in identical blocks of ciphertext, compromising confidentiality. This reveals information to an attacker; if two ciphertext blocks are identical, their corresponding plaintext blocks are also identical. This issue highlights the need for *modes of operation* in block ciphers, which prevent attackers from inferring plaintext information by analyzing ciphertext patterns.

In [NIST, 2001b], various standardized modes of operation are presented, including the Counter (CTR) and Cipher Block Chaining (CBC) modes. These modes are of particular interest due to their simplicity and ease of implementation. In the CTR mode, the underlying block cipher, such as AES, is utilized not to directly compute the ciphertext from the plaintext and key, but rather to generate an intermediate value based on a previously agreed running counter and the key. The resulting intermediate value is XOR-ed with the plaintext to yield the ciphertext. In the CBC mode the input to each block cipher operation is the bitwise XOR between the current plaintext block and the previous ciphertext block, while the ciphertext is taken directly from the block cipher operation. A visualization of the CTR and CBC modes of operation can be seen in Figure 4.10 and in Figure 4.11, respectively.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.   Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 4.9 – AES Key Schedule [NIST, 2001a].

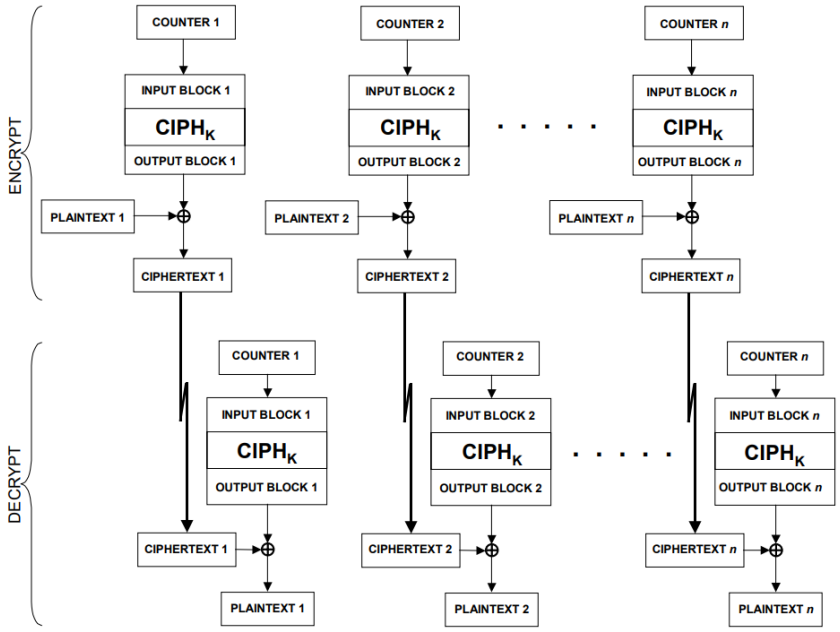


Figure 4.10 – CTR mode of operation [NIST, 2001b]

The CCM mode of operation [NIST, 2007b] employs both CTR mode for encryption and CBC mode for generating authentication tags. 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. This process can be visualized in Figure 4.12. The CCM mode of operation is interesting since it uses the underlying block cipher for both encryption and authentication, unlike other AEAD modes of operation such as GCM [NIST, 2007a], which employ other means for generating authentication tags.

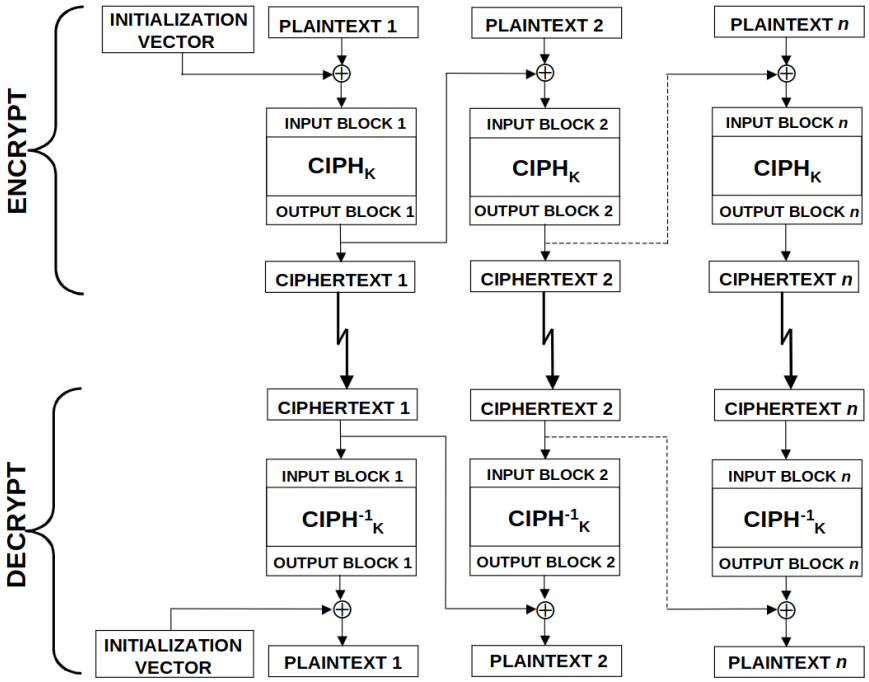


Figure 4.11 – CBC mode of operation [NIST, 2001b]

Hardware acceleration of the underlying block cipher (e.g. AES) improves encryption/decryption and authentication efficiency. Note that one AES operation is performed per associated data block, while two AES operations are performed per plaintext block.

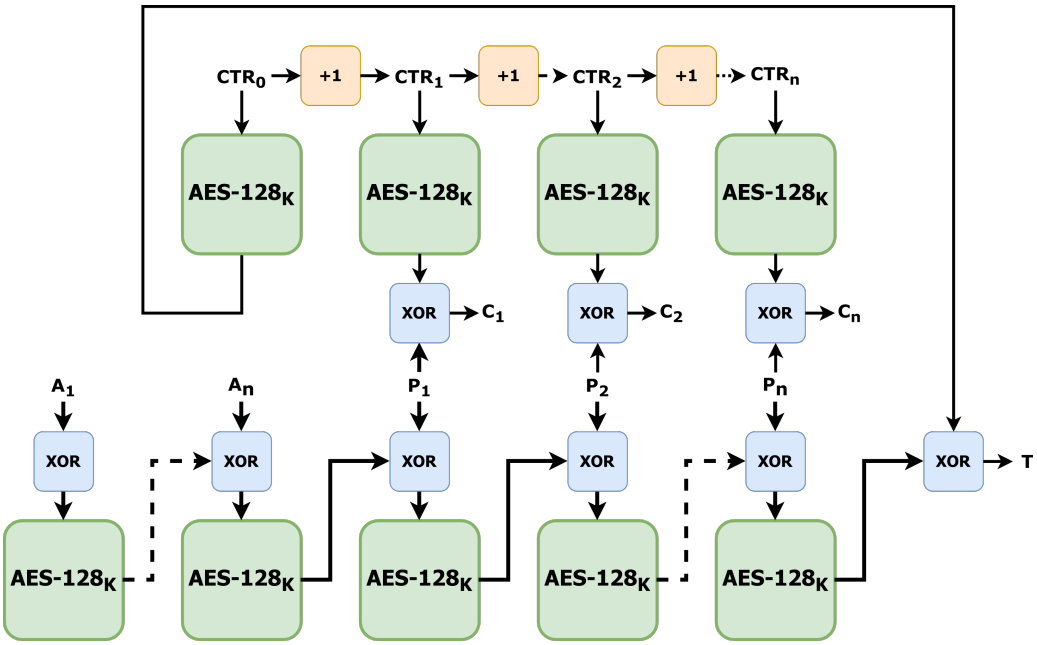


Figure 4.12 – The CCM mode of operation.

AES-128 in CCM mode is the most used algorithm for AEAD in resource-constrained embedded systems, being supported in a wide range of internet protocols, notably TLS [Rescorla, 2018] and DTLS [Rescorla et al., 2022]. For its wide and existing applicability, it was chosen as one of the algorithms for the evaluation performed in this work.

### 4.1.2 Ascon

Unlike AES used in an operation mode such as CCM, Ascon [Dobraunig et al., 2021] AEAD is based on a newer construction, called “sponge in duplex mode”, depicted in Figure 4.13. In contrast with the SHA-3 sponge demonstrated in Section 3.1.2, the duplex sponge is capable of simultaneous input and output of data, where the SHA-3 sponge must only receive data during the absorb phase, while only outputting data in the squeeze phase. *For the reader’s convenience, the exposition on the inner workings of Ascon from Section 3.1.3 is restated below.*

At each stage of the Ascon sponge, a permutation of the internal state is performed ($p^a$ and $p^b$, which differ only in the number of rounds, respectively 12 and 6). This work considers the Ascon-128 variant, with the sponge rate ($A_i$, $P_i$ and $C_i$ width) of 64 bits. Note that in Ascon, one permutation per data block absorbed is performed for both associated data and plaintext, unlike the CCM mode of operation, which computes two block cipher operations for each plaintext block.

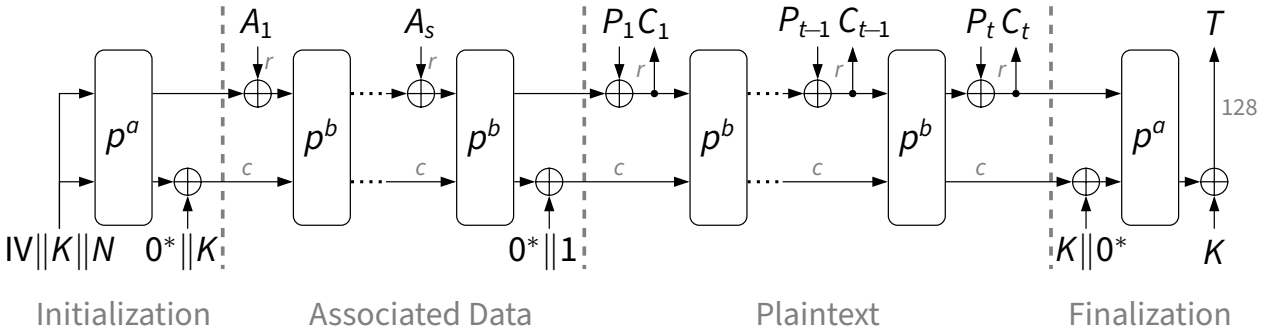


Figure 4.13 – Ascon duplex sponge [Dobraunig et al., 2021].

Internally, the Ascon permutation has a state of 320 bits, split into five 64-bit variables $x_0, ..., x_4$, shown in Figure 4.14. The permutation comprises 3 operations performed in the following order: round constant addition; the SBOX; and linear diffusion.

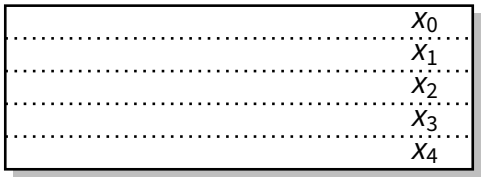


Figure 4.14 – Ascon internal state [Dobraunig et al., 2021].

The round constant addition operation is trivial, with the constant $Cr$ being determined based on the current round number $r$. $Cr$ can be determined by the subtraction of the previous round constant by 15. The initial $C0$ value varies between the $p^a$ and $p^b$ permutations. The $Cr$ value for each round and permutation is shown in Figure 4.15. The $Cr$ constant is zero-extended to the left and XOR-ed into the $x_2$ state variable.

| $p^{12}$ | $p^8$ | $p^6$ | Constant $c_r$ | $p^{12}$ | $p^8$ | $p^6$ | Constant $c_r$ |
|---|---|---|---|---|---|---|---|
| 0 | | | 00000000000000f0 | 6 | 2 | 0 | 0000000000000096 |
| 1 | | | 00000000000000e1 | 7 | 3 | 1 | 0000000000000087 |
| 2 | | | 00000000000000d2 | 8 | 4 | 2 | 0000000000000078 |
| 3 | | | 00000000000000c3 | 9 | 5 | 3 | 0000000000000069 |
| 4 | 0 | | 00000000000000b4 | 10 | 6 | 4 | 000000000000005a |
| 5 | 1 | | 00000000000000a5 | 11 | 7 | 5 | 000000000000004b |

Figure 4.15 – Ascon round constants [Dobraunig et al., 2021].

The SBOX operation consists of a series of bitwise XOR and AND operations performed between each word in the state. The SBOX operation is depicted in Figure 4.16 and its interaction with the Ascon state in Figure 4.17. Note that each bit in a state variable in the SBOX operation depends only on itself and the same relative indexed bit in the remaining state variables. In this manner, the SBOX operation can be computed for any number of bits at a time, usually the machine word size.

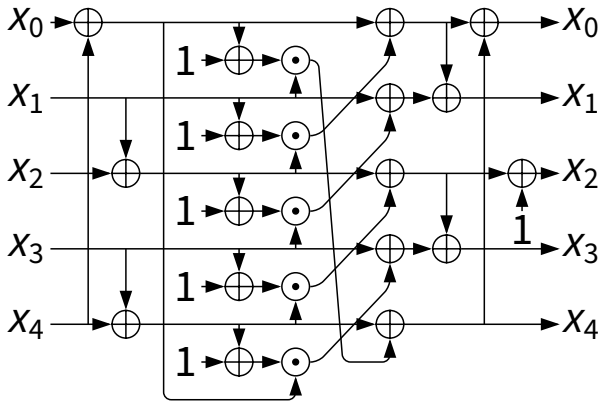


Figure 4.16 – Ascon 5-bit SBOX S(x) [Dobraunig et al., 2021].

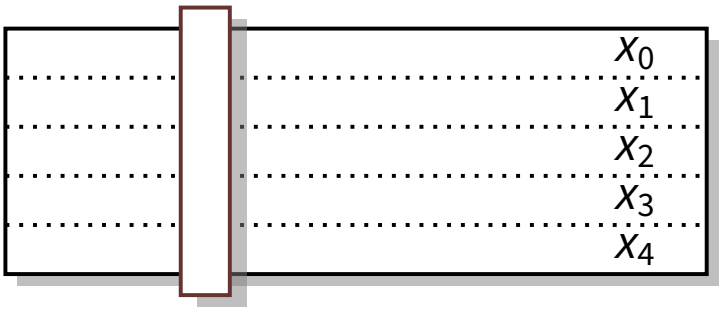


Figure 4.17 – Ascon SBOX applied to the state [Dobraunig et al., 2021].

Finally, the linear diffusion operates individually on each word of the state. Similarly to the SHA-2 sigma functions presented in Section 3.1.1, each word is determined based on the bitwise XOR of rotations of itself. Each Ascon sigma function is presented in Figure 4.18 and its interaction with the Ascon state in Figure 4.19.

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

Figure 4.18 – Ascon linear diffusion operation [Dobraunig et al., 2021].

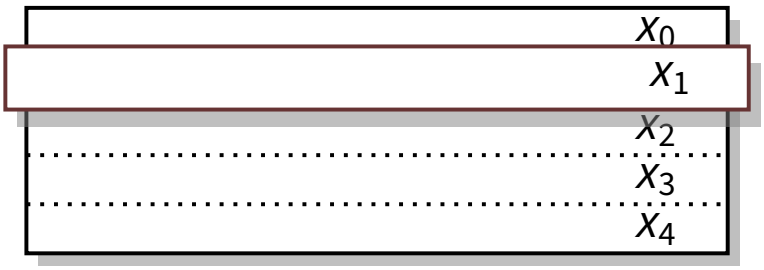


Figure 4.19 – Ascon linear diffusion applied to the state [Dobraunig et al., 2021].

Due to its novelty and upcoming standardization as a result of the NIST LWC competition, showing good performance at a low memory footprint when compared with previous NIST standards, the Ascon family of algorithms has been chosen for evaluation in this work.

## 4.2 Related Work

Previous efforts in open-source RISC-V cores have not sufficiently addressed the need for low-overhead security in resource-constrained embedded systems. The PULP project [ETH Zurich, 2016] provides several RISC-V cores and loosely coupled dedicated cryptography cores, easily integrated via an automated system generation tool. This does not provide an optimal solution for low-energy cryptography workloads due to the long logical distance between the main processor and hardware accelerators. Software productivity is also a negative factor, seeing as custom drivers must be developed for each accelerator.

The OpenTitan project [lowRISC, 2017] uses a similar approach with loosely-coupled accelerators, but uses a higher-complexity version of the Ibex core with *Zb** bit-manipulation extensions for its main processor. It does not intend to provide a platform for use in IoT applications, but in the same manner, as PULP [ETH Zurich, 2016], the use of loosely-coupled accelerators does not satisfy the constraints of the low-complexity context being explored.

The SCARV project [University of Bristol, 2018] implements a 5-stage RISC-V core with cryptography ISEs and a software library using such ISEs to achieve hardware acceleration. The SCARV core is of significantly higher complexity than the simple 2-stage Ibex core used in this work. It does not allow for a fair comparison, nor is it an adequate processor choice for the IoT context being explored. Additionally, the software library is custom-made, again leading to challenges in software productivity in integrating it into other projects.

[Zgheib et al., 2021] propose a tightly-coupled AES accelerator integrated into an Ibex core. The Authors report a 40% increase in FPGA slice usage when integrating the AES accelerator into the Ibex processor, improving performance by 46 times compared to a software implementation from OpenSSL. The authors also report their solution consumes 44.9 times less energy than OpenSSL AES. Note that this AES implementation is not ideal considering an embedded context. Seeing as the `SubBytes` operation is done in an online manner instead of via a lookup table, avoiding cache timing side-channel attacks. In a low-complexity embedded context, such attacks are not applicable as there is no cache to leak timing information. A table-based `SubBytes` such as the one used in the present work is clearly both faster and more compact, but no quantitative evidence is presented in this work to support this claim.

[Marshall et al., 2020] present a comparative analysis of several RISC-V ISE proposals, where [Saarinen, 2020] work is demonstrated to be the superior extension to the 32-bit base ISA. The ISE proposed by [Saarinen, 2020] was later standardized and ratified as the *Zkne* and *Zknd* extensions in the RISC-V scalar cryptography as the ISE for AES encryption and decryption, respectively. Synthesis results are provided in terms of NAND2 equivalent gates. In the context of a SCARV [University of Bristol, 2018] core, the [Saarinen, 2020] ISE shows a 3% area increase, accelerating both encryption and decryption. [Saarinen, 2020] reports a 5% increase in LUT usage in an FPGA-based evaluation considering a lower complexity single-cycle RISC-V core. No power or energy results are provided in [Marshall et al., 2020; Saarinen, 2020].

In the software side of the evaluation in [Marshall et al., 2020], [Saarinen, 2020] shows 3.5x and 1.65x performance gains for AES-128 encryption and encryption key expansion, respectively. The software comparisons consider a higher-performance T-Table with 4 KB LUT implementation, trading-off memory usage for performance.

Further software evaluations concerning the entire RISC-V cryptography ISE (not only *Zkne* as discussed so far) are reported in [Nişancı et al., 2022; Marshall et al., 2021]. [Nişancı et al., 2022] presents an analysis of program and static data memory usage for accelerated and non-accelerated implementations of cryptography algorithms, prioritizing execution speed in their implementations. The Authors show a reduction of 83% in program memory usage in AES-128, but do not seem to consider a reduction in static data memory usage in their accelerated implementation, nor stack usage in their evaluation.

Despite reporting performance in terms of clock cycles instead of instructions retired, the authors of [Nişancı et al., 2022] do not sufficiently describe their hardware platform, simply stating their processor core to be "a 32-bit 5-stage pipelined RISC-V processor that only supports base integer instructions". Crucial facts concerning the complexity of the RISC-V core used in their evaluation such as operand forwarding, branch prediction, and multiplier and shifter topology are omitted. This makes it difficult to compare results, seeing as the complexity of their RISC-V core cannot be correlated to the Ibex core.

[Campos et al., 2020] perform a comparative analysis of several algorithms implemented in C or RISC-V assembly, showing that equivalent C implementations (with the same optimization techniques) generally match or slightly outperform assembly implementations. For AES, the authors evaluate a performance-optimized T-Table implementation, with a 4KB LUT and loop unrolling. This implementation are unapplicable to the IoT context due to severe static data size and code size costs. In this work we consider AES implementations from the TinyCrypt [Intel, 2017] library, with a 256 Bytes SBOX LUT, and a custom T-Table implementation with a 1 KB LUT, neither using loop unrolling.

*[Campos et al., 2020; Cheng et al., 2022] have been previously discussed in the context of Ascon in Section 3.2, restated here for the reader's convenience.*

[Campos et al., 2020] also consider Ascon in their evaluation, considering an inlined ASM implementation of the Ascon permutation, and one permutation using bit-interleaving, described in details in Section 3.1.2. All implementations use loop unrolling.

[Cheng et al., 2022] propose ISEs for several LWC algorithms, among them Ascon. In the case of Ascon, they propose a new custom extension called *XAscon* and evaluate it considering software-only and *Zbkb*-accelerated implementations. The evaluation for the *XAscon* extensions makes use of both *XAscon* and *Zbkb*. The *XAscon* + *Zbkb* implementation shows 2.28x and 1.18x performance gains compared to software-only and *Zbkb*-only implementations, respectively. Authors perform their work in the context of the Rocket core [Chips Alliance, 2023], with 5 pipeline stages and floating point instructions, more complex than expected for an IoT context. Hardware-specific metrics are evaluated in FPGAs but not ASIC implementations. The *XAscon* + *Zbkb* and *Zbkb*-only extended Rocket RV32GC processors come at an area cost of 1.282x and 1.140×, respectively, measured in FPGA LUTs used (Xilinx Kintex-7 xc7k160tfbg676).

In the same manner as in Section 3.2, it is clear that ISEs in the context of low-complexity embedded systems are an under-explored field of study. Most works concern themselves with high-complexity processors, where area costs due to cryptography ISEs are of little relevance to the overall area cost of the processor; or conduct their evaluations using FPGA or open-source EDA tools, not showing accurate values representative of factual ASIC implementations, using commercial EDA tools. Additionally, this work considers a comprehensive evaluation of memory overheads and runtime behaviour in terms of clock cycles, instructions executed and memory accesses for its software aspects.

## 4.3 Implementation

This section presents the efforts made towards the efficient software-only and hardware accelerated implementation of AES and Ascon for AEAD in the Ibex processor.

### 4.3.1 AES Implementation

Recall the SBOX in the AES `SubBytes` step. A common AES optimization technique is to pre-compute each SBOX value and store them in a Look-Up Table (LUT). The SBOX LUT comprises 256 bytes, and is equivalent in size to 64 RV32I instructions. Assuming that online computing `SubBytes` requires more than 64 RV32I instructions, this can be seen as an opportunity for optimization for both performance and code size.

Seeing as the `ShiftRows` and `MixColumns` operations perform shifts and multiplications by constants, this reasoning can be taken one step further by pre-computing not the result of `SubBytes`, but of `MixColumns`. In contrast to an S-Table implementation, which uses pre-computed values only for `SubBytes`, this approach is commonly known as a T-Table implementation, named following $T_0, T_1, T_2, T_3$ as defined below, where $d_{i,j}$, $a_{i,j}$ and $S$ denote the output of `MixColumns`, the round's initial state and the SBOX LUT at row $i$ and column $j$, respectively:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_0] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} = T_0[a_0] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}]$$

$$T_0[x] = \begin{bmatrix} S[x] \cdot 2 \\ S[x] \\ S[x] \\ S[x] \cdot 3 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \cdot 3 \\ S[x] \cdot 2 \\ S[x] \\ S[x] \end{bmatrix} \quad T_2[x] = \begin{bmatrix} S[x] \\ S[x] \cdot 3 \\ S[x] \cdot 2 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \cdot 3 \\ S[x] \cdot 2 \end{bmatrix}$$

Note that each T-Table is a 1-to-4 byte mapping, instead of the previous 1-to-1 byte mapping using S-Tables. This increases memory usage from 256 to 4K bytes, but simplifies an entire AES round to 16 LUT lookups and 16 bitwise XOR operations. The performance to code size trade-off can be balanced by pre-computing only 1 T-Table and computing the remaining table entries online via rotations, at 1 KB memory cost instead of 4KB if storing the 4 T-Tables. This is the approach used in this evaluation.

Hardware acceleration of AES is implemented through two new instructions defined in the *Zkne* extension: `aes32esmi` and `aes32esi`. The `aes32esmi` instruction can be interpreted as computing T-Table entries online in hardware, then XOR-ing the current entry with previous T-Table entries for the same output column. This provides the performance benefits of a T-Table AES implementation without the need for storing LUTs in memory, leading to gains in performance and code size, as well as energy efficiency gains via the reduction in overall instructions executed and memory accesses. Finally, `aes32esi` performs only a single SBOX lookup and XOR, used in computing round keys and the last encryption round, with no `MixColumns`.

The *Zkne* instructions are implemented in the AES Unit module, shown in Figure 4.20. The AES unit is entirely combinational, executing either the `aes32esmi` and `aes32esi` instructions in a single clock cycle. Both instructions are of an extended R type, which expects as inputs 2 registers to be read from the register file, plus a 2-bit immediate Byte Select (*BS*). AND gates are used to gate inputs, minimizing switching activity in the AES unit when it is not being used.

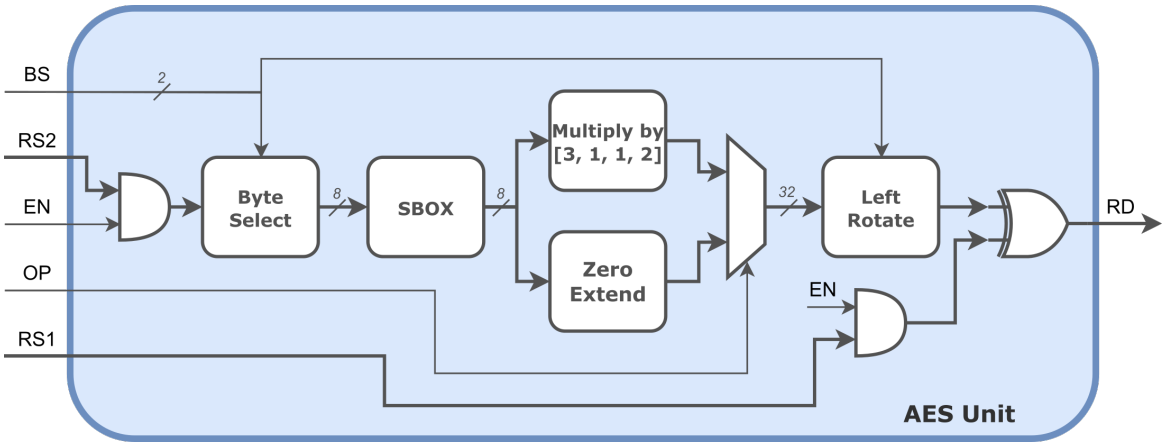


Figure 4.20 – AES Unit [Gewehr and Moraes, 2023]

Note how sub-modules in the AES Unit relate to AES fundamental operations: `SubBytes` is performed one byte at a time in the SBOX sub-module, on the byte determined by *BS* in the value read from *RS2*; `ShiftRows` is computed by selecting the appropriate byte from *RS2*, which contains a column of the current AES state, via the *BS* immediate; `MixColumns` is computed similarly to the T-Table method shown previously, where the result of `SubBytes` is multiplied by [3, 1, 1, 2] and rotated by *BS* bytes, computing $T_{BS}$ accounting for RISC-V little-endianess; `AddRoundKey` is computed by XOR-ing the result of `MixColumns` to the relevant word of the round key given in *RS1*. *RS1* should be equal to *RD*, such that *RD* accumulates the 4 T-Table entries and round key i.e. a column of the next round state.

The integration of the AES (and SHA-2) functional unit in the Ibex processor's ID/EX pipeline stage can be seen in Figure 4.21. New elements added to the processor are colored in blue, existing elements modified to accommodate the new functional units are colored in green, while unmodified elements are colored in yellow.

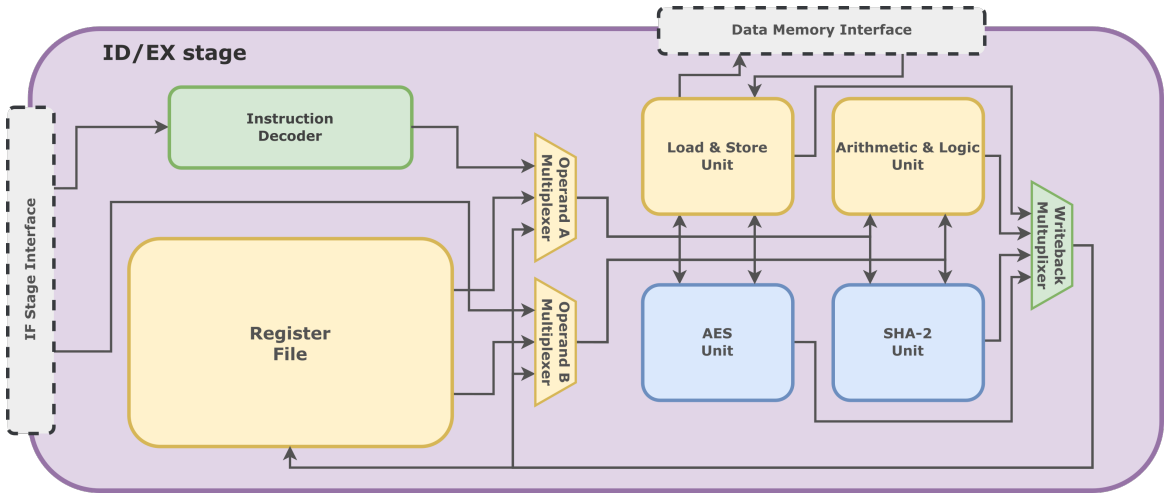


Figure 4.21 – Ibex ID/EX pipeline stage datapath with AES and SHA-2 functional units.

Figure 4.21 also illustrates the benefits of implementing hardware acceleration as ISEs over loosely coupled accelerators in embedded contexts. Note that only minor changes in the instruction decoding and writeback logic are required to integrate functional units into the Ibex processor, while resource sharing with existing processor elements is extensively explored, namely in the register file and load/store unit.

### 4.3.2 Ascon Implementation

Compared with AES, Ascon is designed to naturally allow efficient software implementations without LUTs. No elaborate techniques, such as AES T-Tables, are needed. *For the reader's convenience, details regarding Ascon implementation from Section 3.3.3 are restated here.*

Hardware acceleration is implemented through the *XAscon* ISE, originally proposed in [Cheng et al., 2022]. The *XAscon* ISE provides 10 new instructions, 5 computing the high part of each 64-bit internal state variable $x_0, ..., x_4$, and 5 computing the low part of each 64-bit internal state variable. In this manner, each sigma function can be computed in only 2 instructions, instead of 16 RV32I instructions in a software-only implementation.

*XAscon* instructions are also implemented in their own functional unit, shown in Figure 4.22. The Ascon unit is entirely combinational, with all *XAscon* instructions executing in a single cycle. The Ascon sigma functions lend themselves to an efficient hardware implementation, seeing as rotations by static amounts have zero logic cost. The Ascon unit does not have a well-defined internal structure like the AES unit, and can be seen simply as a sea of XOR gates, computing each sigma function, and a MUX, choosing the high/low part of a sigma function of interest. AND gates are again used to gate inputs, minimizing switching activity when the Ascon unit is not being used. The Ascon unit is integrated into the Ibex ID/EX pipeline state like the AES unit.

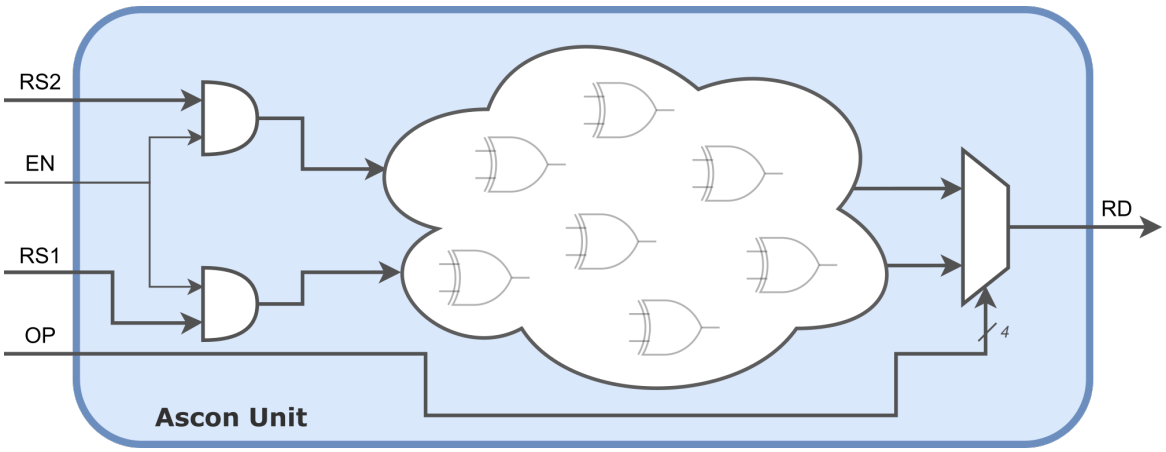


Figure 4.22 – Ascon Unit.

## 4.4 Experimental Evaluation

In this section a comparative evaluation of the AES-128 and AES-256 in CCM mode and Ascon-128 is presented. The evaluation considers the encryption with associated data with maximum payload size of Zigbee (A = 25 bytes, P = 86 bytes) and IPv6 (A = 40, P = 1224 bytes) packets, with and without hardware acceleration from the *Zkne* extension, for AES-128 and AES-256; and *XAscon* for Ascon-128. The evaluation comprises a performance and runtime behaviour comparison as well as energy consumption, memory footprint and die area cost analyses.

### 4.4.1 Performance Evaluation and Profiling

Table 4.1 shows relevant information concerning the core operation of each algorithm being evaluated, namely the AES-128 and AES-256 block ciphers, the Ascon permutation with 6 and 12 rounds. White columns show results for the baseline software and blue columns show results for the hardware-accelerated implementations.

Table 4.1 – Core operation profiling for AEAD algorithms.

| **Core Operation Metric** | AES-128 | | AES-256 | | Ascon (6 rounds) | | Ascon (12 rounds) | |
|---|---|---|---|---|---|---|---|---|
| **Clock Cycles** | 1,490 | 311 | 2,056 | 417 | 814 | 414 | 1,558 | 738 |
| **Instructions Retired** | 1,234 | 240 | 1,722 | 326 | 769 | 359 | 1,501 | 671 |
| **Cycles per byte** | 93.125 | 19.437 | 128.5 | 26.062 | 101.75 | 51.75 | 194.75 | 92.25 |

Tables 4.2 and 4.3 show the performance counters for each algorithm. The best-performing algorithm is Ascon-128, considering both software-only and hardware-accelerated implementations. The most significant performance gains are in the AES algorithms, with a 63% performance gain.

Table 4.2 – AEAD profiling for Zigbee packet max payload size (A = 25 bytes, P = 86 bytes).

| Counters | Zigbee packet max payload size (A = 25 bytes, P = 86 bytes) | | | | | |
|---|---|---|---|---|---|---|
| | AES-128 | | AES-256 | | Ascon-128 | |
| **Cycles** | 30,286 | 11,099 | 39,350 | 12,846 | 16,919 | 9,679 |
| **Instructions** | 23,564 | 7,448 | 31,486 | 8,855 | 15,491 | 8,091 |
| **LSU Busy** | 4,466 | 1,692 | 5,779 | 1,965 | 777 | 937 |
| **Fetch Wait** | 1,290 | 1,055 | 1,052 | 1,087 | 266 | 266 |
| **Loads** | 3,983 | 1,296 | 5,280 | 1,553 | 388 | 468 |
| **Stores** | 483 | 396 | 499 | 412 | 389 | 469 |
| **Jumps** | 179 | 193 | 185 | 193 | 125 | 125 |
| **Branches** | 787 | 711 | 848 | 746 | 242 | 242 |
| **Taken Branches** | 560 | 493 | 621 | 523 | 140 | 140 |
| **Compressed Inst.** | 11,039 | 1,789 | 14,847 | 1,833 | 2,275 | 1,383 |
| **Multiplier Busy** | 0 | 0 | 0 | 0 | 18 | 18 |
| **Divider Busy** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Word** | 3,231 | 842 | 4,514 | 1,102 | 381 | 461 |
| **Load Half Word** | 1 | 1 | 1 | 1 | 0 | 0 |
| **Load Byte** | 751 | 453 | 765 | 450 | 7 | 7 |
| **Store Word** | 240 | 155 | 255 | 171 | 367 | 447 |
| **Store Half Word** | 9 | 9 | 9 | 9 | 0 | 0 |
| **Store Byte** | 234 | 232 | 235 | 232 | 22 | 22 |

Table 4.3 – AEAD profiling for IPv6 packet max payload size (A = 40 bytes, P = 1224 bytes).

| Counters | IPv6 packet max payload size (A = 40 bytes, P = 1224 bytes) | | | | | |
|---|---|---|---|---|---|---|
| | AES-128 | | AES-256 | | Ascon-128 | |
| **Cycles** | 298,741 | 110,419 | 387,519 | 127,576 | 147,190 | 81,950 |
| **Instructions Retired** | 232,045 | 73,787 | 309,751 | 87,492 | 136,950 | 70,100 |
| **LSU Busy** | 43,156 | 16,072 | 55,909 | 18,633 | 6,520 | 8,130 |
| **Fetch Wait** | 13,685 | 11,196 | 11,365 | 11,766 | 1,682 | 1,682 |
| **Loads** | 39,024 | 12,742 | 51,761 | 15,287 | 3,260 | 4,065 |
| **Stores** | 4,132 | 3,330 | 4,148 | 3,346 | 3,260 | 4,065 |
| **Jumps** | 1,618 | 1,775 | 1,624 | 1,775 | 520 | 520 |
| **Conditional Branches** | 8,237 | 7,589 | 8,870 | 7,910 | 1,514 | 1,514 |
| **Taken Conditional Branches** | 6,215 | 5,576 | 6,848 | 5,892 | 1,000 | 1,000 |
| **Compressed Instructions** | 106,960 | 15,483 | 145,088 | 16,385 | 19,279 | 11,282 |
| **Multiplier Busy** | 0 | 0 | 0 | 0 | 4 | 4 |
| **Divider Busy** | 0 | 0 | 0 | 0 | 0 | 0 |
| **Load Word** | 31,402 | 7,706 | 44,125 | 10,254 | 3,260 | 4,065 |
| **Load Half Word** | 1 | 1 | 1 | 1 | 0 | 0 |
| **Load Byte** | 7,621 | 5,035 | 7,635 | 5,032 | 0 | 0 |
| **Store Word** | 1,527 | 727 | 1,542 | 743 | 3,244 | 4,049 |
| **Store Half Word** | 80 | 80 | 80 | 80 | 0 | 0 |
| **Store Byte** | 2,525 | 2,523 | 2,526 | 2,523 | 16 | 16 |

Table 4.4 shows throughput for associated data and plaintext for each AEAD algorithm. AES-128 in CCM mode throughputs for Associated Data are much greater than Ascon's, while Plaintext throughputs favor Ascon in the software only case and AES-128 in CCM mode in the hardware accelerated case. Note that AES-128 in CCM mode with hardware acceleration having better core operation throughputs does not translate to a better overall performance as shown in Tables 4.2 and 4.3. The TinyCrypt implementation of CCM mode targets portability as one of its main concern, and as such the data IO is based on arrays of bytes. The Ascon implementation uses 32 bit wide loads and stores up until array sizes are less than 4 bytes. This can be visualized in Tables 4.2 and 4.3 in the Load Byte and Load Word rows. This difference in the way data is manipulated accounts for AES-128 in CCM mode having performance than Ascon in the hardware accelerated case despite having better core operation throughputs.

Table 4.4 – AEAD throughput considering core operations (smaller is better).

| **Metric** | | AES-128 CCM | | AES-256 CCM | | Ascon | |
|---|---|---|---|---|---|---|---|
| AD | Cycles per byte | **93.125** | **19.437** | 128.5 | 26.062 | 101.75 | 51.75 |
| | Instructions per byte | **77.125** | **15** | 107.625 | 20.375 | 96.125 | 44.875 |
| P | Cycles per byte | 186.25 | **38.874** | 257 | 52.124 | **101.75** | 51.75 |
| | Instructions per byte | 154.25 | **30** | 215.25 | 40.75 | **96.125** | 44.875 |

### 4.4.2 Memory Footprint Evaluation

The total memory footprint of the algorithms being evaluated are shown in Figure 4.23. Memory usage due to static data, namely the AES T-Table, is shown in yellow. Code size is shown in green, while the maximum stack usage is shown in blue. The baseline software implementations are shown with bars hatched with circles, while hardware accelerated implementations using instructions from the *Zkne* and *XAscon* extensions are shown in bars hatched with diagonal stripes.

The memory footprint of all algorithms is dominated by code size, with the smallest memory footprint being of the Ascon-128 between software-only implementations and AES-128 for the hardware-accelerated implementations. Unlike as previously seen with hash functions, in which ISEs only influence memory usage due to code size, the *Zkne* extension also eliminates the need for storing a pre-computed AES T-Table in memory. The static data (1 KB AES T-Table) makes a significant impact on the total memory footprint. The Ascon round constants are determined at runtime and do not need to be pre-computed as with the other algorithms. For extremely memory-constrained applications, the 12 round and 6 round permutations could be merged into a single function (implementation shown in Figure 4.23 implements the two permutation in separate functions), reducing code size by approximately 340 bytes, bringing its hardware accelerated total memory footprint to roughly the same as hardware accelerated AES-128 in CCM mode.

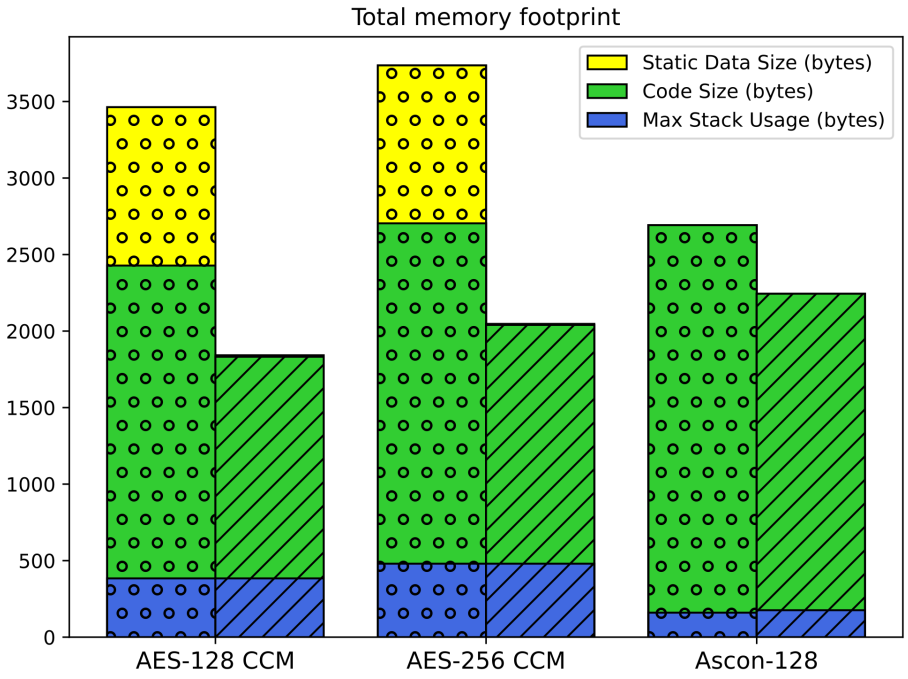


Figure 4.23 – Memory footprint of AEAD algorithms.

### 4.4.3 Energy Consumption Evaluation

The energy consumption of each algorithm for the Zigbee and IPv6 scenarios being explored are shown in Figure 4.24. Energy spent due to data and instructions memory operations are shown, respectively, in yellow and green. Energy spent due to data processing inside the Ibex processor is shown in blue. The baseline software implementations are shown with bars hatched with circles, while hardware accelerated implementations using instructions from the *Zkne* and *XAscon* extensions are shown in bars hatched with diagonal stripes.

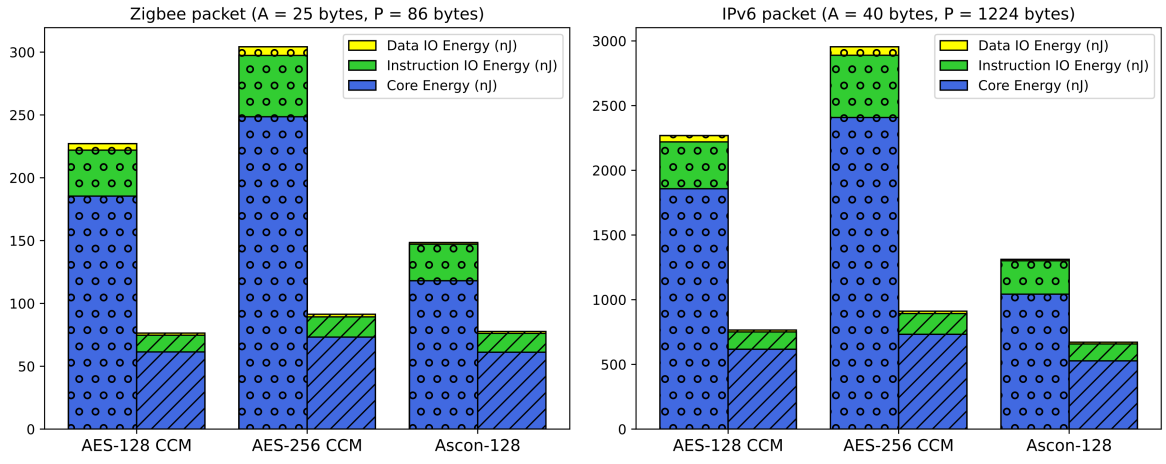


Figure 4.24 – Energy consumption of AEAD algorithms.

Total energy consumption dominated by core energy, due to internal processing of data, rather than moving data to/from memory. Core energy is proportional to the amount of instructions executed, with the lowest consuming algorithm being SHA-256, which is also the best performing algorithm. As with memory usage, AES-128 and AES-256 benefit more from hardware acceleration than Ascon-128 from an energy consumption standpoint.

### 4.4.4 Die Area Costs Evaluation

Table 4.5 shows synthesis results for the Ibex core and each ISE being explored. The *Zkne* and *Zknh* combination showing a 11% increase and *Xascon* showing a 9% increase.

| **Synthesis results** | Ibex baseline | Ibex + Zknh + Zkne | Ibex + XAscon |
|---|---|---|---|
| Cell Area ($\mu m^2$) | 11,238 | 12,447 | 12,210 |
| Net Area ($\mu m^2$) | 6,992 | 7,278 | 8,181 |
| Total Area ($\mu m^2$) | 18,230 | 19,726 | 20,391 |
| Cell Count (# instances) | 10,289 | 11,687 | 11,010 |
| Equivalent NAND2 gates | 34,433 | 38,132 | 37,408 |
| Slack @ 500 MHz | 0 | 0 | 0 |

Table 4.5 – Area comparison of ISEs for hardware acceleration of AEAD.

## 4.5 Conclusion and Final Remarks

Considering the analyses presented in Section 4.4, at the 128 bit security level Ascon-128 can be seen as a better choice for an AEAD algorithm in resource constrained embedded systems. Ascon-128 shows better performance both in software only and hardware accelerated implementations, showing similar energy consumption and ISE area costs as AES-128 in CCM mode. Memory footprint severely favors Ascon-128 for software only implementations, while for hardware accelerated implementations they are roughly the same.

If upgrading a system to the 256 bit security level, the use of AES-256 is required. Using hardware accelerated AES-256 in CCM mode brings significant improvements in performance, memory footprint and energy consumption when compare to a software implementation of AES-128 in CCM mode, at an area cost to the order of 10% of the baseline Ibex area (with no ISEs) or 4 kGE.

In summary, hardware acceleration of AEAD via ISEs bring average improvements of 58% in performance, 61% in energy consumption (Zigbee packet case study) and 35% in total memory footprint, at an average area cost of 10%, considering acceleration for both hashing and AEAD.

# 5. HARDWARE ACCELERATION OF ASYMMETRIC CRYPTOGRAPHY USING ISES

This chapter presents efforts made toward hardware acceleration using ISEs for asymmetric cryptography, using the Crystals-Kyber algorithm. Section 5.1 introduces asymmetric cryptography and the inner workings of Kyber. Section 5.2 reviews previous relevant works in the literature. Section 5.3 details the implementation of hardware acceleration using ISEs in the Ibex processor. Section 5.4 shows the evaluation of the implementations following the methodology detailed in Section 2.3. Finally, Section 5.5 presents conclusions and final remarks.

## 5.1 Introduction to Asymmetric Cryptography

Symmetric cryptography was described in Section 4.1 as a manner in which two parties can communicate while keeping the contents of the messages sent between each other secret (the confidentiality principle). Using symmetric cryptography, Alice can send Bob an encrypted message using a shared secret key. Bob can use the same key to decrypt the message sent by Alice. Note that the same key is used to both encrypt and decrypt.

For the scenario described above, in asymmetric cryptography, different keys for encryption and decryption would be used. As in symmetric cryptography, a well-designed asymmetric algorithm, the only way to determine plaintext from ciphertext is with knowledge of the decryption key. Note that to maintain the confidentiality principle in this scheme, only the decryption key must be kept secret, while the encryption key can be broadcast publicly.

This process can be visualized in Figure 5.1. If Alice wants to send Bob a secret message, she must first obtain from Bob his encryption key $k_{pub}$. As there is no need to keep it secret, the encryption key is also called the public key. Once Alice has $k_{pub}$, she can encrypt the plaintext $x$ with $k_{pub}$, which yields the ciphertext $y$. $y$ can be decrypted by Bob using the decryption key $k_{pr}$. Since $k_{pr}$ is not sent to Alice or exposed to any third parties (to maintain confidentiality), the decryption key is also called the private key. Note that $k_{pub}$ and $k_{pr}$ are generated as a pair. Bob can only decrypt ciphertexts using $k_{pr}$ if the plaintext was encrypted using the related $k_{pub}$.

One of the main advantages of asymmetric schemes is that no previous secure channel is needed. In contrast with symmetric cryptography, the shared secret key must be either known in advance or sent between both communicating parties through a separate secure channel. Using asymmetric cryptography, no secure channel is needed, and all encrypted communication can take place in a non-secure channel without compromising the secrecy of messages sent.

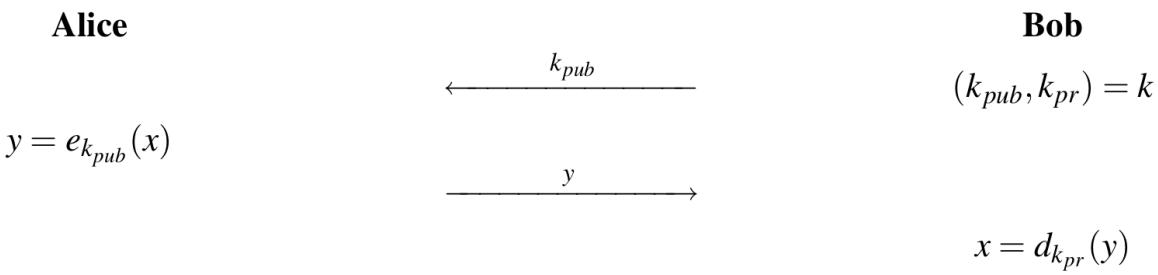


Figure 5.1 – Asymmetric cryptography [Paar and Pelzl, 2009].

In computational performance terms, traditional asymmetric algorithms such as RSA and Elliptic Curve-based schemes are much less performant than symmetric schemes while having much bigger key sizes, incurring costs in both processing time and memory usage. A common usage of asymmetric cryptography is in conjunction with symmetric cryptography, where an asymmetric algorithm is used not to send and receive general data messages, but to send a random and secret shared key, which both parties will use to communicate securely using a symmetric cryptography algorithm such as AES. This usage example is illustrated in Figure 5.2. Note that this scheme is vulnerable to Man-in-the-Middle attacks if the identities of Alice and Bob are not previously authenticated, which falls out of the scope of this work.

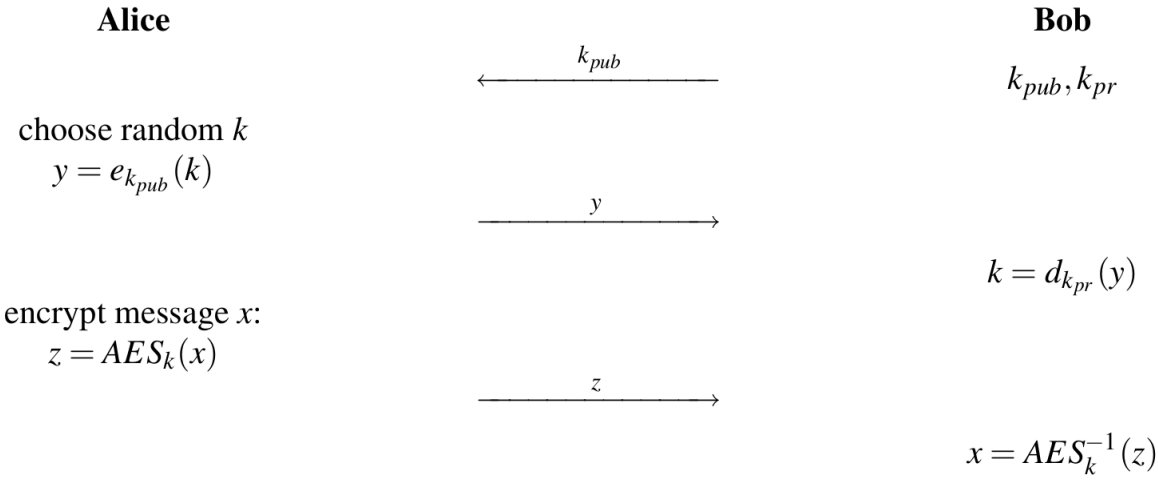


Figure 5.2 – Asymmetric cryptography usage for key establishment [Paar and Pelzl, 2009].

An asymmetric algorithm used to send a shared key for symmetric encryption is called a Key Encapsulation Mechanism (KEM). Traditional KEM methods, such as RSA and Elliptic Curves, have been deemed insecure with the advent of Shor's algorithm [Shor, 1999]. This algorithm presents a polynomial-time solution, on a quantum computer, to the underlying mathematical problems on which the security of these constructions is predicated. Thus, novel approaches must be developed to establish secure keys that can withstand both classical and quantum-based attacks.

### 5.1.1 Crystals-Kyber (ML-KEM)

Among the various mathematical problems suitable for constructing quantum-secure cryptographic schemes, those involving lattice structures are of particular interest. A $n$-dimensional lattice can be understood as a discrete subset of a $n$-dimensional vector space. Like vector spaces, lattices are defined by $n$ linearly independent $n$-dimensional vectors. Such a set of $n$-dimensional vectors are called a basis of the lattice. In cryptography applications, it is common to consider only integer lattices, in which the basis and all members of the lattices have integer coefficients. Furthermore, coefficients $c_i$ are usually restricted to a $0 < c_i < q$ subset of the integers, where $q$ is determined by the specific application. Operations involving coefficients are always performed modulo $q$ for each coefficient.

Formally, let $\mathbf{v} \in \mathbb{Z}_q^n$ to be a $n$-dimensional vector whose coefficients $c_i \in (0, 1, ..., q-1)$ and $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ a matrix of $n$ vectors $\in \mathbb{Z}_q^n$. Throughout this text, matrices will be represented by uppercase bold letters while vectors will be represented by lowercase bold letters.

One of the lattice problems of interest to cryptography is the Closest Vector Problem (CVP): given a basis for a lattice and a point in the vector space of the same dimensions, the goal is to find the point in the lattice that is closest to the given point in the vector space. Despite showing resistance to classical and quantum-based attacks, CVP has no immediate applications in cryptography, but is crucial to the understanding of higher complexity problems that are used in practical applications, such as Learning With Errors (LWE).

The LWE problem closely resembles asymmetric cryptography, seeing as it is integral to the problem definition that some pieces of data are public while others are private: knowing a public basis matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ and a vector $\mathbf{t} \in \mathbb{Z}_q^n$, the goal is to find a secret $\mathbf{s} \in \mathbb{Z}_q^n$ such that $\mathbf{As} + \mathbf{e} = \mathbf{t}$, where $\mathbf{e}$ is sampled from a random Gaussian error distribution. Note that the LWE problem can be interpreted as forcing a CVP in a lattice of choice: $\mathbf{As}$ gives a point in the lattice defined by $\mathbf{A}$. When $\mathbf{As}$ is added to the random error distribution $\mathbf{e}$, this offsets the point in the lattice by a small amount, such that, via the CVP, it is hard to obtain the point $\mathbf{As}$ knowing only $\mathbf{t}$ and $\mathbf{A}$[1].

Note that the LWE problem operates on the algebraic structure of integers. From the implementation point of view, this is somewhat undesirable, considering a multiplication between two vectors (dot product) does not yield another vector, but an integer. For a $n$-dimensional vector, each element in the output vector requires $n$ multiplication operations. Consequently, an $O(n^2)$ time complexity is associated with LWE-based schemes. This issue can be solved if instead of using vectors of integers, polynomials are used instead, where the result of a multiplication between two polynomials yields another polynomial, not an integer.

[1] This is called the "search" variant of the LWE problem. Another variant, called the "distinguish" variant is defined, where the $\mathbf{t}$ vector should be distinguished from a random distribution, outputting a "yes" or "no" answer, which is desirable for formal security proofs. For the sake of simplicity of the exposition, only the "search" variant will be considered.

The Module Learning With Errors (MLWE) problem is a variant of the standard LWE problem in which polynomials are used instead of integer vectors. In MLWE the **A** matrix is composed of $k \times k$ members of the $\mathbb{Z}_q[X]/(X^n + 1)$ polynomial ring i.e. the polynomials of degree less than $n$ with coefficients $c_i \in (0, 1, ..., q - 1)$, while the **t** and **s** vectors are composed of $k$ members of $\mathbb{Z}_q[X]/(X^n - 1)$. Note that this implies in an increase in memory usage when compared to standard LWE for $k > 1$. However, the gains in time complexity, directly translate to faster execution in practice and are found to offset the increase in memory required.

Crystals-Kyber [Avanzi et al., 2022] is a Key Encapsulation Mechanism (KEM), a method for establishing secret keys for symmetric cryptography. Based on the MLWE problem, it shows resistance against classical and quantum attacks. Kyber has a fixed plaintext size of 256 bits, allowing for securely establishing a key for block ciphers as strong as AES-256, which is the highest NIST security level for the AES block cipher. Kyber has 3 defined security levels, allowing for a faster and more compact implementation if the highest security level is not required for a specific scenario, i.e. using AES-128 (with Kyber-512) or AES-192 (with Kyber-768).

Table 5.1 shows parameters for each Kyber security strength. The $n$, $k$ and $q$ parameters set the polynomial ring of the form $\mathbb{Z}_q[X]/(X^n + 1)$ for the MLWE problem as $\mathbb{Z}_{3329}[X]/(X^{256} + 1)$ and the size for matrices and vectors of polynomials in the same ring. $\eta_1$, $\eta_2$, $d_u$ and $d_v$ specify the error distribution sampling and ciphertext compression factor, balancing security, ciphertext size and decryption failure probability, denoted by $\delta$.

Table 5.1 – Kyber round-3 parameters [Avanzi et al., 2022]

| Kyber version | $n$ | $k$ | $q$ | $\eta_1$ | $\eta_2$ | $(d_u, d_v)$ | $\delta$ |
|---|---|---|---|---|---|---|---|
| Kyber-512 | 256 | 2 | 3329 | 3 | 2 | $(10, 4)$ | $2^{-139}$ |
| Kyber-768 | 256 | 3 | 3329 | 2 | 2 | $(10, 4)$ | $2^{-164}$ |
| Kyber-1024 | 256 | 4 | 3329 | 2 | 2 | $(11, 5)$ | $2^{-174}$ |

Like any KEM as presented in Section 5.1, Kyber can be separated into 3 steps: `KeyGen`, `Enc` and `Dec`, respectively the public and private key generation, encryption of plaintext and decryption of ciphertext. Key generation can be understood directly from the MLWE problem statement, with the vector **t** and matrix **A** used as the public key and the vector **s** used as the private key[2]. The coefficients of each polynomial in the **A** matrix are sampled at random from $\mathbb{Z}_q$, with equal probability for each possible coefficient. Coefficients in **s** and **e** are sampled from a Centered Binomial Distribution (CBD), parameterized by the $\eta_1$ parameter from Table 5.1. $CBD_\eta$ returns a random element in the $-\eta, ..., 0, ..., \eta$ range from $2\eta$ pseudo-random bits, as shown in Algorithm 5.1.

[2]The notation in this section refers to polynomial vectors and matrices sized by the $k$ parameter as bold letters while single-dimensional (insensitive to $k$) polynomials are denoted by non-bold letters

**Algorithm 5.1** *$CBD_\eta$* sampling

**Require:** *B is an array of $2\eta$ pseudo-random bits*
**Ensure:** *Returns a pseudo-random element $\in Z_q$ in {0, 1, ..., $\eta$, $q-\eta-1$, ..., $q-1$}*
$a \leftarrow \sum_{i=0}^{\eta-1} B_i$ ▷ Sum $\eta$ lower bits of B
$b \leftarrow \sum_{i=\eta}^{2\eta-1} B_i$ ▷ Sum $\eta$ upper bits of B
**return** $a - b \bmod q$

In the `Enc` step, a new value $\mathbf{r}$ is sampled from $\text{CBD}_{\eta_1}$, analogous to $\mathbf{s}$ in the `KeyGen` step. $\mathbf{r}$ is used to compute two values $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$ and $v = \mathbf{t}^T\mathbf{r} + e_2 + m'$ are generated from the public key ($\mathbf{A}$ and $\mathbf{t}$) and scaled plaintext ($m'$), where $\mathbf{e}_1$ and $e_2$ are sampled from $\text{CBD}_{\eta_2}$. Each bit in the plaintext is scaled as to be a coefficient in $\mathbb{Z}_q$: If a plaintext bit is 1, it is scaled up to $q/2$, else, it remains as $0 \in \mathbb{Z}_q$.

The ciphertext ($\mathbf{u}$, $v$) is compressed according to the $(d_u, d_v)$ parameters in Table 5.1, reducing ciphertext size and introducing additional error due to lossy compression. The compression algorithm is shown in Algorithm 5.2, where a coefficient of a polynomial, a 12 bit integer, is compressed to a $d$ bit integer, or more formally, a mapping from $\mathbb{Z}_q$ to $\mathbb{Z}_{2^d}$.

**Algorithm 5.2** *$Compress_d$* Kyber coefficient compression

**Require:** *A is a polynomial coefficient in $\mathbb{Z}_q$*
**Ensure:** *Returns an element $\in \mathbb{Z}_{2^d}$*
**return** $= \lceil (2^d/q) \cdot A \rfloor \bmod 2^d$ ▷ $\lceil a \rfloor$ represents rounding $a$ to the nearest integer

$\mathbf{u}$ and $v$ are used in the `Dec` step to obtain $m'' = v - \mathbf{s}^T\mathbf{u}$. At this point in the decryption step, $m''$ contains the scaled plaintext $m'$ plus several small errors terms and the error due to compression, when all summed up are less than $q/2$. The actual plaintext $m$ is recovered from $m''$ by performing a reverse scaling process, where a coefficient $\in \mathbb{Z}_q$ is mapped to 1 if it is greater than $q/2$, and mapped to 0 if it is less than $q/2$.

Note that a significant portion of the intermediary operations in Kyber are polynomial multiplications. It is worth noting that the time complexity of the trivial algorithm for multiplying two $n$-degree polynomials is $O(n^2)$. However, by using the Number Theoretic Transform (NTT), it is possible to reduce the time complexity of polynomial multiplication to $O(n\ log_n)$. Therefore, using NTT to accelerate multiplications is a strong argument for implementing MLWE-based schemes over LWE-based schemes.

Polynomial multiplication ($\mathbf{c} = \mathbf{ab} \mid \mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{Z}_q[X]/(X^n + 1)$) via the NTT works by performing a forward transform on the two polynomials of interest ($\hat{\mathbf{a}} = NTT(\mathbf{a})$, $\hat{\mathbf{b}} = NTT(\mathbf{b})$), pairwise multiplying the coefficients of the two transformed polynomials ($\hat{\mathbf{c}}_i = \hat{\mathbf{a}}_i\hat{\mathbf{b}}_i\ mod\ q$), then taking the inverse transform (INTT) of the pairwise multiplication ($\mathbf{c} = INTT(\hat{\mathbf{c}})$). Concisely, $\mathbf{c} = \mathbf{ab} = INTT(NTT(\mathbf{a}) \cdot NTT(\mathbf{b}))$, where "$\cdot$" denotes pairwise multiplication mod $q$.

A simplified version of the NTT with $n = 8$ can be visualized in Figure 5.3. Note that there are $log_n$ layers, in each $n$ operations are performed, giving the previously mentioned $O(n\ log_n)$ time complexity. The operation in question for the forward transform is

called the Cooley-Tukey (CT) butterfly, which takes in two coefficients and a pre-computed constant called the "twiddle factor", determined from the index in each layer of the two input operands. The inverse transform follows the same structure, but performing the inverse base operation, called the Gentleman-Sande (GS) butterfly. The CT and GS butterflies are depicted in Figure 5.4.

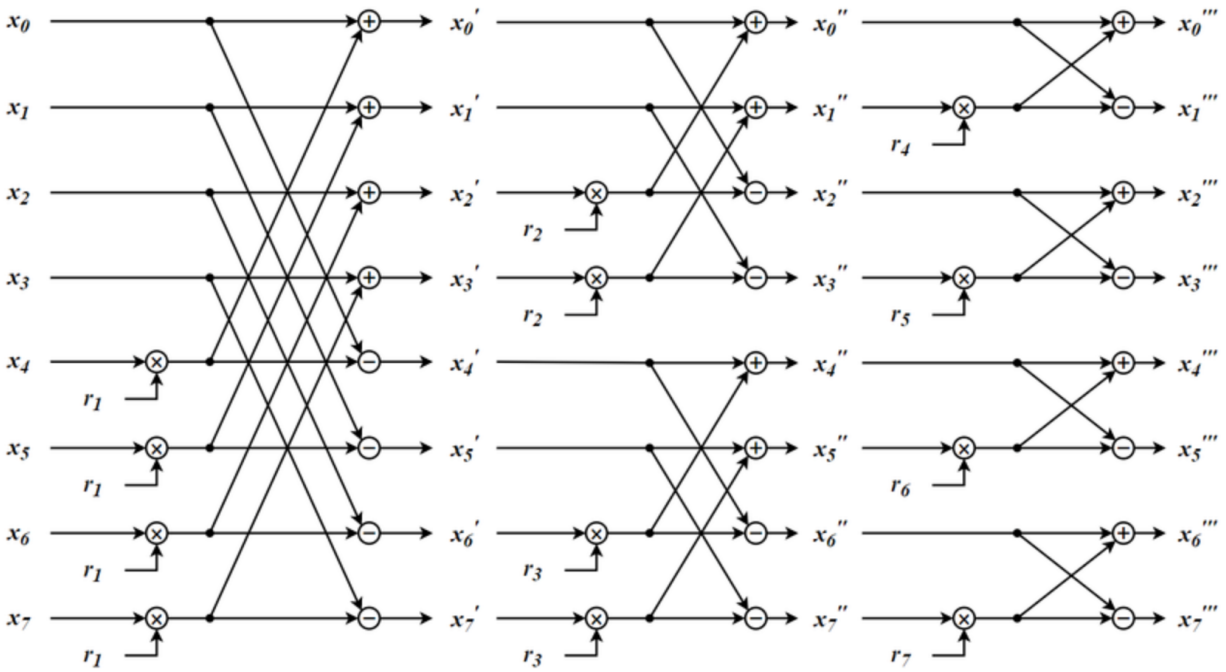


Figure 5.3 – NTT example for $n = 8$ [Di Matteo et al., 2023].

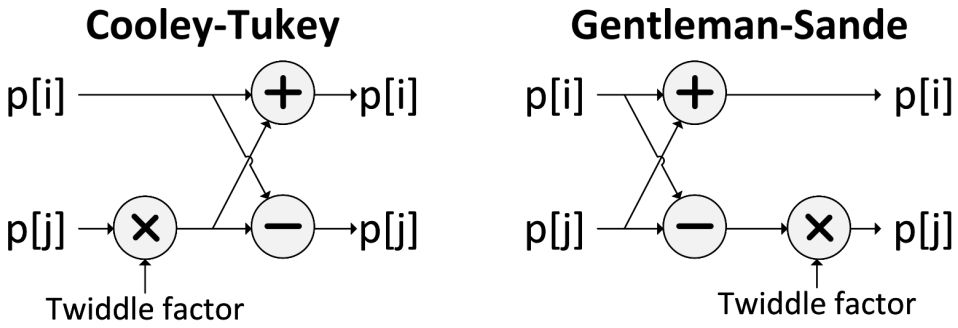


Figure 5.4 – Cooley-Tukey (CT) & Gentleman-Sande (GS) butterflies [Nannipieri et al., 2021].

The reader may notice a similarity of the process depicted in Figure 5.3 to a standard Fast Fourier Transform (FFT). This is not a coincidence. The NTT can be understood simply as an FFT that considers arithmetic in a $\mathbb{Z}_q$ ring instead of the complex numbers, as in the usual FFT. For a deeper conceptual exposition on the NTT, as well as a comprehensive review of implementation techniques, the reader should refer to Satriawan et al. [2023].

This constitutes the base Kyber algorithm, in the Kyber specification called Kyber.CPAPKE, constructed as to resist a Chosen Plaintext Attack (CPA). From this, a Chosen Ciphertext Adaptive Attack (CCA2) -secure scheme called Kyber.CCAKEM is constructed

via the Fujisaki-Okamoto (FO) transform [Avanzi et al., 2022]. This provides a second parametrization of Kyber, in which the hashes for the FO transform and pseudo-random functions for random sampling are chosen as either the hash functions and XOFs from the SHA-3 standard or hashes from the SHA-2 standard and AES-256 in CTR mode as a PRNG, respectively. The version of Kyber in which the SHA-3 standard is used is referred to as "standard Kyber" or "Keccak-based Kyber", while "Kyber-90s" uses SHA-2 and AES. A third parametrization of Kyber called "Kyber-Ascon" is proposed in this work, using the Ascon family of algorithms for the same functionalities. Seeing as Ascon only provides primitives at the 128 bit security level, Kyber-Ascon is evaluated only for Kyber-512. The specific primitives used in each symmetric parametrization of Kyber are shown in Table 5.2:

Table 5.2 – Symmetric primitives for each Kyber parametrization.

| Kyber Parameterization / Primitive | **XOF** | **H** | **G** | **PRF** | **KDF** |
|---|---|---|---|---|---|
| **Kyber-Keccak** | SHAKE128 | SHA3-256 | SHA3-512 | SHAKE256 | SHAKE256 |
| **Kyber-90s** | AES-256 CTR | SHA-256 | SHA-512 | AES-256 CTR | SHA-256 |
| **Kyber-Ascon** | Ascon-XOF | Ascon-XOF | Ascon-XOF | Ascon-XOF | Ascon-XOF |

## 5.2 Related Work

[Abdulrahman et al., 2022] presents the state-of-the-art Kyber implementation in the context of microcontroller processors, namely the ARM Cortex-M4. Several techniques from previous works are employed in this implementation as to provide the best known performance for a microcontroller-class processor. One of the most notable features of this implementation is the use of the CT butterfly for computing the INTT, instead of using the GS butterfly as commonly done in other works. Other optimizations such as the fast Barrett reduction, taking 6 clock cycles to be computed, are of less interest, seeing as in this work modular reductions will be implemented natively in hardware.

[Greconici, 2020] presents an implementation optimized explicitly for the RISC-V architecture. The paper evaluates the implementation empirically on the VexRiscV implementation, considering only the base RV32IM instruction set, without hardware acceleration using scalar cryptography extensions or any other ISA extensions being explored. The study optimizes the NTT for the first and second round submissions for the NIST competition, employing a layer merging strategy minimizing memory operations for intermediary values while computing the NTT and INTT. After optimizing the NTT, the paper reports that up to 70% of the clock cycles in the entire algorithm are spent in computing the Keccak permutation, suggesting that improving only the NTT is insufficient.

[Huang et al., 2022] presents the use of the recently proposed Plantard reduction [Plantard, 2021] in the context of lattice-based cryptography. Targeting the Cortex-M4 processor, a modified version of the Plantard reduction allowing for signed output is proposed

and implemented. Performance improvements of 25.02% and 18.56% are reported for the Kyber NTT and INTT, respectively. This work makes extensive use of SIMD instructions in the Cortex-M4, not available in low-complexity processors such as Ibex. Furthermore, Plantard arithmetic required that the pre-computed twiddle factors in the NTT to be stored as 32-bit integers instead of the usual 16-bit precomputed values in the reference Kyber implementation. In a further development of the work presented above, [Huang et al., 2023] presents the use of Plantard reductions in Kyber now in the low-complexity context of Cortex-M3 and RISC-V SiFive E31 processors, presenting both speed- and stack- optimized implementations. Unfortunately, at the time this document was written, their implementation was not made publicly available such as to be compared with the implementation from this work.

[Albrecht et al., 2018] explore the use of loosely coupled SHA-2 and AES accelerators in a smart-card SoC (SLE 78) to minimize some of the cycles spent in Keccak permutations, using the Kyber-90s version of Kyber. Conversely, [Park et al., 2022] explores the use of loosely coupled RSA/ECC accelerators to speed up polynomial arithmetic. This is done through the Kronecker substitution technique, used to compute polynomial multiplications as regular integer multiplications, which can be accelerated on existing RSA/ECC hardware.

[Banerjee et al., 2019] presents a generic accelerator for MLWE algorithms, implementing operations such as modular multiplication for a range of prime moduli and various random samplers from a SHA-3 PRNG. Their proposal can be visualized in Figure 5.5. This accelerator provides integration between several intermediary operations, yielding great performance improvements, at large area usage. Due to its significant costs in die area (106K Equivalent Gates), it is not an appropriate solution for the embedded context in question.

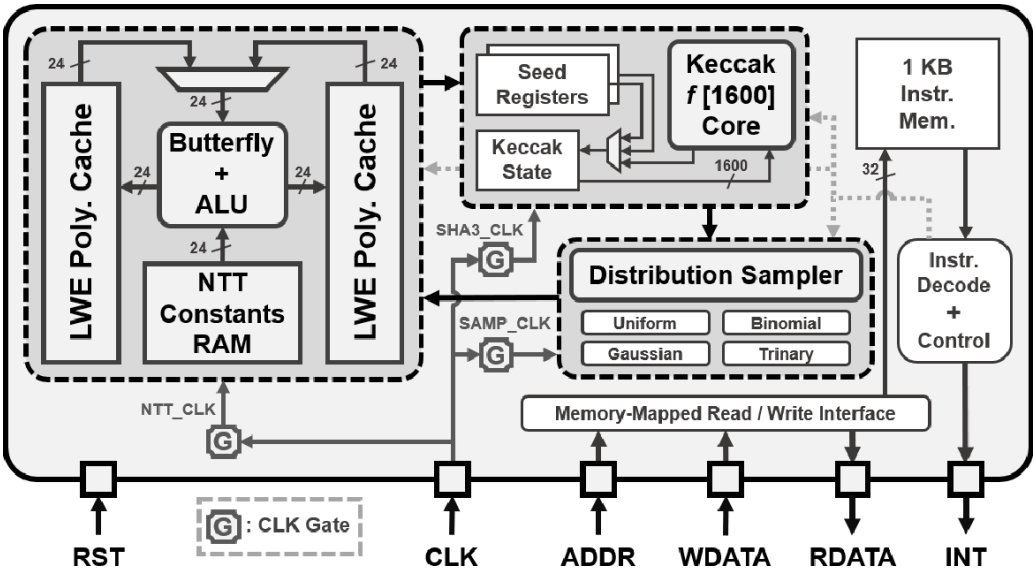


Figure 5.5 – MLWE accelerator from [Banerjee et al., 2019].

[Xin et al., 2020] propose a RISC-V vector co-processor for lattice-based post-quantum cryptography. Figure 5.6 presents the co-processor architecture. The co-processor implements the usual operations needed in lattice-based algorithms, such as random sam-

pling, which is integrated into a Keccak-1600 core, parallel butterflies for the NTT/INTT, as well as arithmetic operations with integrated modular reductions. This being a vector implementation, parallelism is explored extensively, not only in arithmetic operation execution itself but in buffering relevant inputs, such as the output of the Keccak PRNG in a FIFO and twiddle factors for the NTT in a local RAM. This implementation offers a significant performance increase when compared to a scalar software implementation, but is very expensive in terms of area, making it not suitable for the low-resource context being explored.

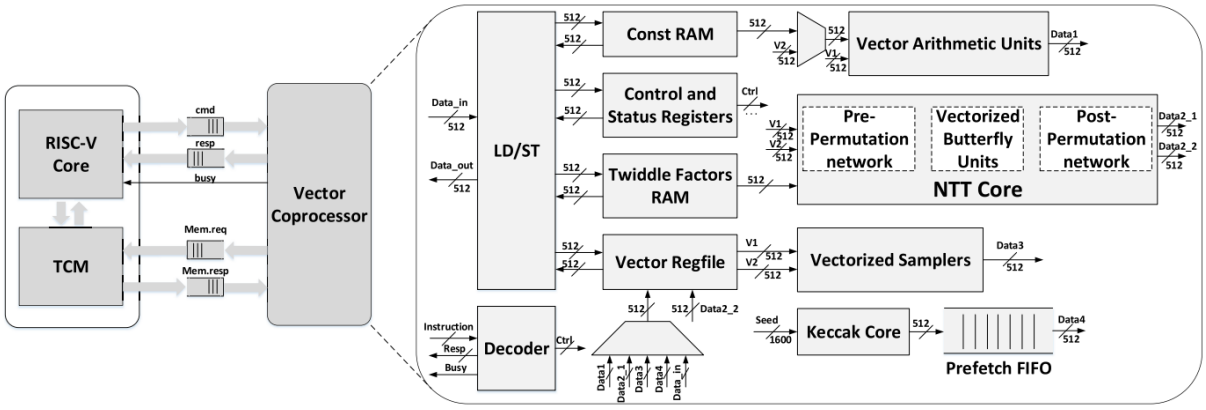


Figure 5.6 – Vector co-processor from [Xin et al., 2020].

[Nannipieri et al., 2021] shows a superscalar 6-stage 64-bit (CVA6) RISC-V processor extended with custom instructions for the Kyber and Dilhitium algorithms. These instructions are implemented based on a separate functional block, named PQ-ALU, illustrated in Figure 5.7. The PQ-ALU implements the usual modular arithmetic operations, as well as a one-cycle CT butterfly. This approach is not usable in a low-resource context due to a lack of resource sharing, both within the PQ-ALU, and among existing integer arithmetic elements in the base processor. We can observe 5 16-bit multipliers in a combinational path within the PQ-ALU, which, surprisingly, the authors report not to be the critical path of the extended CVA6 processor. This approach is reasonable considering the context of the high-complexity CVA6 64-bit superscalar processor, but not for the low-complexity Ibex core.

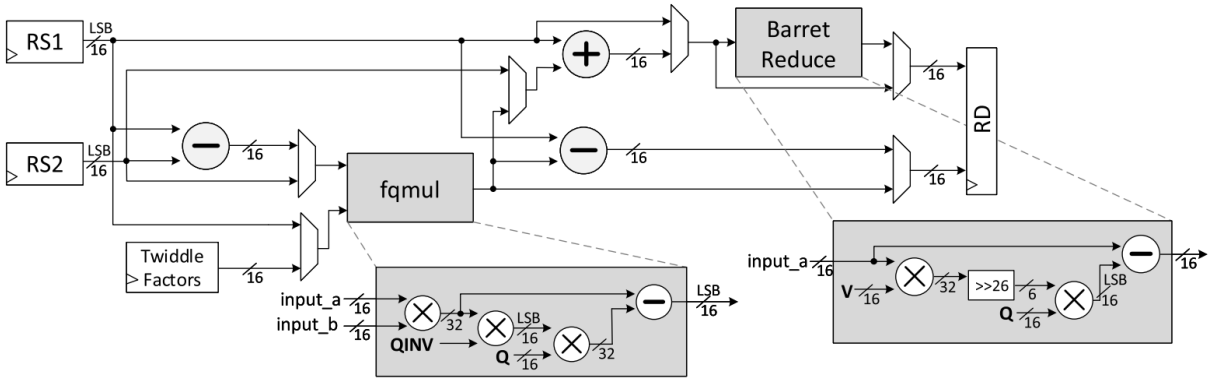


Figure 5.7 – PQ-ALU from [Nannipieri et al., 2021].

[Lee et al., 2022] presents a co-processor based approach, with the usual operations needed for Kyber and other MLWE-based algorithms, such as Keccak, sampling, and polynomial multiplication, as well as the standardized RISC-V scalar cryptography extensions, accelerating AES and SHA-2 as well. Figure 5.8 presents the proposed architecture. No quantitative results are presented, but we can conclude that this approach is not optimal, considering that the arithmetic modules are completely separate from the processor integer ALU, and intermediary results are committed to a separate register file in the co-processor. Again, resource sharing is not explored to the extent necessary for IoT applications.

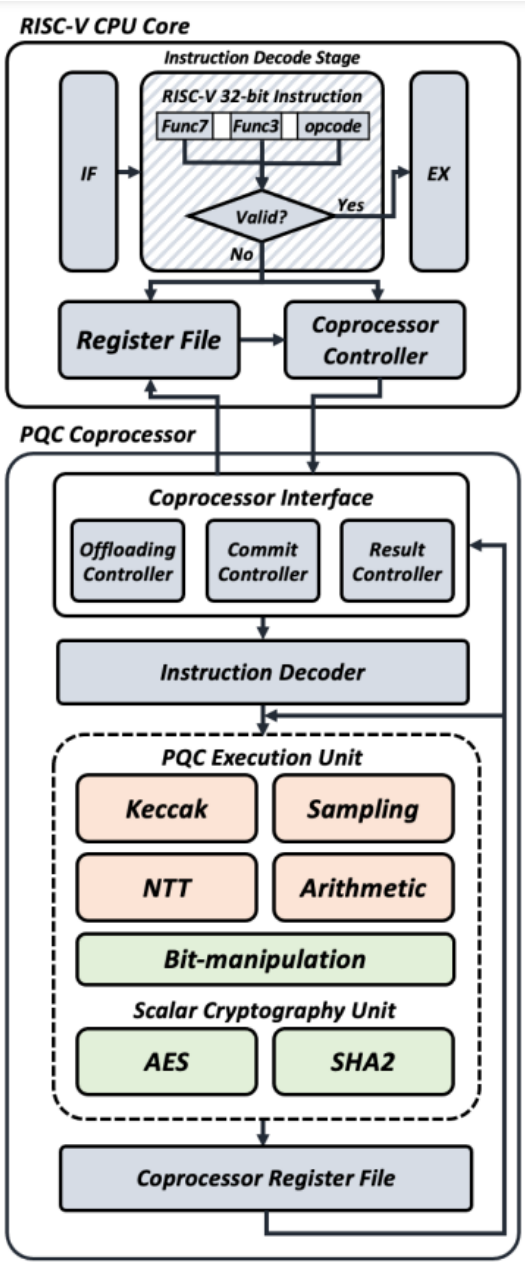


Figure 5.8 – Scalar co-processor from Lee et al. [2022].

[Karabulut and Aysu, 2020] presents a different approach to optimizing the NTT. Instead of implementing new instructions as in previous works, the authors propose several modifications to the base processor control signals, such that memory dependence prediction and out-of-order execution are accomplished when purely software butterflies are detected during code execution. Again, this proposal is made in the context of a complex processor and does not apply to Ibex.

In [Alkim et al., 2020], hardware-software co-design is carefully considered in the context of a 32-bit 5-stage RISC-V processor. New instructions for optimizing the NTT computation are implemented such that twiddle factors can be generated locally instead of being fetched from a pre-computed table stored in memory, as usually done in software implementations, reducing code size and the amount of memory accesses, which leads to energy consumption gains. However, resource sharing is not explored. In the base processor (VexRiscv) used in this proposal, regular integer multiplication computations are distributed between pipeline stages of the processor, and the same structure is followed for modular multiplication and reductions, shown in Figure 5.9 and Figure 5.10. Furthermore, the Authors implement the modular multiplication and reduction completely independently of the integer multiplication units in the base processor, such that the processor can be instanced without the integer multiplier while the optimizations proposed by the Authors still apply. Again, this is not an optimal use of resource sharing between regular integer arithmetic elements and modular arithmetic elements.

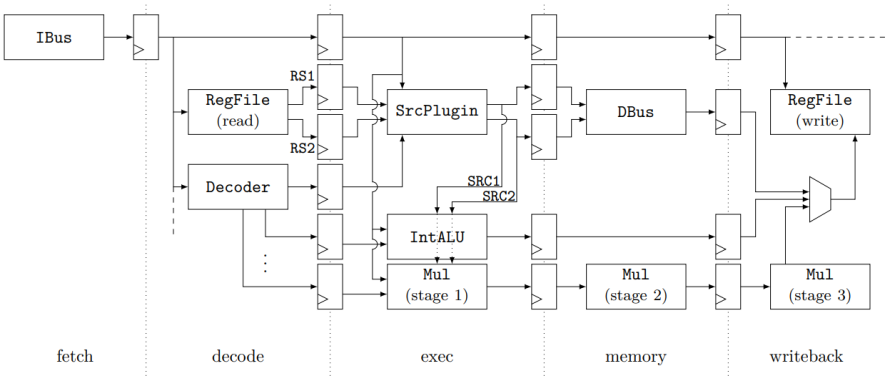


Figure 5.9 – Base processor (VexRiscV) from [Alkim et al., 2020].

[Miteloudi et al., 2023] presents a custom extension for polynomial arithmetic, supporting the Kyber and Dilithium algorithms. The supported operations are modular addition, subtraction, and multiplication, as well as CT and GS butterflies, implemented in a separate ALU, decoupled from the regular integer arithmetic elements in the RI5CY core. All operations do not incur pipeline stalls. This implementation considers the hardware acceleration of only modular arithmetic, and not other important elements in Kyber, as demonstrated in Section 5.3. For the single-cycle butterfly computations, it also considers a register file with two write ports, which is highly unusual and not expected to be supported in most low-complexity RISC-V implementations.

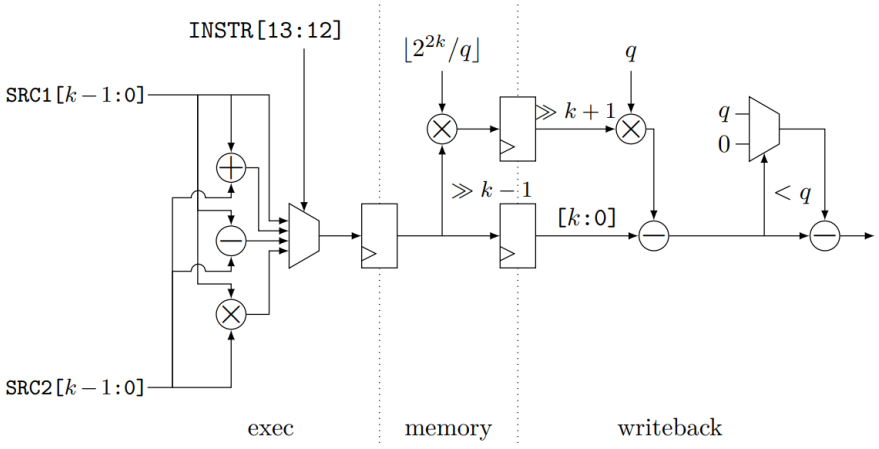


Figure 5.10 – NTT extension added to VexRiscV in [Alkim et al., 2020].

Summarizing, we can see that none of the works reviewed sufficiently address the issue of Post-Quantum Cryptography and the Kyber algorithm in resource-constrained environments considering Instruction Set Extensions (ISEs). The solutions proposed are either of naturally high complexity or mostly embedded in complex processors. An implementation that explores extensive resource sharing with existing integer arithmetic elements in a naturally low-complexity processor is still missing in the literature, providing a significant performance increase while decreasing energy consumption at reasonable die area costs.

## 5.3 Implementation

When proposing new RISC-V extensions, several principles should be followed as to design new scalar instructions that follow the overall design philosophy of the RISC-V architecture. Such principles have been previously stated concisely in [Marshall et al., 2020] in the context of the original proposals for an AES RISC-V extension, which later resulted in the standardization of the *Zkne* extension. These principles are re-stated below:

- The ISE must align with the wider RISC-V design principles. This means it should favour simple building-block operations, and use instruction encodings with at most 2 source registers and 1 destination register. This avoids the cost of a general-purpose register file with more than 2 read ports or 1 write port.
- The ISE must use the RISC-V general-purpose scalar register file to store operands and results, rather than any vector register file.
- The ISE must not introduce special-purpose architectural state, nor rely on special-purpose micro-architectural state (e.g., caches or scratch-pad memory).

- The ISE must enable data-oblivious execution (in [Marshall et al., 2020], of AES), preventing timing attacks based on execution latency (e.g., stemming from accesses to a pre-computed S-box).
- The ISE must be efficient, in terms of improvement in execution latency per area required: this balances the value in both metrics vs. an exclusive preference for one or the other. Efficiency wrt. auxiliary metrics, e.g., memory footprint or instruction encoding points, is an advantage but not a requirement.

In this work the *XKyber* extension is proposed, containing 6 new instructions designed to optimize Kyber implementations in low cost RISC-V 32 bit processors, following the general principles outlined above. Four of these instructions rely on the fact that Kyber polynomial coefficients are 12 bit integers, stored in memory as arrays of 16 bit variables, meaning that 2 coefficients can fit in the native 32 bit word size. Processing can be performed in parallel for 2 coefficients at a time using regular integer arithmetic elements in the base processor, without the need for costly vector co-processors or extensive additional logic. Single Instruction Multiple Data (SIMD) processing also reduces the amount of memory accesses, seeing as 2 coefficients are loaded/stored from/to memory in a single word-wide access.

The 6 new instructions are presented in Table 5.3. Their implementation and integration into Ibex (along with all other extensions discussed in this work, shown with dotted lines) is illustrated in Figure 5.11. Existing elements are shown in green, added elements shown in blue and control signals shown in orange.

Table 5.3 – XKyber instructions

| Instruction | Functionality |
|---|---|
| kybercbd2 RD, RS1 | $RD[11:0] \leftarrow CBD_2(RS1[3:0])$; $RD[27:16] \leftarrow CBD_2(RS1[7:4])$; (Algorithm 5.1) |
| kybercbd3 RD, RS1 | $RD[11:0] \leftarrow CBD_3(RS1[5:0])$; $RD[27:16] \leftarrow CBD_3(RS1[11:6])$; (Algorithm 5.1) |
| kyberadd RD, RS1, RS2 | $RD[11:0] \leftarrow RS1[11:0] + RS2[11:0] \ mod\ q$; $RD[27:16] \leftarrow RS1[27:16] + RS2[27:16] \ mod\ q$; |
| kybersub RD, RS1, RS2 | $RD[11:0] \leftarrow RS1[11:0] + RS2[11:0] \ mod\ q$; $RD[27:16] \leftarrow RS1[27:16] + RS2[27:16] \ mod\ q$; |
| kybermul RD, RS1, RS2 | $RD \leftarrow RS1[11:0] \times RS2[11:0] \ mod\ q$; |
| kybercompress RD, RS1, RS2 | $RD \leftarrow Compress_{RS2}(RS1[11:0])$; (Algorithm 5.2) |

The `kybercbd2` and `kybercbd3` instructions implement CBD sampling as previously shown in Algorithm 5.1, sampling 2 coefficients $\in \mathbb{Z}_q$ at a time, differing by the $CBD_\eta$ $\eta$ value. The coefficients are placed in the upper and lower 16 bits of the target register, facilitating storing both coefficients in memory with a single word-wide store instruction. For the $CBD_3$ case, 2 full adders are needed to compute the $A$ and $B$ variables in Algorithm 5.1, as well as a 3-bit adder and 3 inverters to compute the $A - B$ difference for each of the 2 $CBD_3$ computations. For the $CBD_2$ case, 2 half adders are enough to compute the $A$ and $B$ variables. Once $A - B$ is computed, a value in $\{-\eta, ..., \eta\}$ is obtained, which must be brought to the $\{0, ..., \eta, q-\eta-1, ..., q-1\}$ range, effectively computing $mod\ q$ in Algorithm 5.1. This is performed by conditionally adding $q$ if the computed difference is less than 0. The conditional addition logic is shared with the `kyberadd`, `kybersub` and `kybermul` instructions.

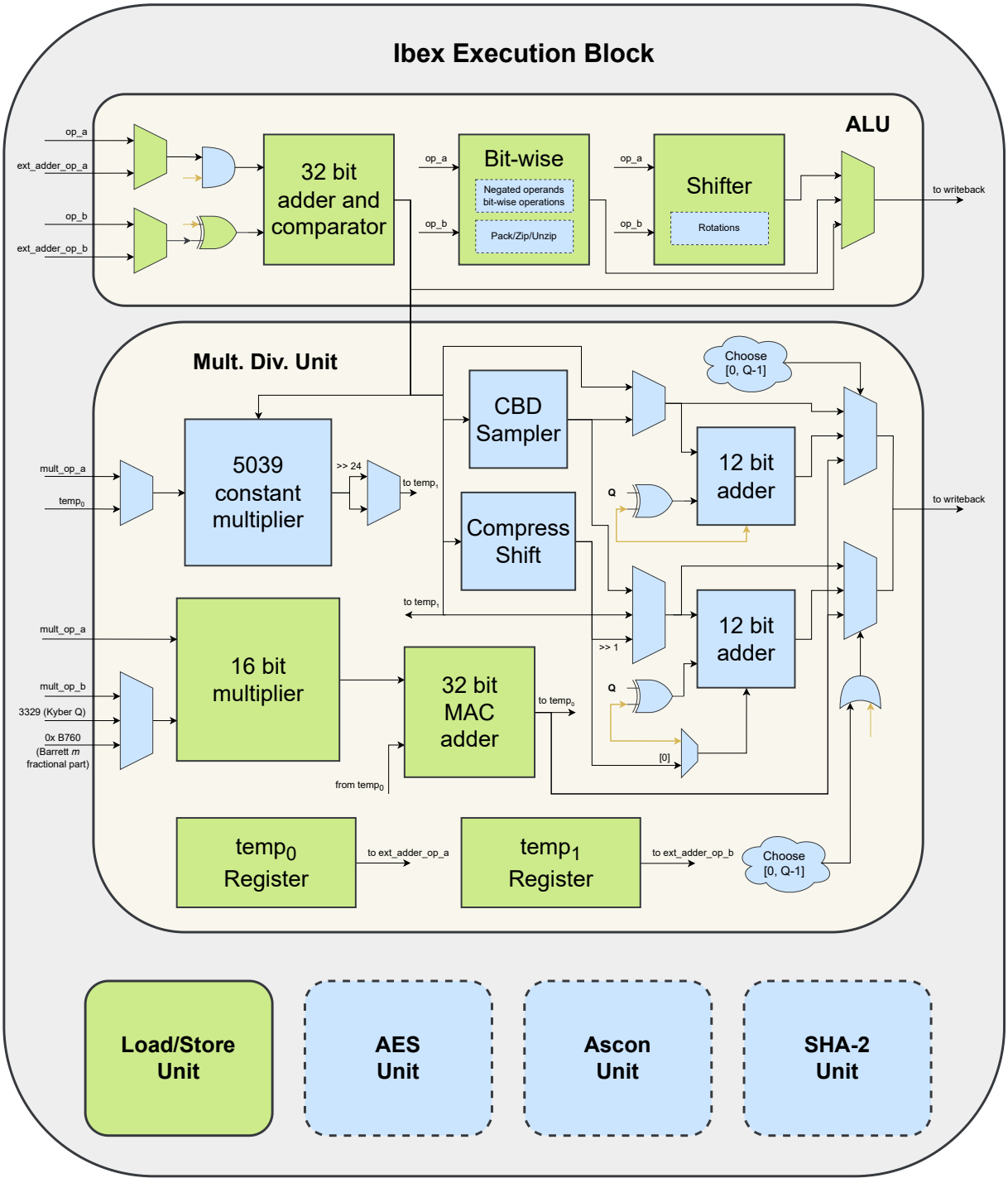


Figure 5.11 – Ibex execution block with extensions.

`kyberadd` and `kybersub` compute 2 coefficient additions or subtractions in $\mathbb{Z}_q$ in parallel. In this manner two coefficients can be fetched and stored back in memory in single word-wide instructions in addition to being processed simultaneously. The additions/subtractions themselves are performed in the existing 32 bit adder in the ALU, with trivial masking for stopping carry propagation from the coefficient in the lower bits to the coefficient in the higher bits. Like with `kybercbd2` and `kybercbd3`, *mod q* is performed via conditional addition or subtraction, subtracting $q$ if if the intermediary result is $> q$ in the `kyberadd` case or adding $q$ if the intermediary result is $< 0$ in the `kybersub` case.

`kybermul` performs a single (non-SIMD) multiplication in $\mathbb{Z}_q$. Note than the modular reduction cannot be performed as a simple conditional subtraction as previously done with `kyberadd` due to the fact that the intermediary multiplication is in the $\{0, ..., (q-1)^2\}$ range, where a conditional subtraction by $q$ could only bring it to the $\{0, ..., (q-1)^2 - q\}$ range. For efficiently computing the modular reduction of the intermediary product, the Barrett reduction [Barrett, 2020] is used. The generic algorithm for the Barrett reduction is presented in Algorithm 5.3. In this work the $k$ variable is set as 24.

**Algorithm 5.3** Modular multiplication in $\mathbb{Z}_q$ with Barrett reduction [Barrett, 2020]

**Require:** $a, b \in \mathbb{Z}_q, k \approx 2 \cdot \lceil log_2\ q \rceil, m = \lfloor 2^k/q \rfloor = 5039$
**Ensure:** $z \in \mathbb{Z}_q$
$temp_0 \leftarrow a \cdot b$
$temp_1 \leftarrow (temp_0 \cdot m) \gg k$
$temp_1 \leftarrow (temp_1 \cdot q)$
$ret \leftarrow ((temp_0 - temp1) \geq q)\ ?\ (temp_0 - temp_1) - q\ :\ (temp_0 - temp_1)$
**return** *ret*

The modular multiplication with Barrett reduction is implemented in 4 clock cycles, making extensive use of resource sharing with existing elements in the Ibex processor's 3-cycle multiplier and 2 internal registers $temp_0$ and $temp_1$ for intermediary results. The computations performed at each clock cycle follow the structure presented in Algorithm 5.3, with one line of Algorithm 5.3 being computed per clock cycle.

First, the 12 bit by 12 bit multiplication in line 1 is computed in the 16 bit multiplier, with its result being stored in $temp_0$, exactly like the first cycle of 32 bit by 32 bit multiplications from the *M* extension. In the second cycle, the product of $temp_0$ and the $m$ constant is computed via a constant multiplier. The multiplier implementation stems from the fact that the 5039 constant can be factored as sums and subtractions as powers of two in the form $5039 = 2^{12} + 2^{10} - 2^6 - 2^4 - 2^0$. Seeing as multiplications by power of two can implemented as trivial left-shifts, a multiplication by 5039 can then be written as $x \cdot 5039 = (x \ll 12) + (x \ll 10) - (x \ll 6) - (x \ll 4) - x$. In this work, the $(x \ll 12) + (x \ll 10)$ share is computed in the existing Ibex ALU, requiring a single additional 32-multiplexer such that one of the operands is $(x \ll 2)$, while the other operand is simply $x$. This result, left-shifted by 10, is added to $(x \ll 6) - (x \ll 4) - x$ as computed in 3 new adders instantiated in the multiplier. In this manner, the additional resources required for the 5039 constant multiplier are only 3 adders and a multiplexer. At the end of the second cycle, $temp_1$ is written the result of the constant 5039 multiplication shifted by $k = 24$.

In the third cycle of `kybermul` the multiplication of $temp_1$ and the $q$ constant is performed. This multiplication is computed in the 16 bit generic multiplier, requiring the addition of additional multiplexers to its existing mux logic, but not a second dedicated constant multiplier. Note that the $q$ constant mux is a logical mux, not a physical mux, since it needs to multiplex a constant value. This result is written to the $temp_1$ register, overwriting the previ-

ous value which is not needed anymore. The fourth and final cycle of `kybermul` computes the subtraction of $temp_1$ from $temp_0$ and fits its range into $\{0, ..., q-1\}$. $temp_0 - temp_1$ is computed in the ALU, while the range adjustment is computed using the same logic as previously described for `kyberadd`.

Finally, `kybercompress` performs the (lossy) compression of a 12 bit coefficient into a $d$ bit value. Like `kybermul`, it is also a non-SIMD instruction. First, note that the Barrett reduction constant $m = \lfloor 2^k/q \rfloor$, as shown in Algorithm 5.3, is similar to the $2^d/q$ multiplication in Algorithm 5.2. Also note that compression requires rounding to the nearest integer, while $m$ is truncated. In order to reuse the $m = 5039$ constant multiplier for computing coefficient compression, the $2^d/q$ multiplication is computed using both the constant multiplier and the generic 16 bit multiplier. In this manner, seeing as the $2^d/q$ multiplication can be rewritten as $2^k/q \gg (k-d)$, the constant 5039 multiplier computes multiplication for the integer part while the generic multiplier computes multiplication by the fractional part, storing the results in $temp_0$ and $temp_1$, respectively.

In the second and final clock cycle of `kybercompress`, both integer and fractional shares of the $2^d/q$ multiplication are added, and shifted by $24 - d - 1$. The LSB of this intermediary shift value determines if +1 will be added to the final result, computing the nearest integer rounding. The +1 addition to the combined integer and fractional shares shifted by $24 - d$ is computed in the range adjusting adder, using only the carry in bit with the $q$ constant masked by AND gates. Finally, the adjust adder result is masked by $d$ bits, effectively computing $mod\ 2^d$, yielding the final compression result.

## 5.4 Experimental Evaluation

In this section, a comparative evaluation of the Kyber-512, Kyber-768, and Kyber-1024 security strengths is presented, considering the parametrization of symmetric primitives Kyber-Keccak, Kyber-90s, and the novel Kyber-Ascon for each of the `KeyGen`, `Enc`, `Dec` steps. The evaluation is performed both with and without hardware acceleration for the symmetric primitives as previously discussed in Sections 3.4 and 4.4 and Kyber-specific operations via the *XKyber* extension. The evaluation comprises a performance comparison as well as energy consumption, memory footprint and die area cost analyses.

### 5.4.1 Performance Evaluation and Profiling

Kyber performance relies on symmetric primitives for its internal sampling and hashing operations. The choice of Kyber symmetric primitives for each parametrization is shown in Table 5.2, restated below for the reader's convenience. Table 5.5 shows a through-

put comparison between symmetric primitives choices for the *XOF* and *PRF* functions, used internally in Kyber for generating the $\boldsymbol{A}$ matrix and error terms, respectively. AES-256 in CTR mode shows the best throughput out of all primitives compared in Table 5.5, considering both software-only and hardware-accelerated implementations. Ascon-XOF shows better throughput than SHAKE256, but worse throughput than SHAKE128. The choice of the *G* and *H* hash functions and *KDF* key derivation function are also impactful. The same core operation throughput analysis for these functions, previously presented in Section 3.4.1, also applies, favoring SHA-256 and SHA-512 over SHA3-256 and SHA3-512, which again favors Kyber-90s over Kyber-Keccak. Ascon-XOF has the same characteristics as Ascon-Hash, which shows worse throughput than both SHA-256 and SHA-512, but better than SHA3-256 and SHA3-512.

Table 5.4 – Symmetric primitives for each Kyber parametrization.

| Kyber Parameterization / Primitive | **XOF** | **H** | **G** | **PRF** | **KDF** |
|---|---|---|---|---|---|
| **Kyber-Keccak** | SHAKE128 | SHA3-256 | SHA3-512 | SHAKE256 | SHAKE256 |
| **Kyber-90s** | AES-256 CTR | SHA-256 | SHA-512 | AES-256 CTR | SHA-256 |
| **Kyber-Ascon** | Ascon-XOF | Ascon-XOF | Ascon-XOF | Ascon-XOF | Ascon-XOF |

Table 5.5 – Core operation profiling for each Kyber XOF and PRF choice (smaller is better).

| **Core Operation Metric** | AES-256 CTR | | SHAKE128 | | SHAKE256 | | Ascon-XOF | |
|---|---|---|---|---|---|---|---|---|
| **Clock Cycles** | 1,490 | 311 | 30,197 | 22,355 | 30,197 | 22,355 | 1,558 | 738 |
| **Instructions Retired** | 1,234 | 240 | 22,715 | 13,713 | 22,715 | 13,713 | 1,501 | 671 |
| **Cycles per byte** | 128.500 | **26.062** | 179.744 | 133.065 | 222.037 | 164.375 | 194.750 | 92.256 |
| **Instructions per byte** | 107.625 | **20.375** | 135.208 | 81.625 | 167.022 | 100.831 | 187.625 | 83.875 |

Figure 5.12 shows the performance of each of the `KeyGen`, `Enc`, `Dec` steps in Kyber-512, Kyber-768 and Kyber-1024, considering the symmetric primitive choices of Kyber-Keccak, Kyber-90s and the novel Kyber-Ascon. The observed performances follow the analysis the core operation throughputs shown in Table 5.5, with Kyber-90s being the fastest Kyber symmetric parametrization. Notably, software only Kyber-90s is faster than hardware accelerated Kyber-Keccak and Kyber-Ascon for all 3 Kyber steps. Kyber-Keccak and Kyber-Ascon show very similar performance, with a slight advantage to Kyber-Ascon comparing software only implementations (< 1% difference) and a greater advantage to Kyber-Ascon comparing hardware accelerated implementations (17% difference). Kyber-Ascon shows the most significative performance benefits from hardware acceleration from ISEs (28% difference).

Figure 5.13 (note the scale difference for each plot) shows the performance of operations within Kyber that are accelerated by the *XKyber* extension in polynomial arithmetic, CBD and coefficient compression operations. Significant gains are are shown in each of the operations, with the most significant clock cycle gains being in polynomial pointwise multiplication, (67% gain), CBD3 (72% gain) and polynomial coefficient compression (Algorithm 5.2) $\text{Compress}_4$ (78% gain).

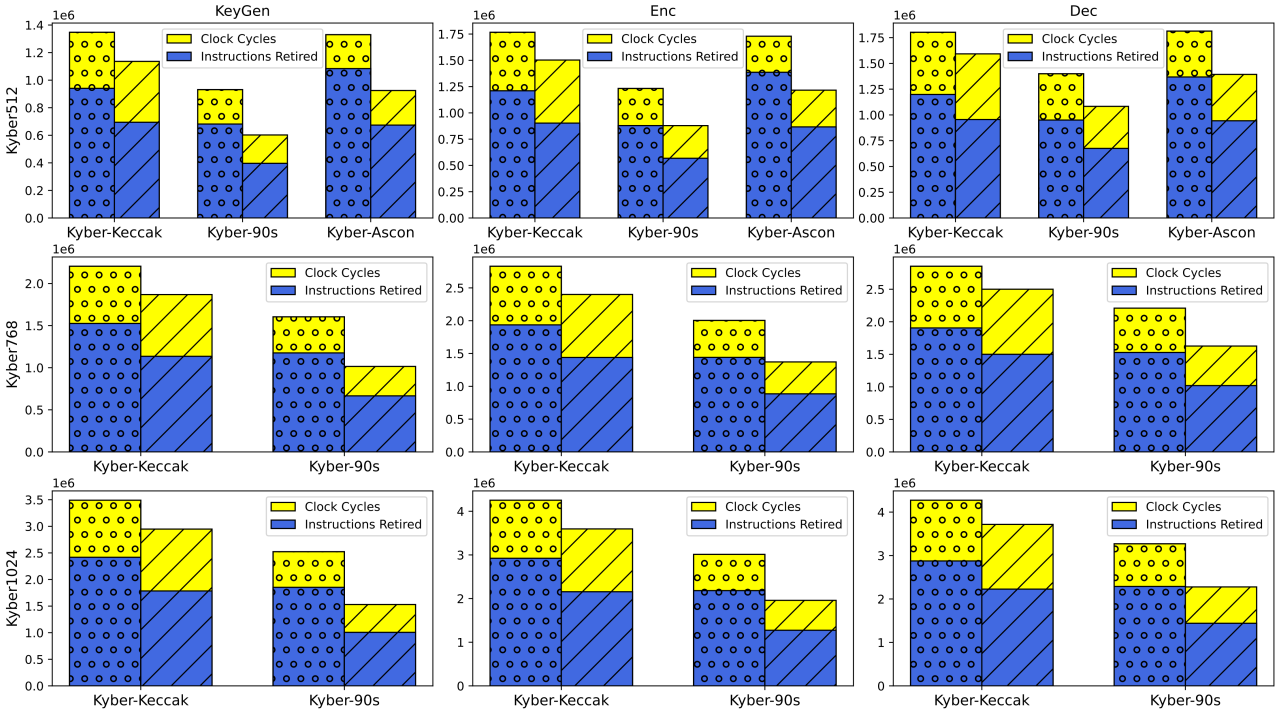


Figure 5.12 – Performance of hardware acceleration of Kyber symmetric primitives.

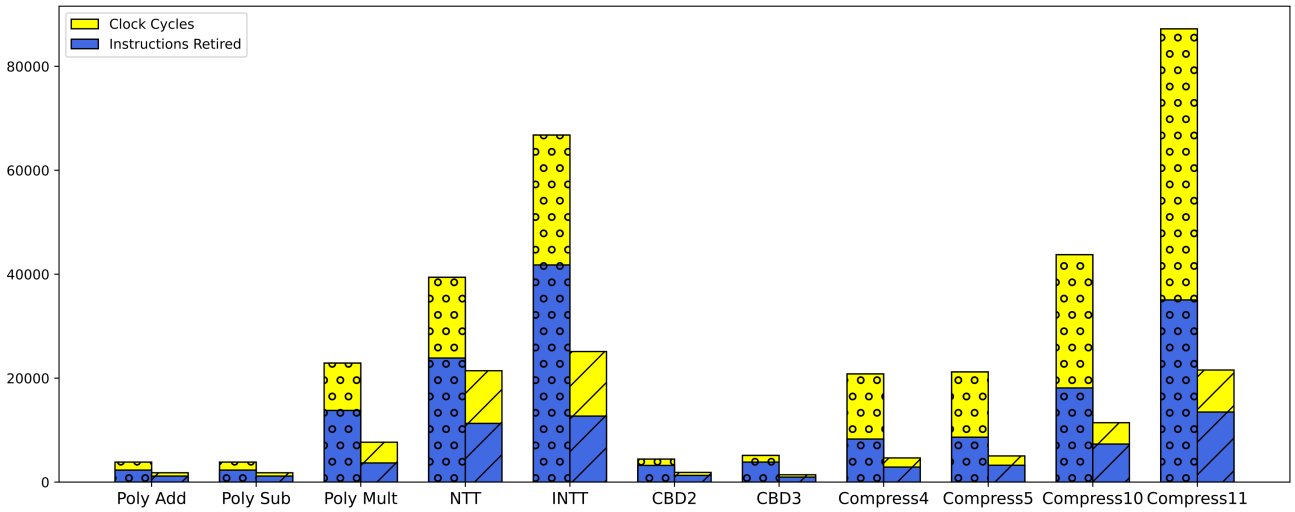


Figure 5.13 – Performance of XKyber extension in Kyber internal operations.

Figure 5.14 shows (note the scale difference for each plot) the performance of the 3 Kyber steps using the *XKyber* extension. A baseline software implementation of Kyber-90s is compared to an implementation with only the symmetric primitives accelerated via the *Zkne* and *Zknh* extensions and an implementation with both symmetric acceleration and Kyber-specific acceleration. The impact of accelerating Kyber operations via *XKyber* is roughly the same as accelerating the symmetric primitives via *Zkne* and *Zknh*. From Figure 5.12, Kyber-90s is the best case for hardware acceleration of symmetric primitives. If the Kyber variant in question were Kyber-Keccak or Kyber-Ascon, hardware acceleration via *XKyber* would outperform hardware acceleration of symmetric primitives via *Zbkb* or *XAscon*.

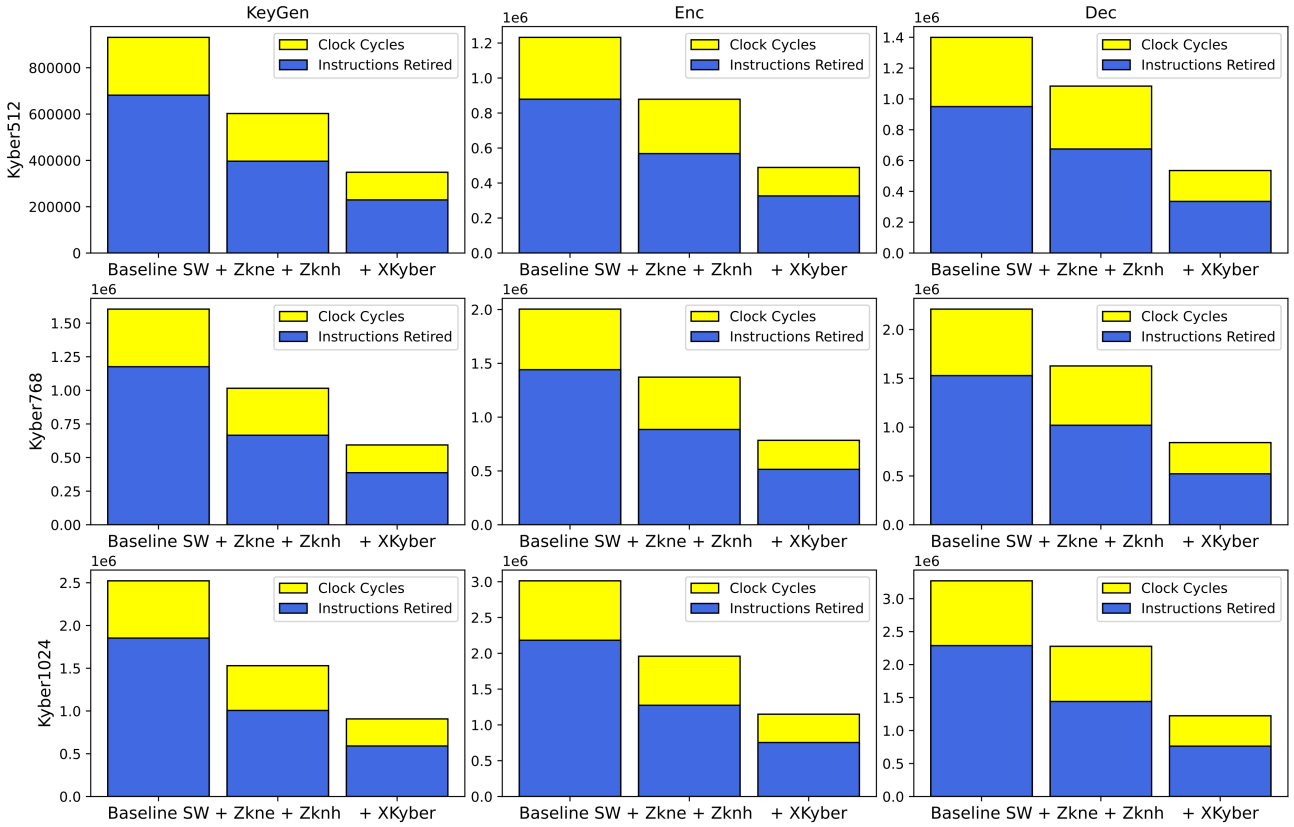


Figure 5.14 – Performance of XKyber extension in Kyber-90s.

### 5.4.2 Memory Footprint Evaluation

Kyber memory usage is not significantly influenced by the choice of symmetric primitives or the hardware acceleration via ISEs of those primitives. However, the internal Kyber operations have reduced code size as shown in Figure 5.15. The baseline software implementations are shown with bars hatched with circles, while hardware accelerated implementations using instructions from the *XKyber* extension are shown in bars hatched with diagonal stripes. In addition to reducing the code size of the operations themselves, auxiliary functions such as Barrett and Montgomery reductions and conditional subtractions are not needed anymore, seeing as operands with *XKyber* instructions are always $\in \mathbb{Z}_q^n$. A software-only implementation of Kyber-512 with the Kyber-Keccak symmetric parametrization shows a total code size of 5090 bytes (not including symmetric primitives), while an implementation with *XKyber* instructions shows a code size of 4314 bytes, a 15% improvement.

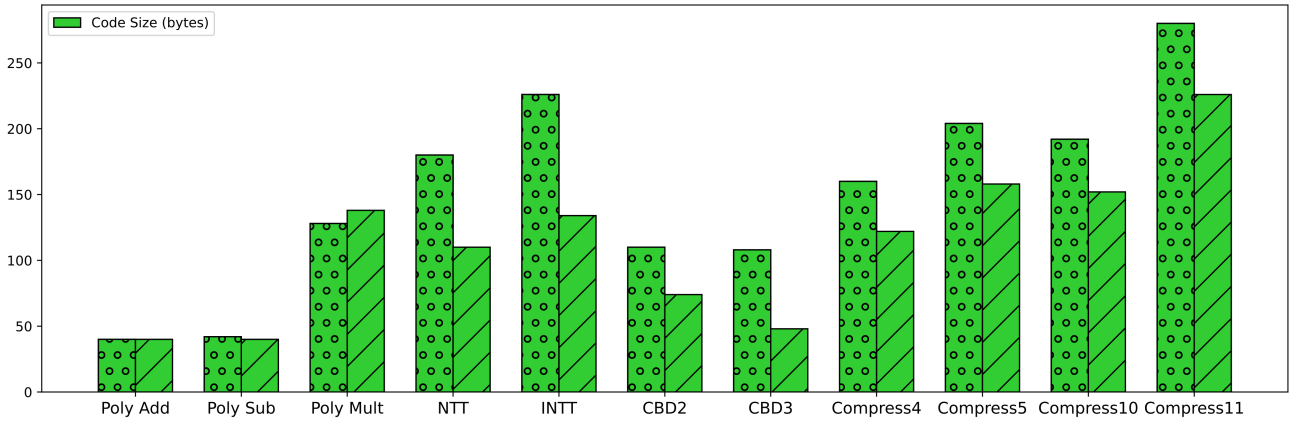


Figure 5.15 – Memory footprint of Kyber operations accelerated by XKyber.

### 5.4.3 Energy Consumption Evaluation

The energy consumption of each Kyber step in each Kyber symmetric parametrization is shown in Figure 5.16. Energy spent due to data and instructions memory operations are shown, respectively, in yellow and green. Energy spent due to data processing inside the Ibex processor is shown in blue. The baseline software-only implementations are shown with bars hatched with circles, while implementations with hardware-accelerated symmetric primitives are shown in bars hatched with diagonal stripes. Note the scale difference for each plot.

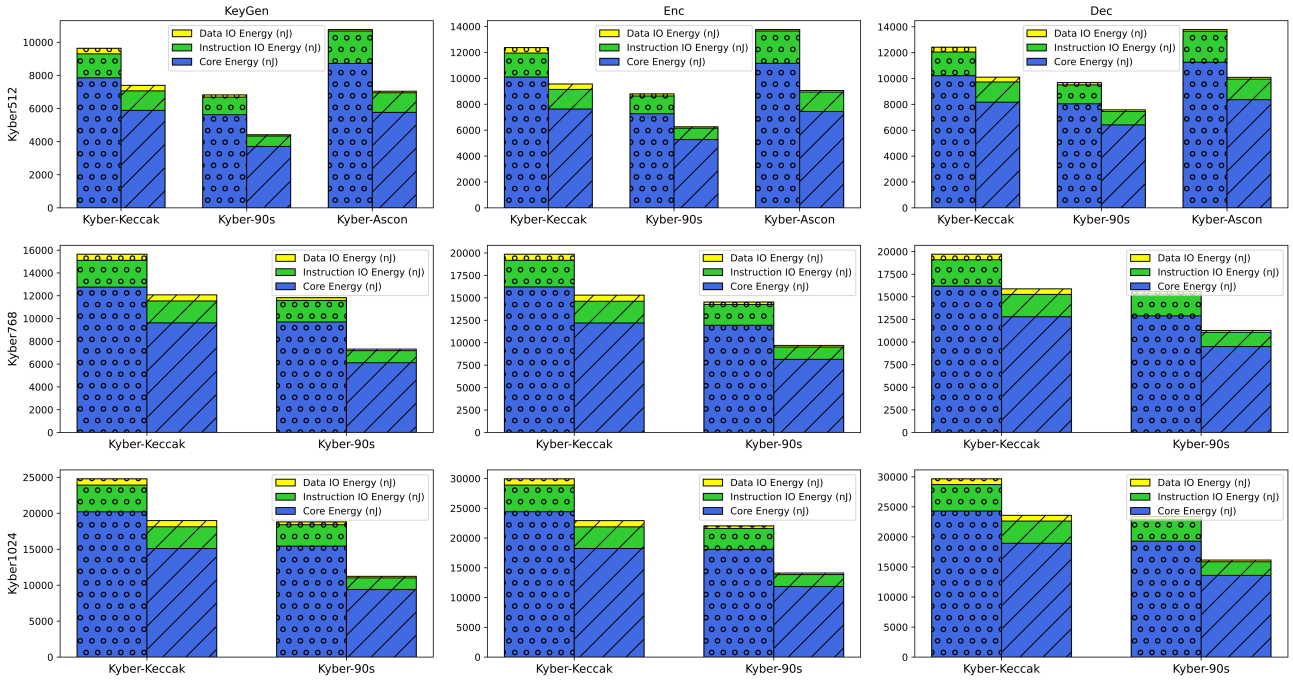


Figure 5.16 – Energy consumption of Kyber variants.

The core energy dominates the total energy consumption due to the internal processing of data rather than moving data to/from memory. Despite both having roughly the same performance, Kyber-Ascon shows higher energy consumption than Kyber-Keccak. This is explained by the fact that Kyber-Ascon executes more instructions during the same amount of clock cycles, increasing average power, and therefore, total energy. Following the performance trend shown in Figure 5.12, software-only Kyber-90s shows slightly better energy consumption than hardware-accelerated Kyber-Keccak, which further improves when considering hardware acceleration.

Figure 5.17 shows the energy consumption of each Kyber step and security strength, considering a baseline software-only implementation of Kyber-90s, an implementation with hardware accelerated symmetric primitives only, and an implementation with hardware-accelerated symmetric primitives and Kyber internal operations via *XKyber*. The total energy consumption follows the trend for performance, with energy consumption gains obtained from the *Zkne* and *Zknh* extensions being equivalent to gains obtained from the *XKyber* extension. Note the scale difference for each plot.

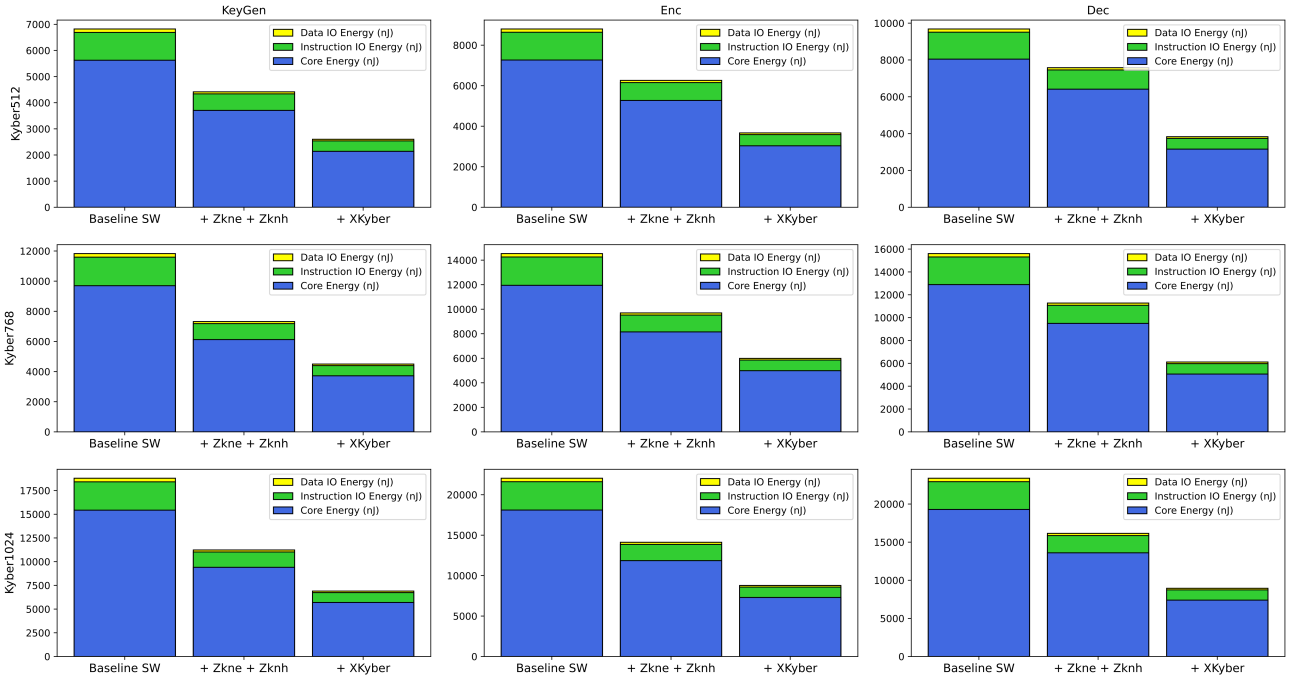


Figure 5.17 – Energy consumption of XKyber extension in Kyber-90s.

### 5.4.4 Die Area Costs Evaluation

Table 5.6 shows synthesis results for the Ibex core and each ISE being explored. The *Zkne* and *Zknh* combination showing a 11% increase and *Xascon* showing a 9% increase. *Zbkb* shows a smaller 1% cell area increase, but shows the greatest increase in cell count at 13%. *XKyber* shows a similar increase of 4 kGE as the *Zkne* and *Zknh* case.

| **Synthesis results** | Ibex baseline | Ibex + Zknh + Zkne | Ibex + Zbkb | Ibex + XAscon | Ibex + Zknh + Zkne + XKyber |
|---|---|---|---|---|---|
| Cell Area ($\mu m^2$) | 11,238 | 12,447 | 11,307 | 12,210 | 13,734 |
| Net Area ($\mu m^2$) | 6,992 | 7,278 | 5,142 | 8,181 | 7,568 |
| Total Area ($\mu m^2$) | 18,230 | 19,726 | 16,449 | 20,391 | 21,302 |
| Cell Count (# instances) | 10,289 | 11,687 | 11,769 | 11,010 | 13,131 |
| Equivalent NAND2 gates | 34,433 | 38,132 | 34,642 | 37,408 | 42,076 |
| Slack @ 500 MHz | 0 | 0 | 0 | 0 | 0 |

Table 5.6 – Area comparison of ISEs for Kyber hardware acceleration.

## 5.5 Conclusion and Final Remarks

Considering the analyses presented in Section 5.4, the choice of Kyber symmetric primitives significantly impacts the performance and energy consumption of Kyber in resource-constrained embedded systems. Compared to the standard Kyber-Keccak, Kyber-90s offers 28% better performance while consuming 27% less energy in a software-only implementation of Kyber-512. Kyber-Ascon shows similar performance to Kyber-Keccak, but may be an attractive option for severely memory-constrained systems at the 128-bit security level, considering the memory footprint of primitives as previously discussed in Sections 3.4.2 and 4.4.2.

Hardware acceleration of Kyber symmetric primitives also shows significant gains. In Kyber-512, 32% gains for both performance and energy consumption for the Kyber-90s parametrization are observed. Additionally, adding the *XKyber* extension for accelerating internal Kyber operations can enhance performance and energy consumption even further, providing gains of 46% and 44%, respectively, considering Kyber-512 with hardware acceleration via *Zkne* and *Zknh*. As an extra benefit, *XKyber* reduces Kyber code size by 15%. Hardware acceleration comes at an area cost of 10% of a baseline Ibex core, both for symmetric primitives and *XKyber*.

# 6. CONCLUSIONS AND FUTURE WORK

With the increasing deployment of low-complexity embedded systems such as IoT sensor nodes, battery-powered and energy-harvesting devices, secure communication is a core requirement. Such devices often do not have the necessary resources for locally computing data processing demands, delegating intensive work to the cloud or network edge. Due to this emerging computing paradigm, a significant amount of data must be securely transmitted to and from several network endpoints. This is not an easy task to accomplish, seeing as the cryptographic algorithms that enable the confidentiality and integrity of the data being transmitted are associated with non-trivial overheads in application performance and memory usage, which are critical concerns in the context of resource-constrained devices.

In this work, the use of Instruction Set Extensions (ISEs) is demonstrated to lead to significant improvements in implementations of hash functions, authenticated encryption, and post-quantum key encapsulation in RISC-V-based resource-constrained embedded systems. In summary, the use of ISEs in hash functions provides gains of 32%, 38% and 16% in performance, energy consumption, and code size, respectively. Gains in authenticated encryption are of 58%, 61% and 35% in performance, energy consumption, and code size, respectively. Area costs are of at most 10% of the baseline Ibex processor with no ISEs, corresponding to 4K equivalent gates.

Post-quantum key encapsulation with Kyber is also evaluated with respect to its security strength levels and parametrizations concerning symmetric primitives and the hardware acceleration of those symmetric primitives via ISEs. The Kyber-90s parametrization is found to be considerably superior to the standard Keccak-based version, with a software-only implementation outperforming hardware-accelerated Keccak-based Kyber. Further hardware acceleration is investigated in the internal Kyber operations in a novel *XKyber* ISE. Hardware acceleration of the symmetric primitives in Kyber-90s show performance and energy gains of 32%. Combining hardware acceleration via *XKyber* and of the symmetric primitives in Kyber-90s, further gains of 46% and 44% in performance and energy consumption are observed, while also reducing code size by 15%. *XKyber* area costs are again of 10% of the baseline Ibex processor with no ISEs.

For future work, one possibility is to investigate the use of TurboSHAKE [Bertoni et al., 2023], a variant of the SHAKE XOFs using the Keccak permutation with 12 rounds instead of the usual 24 rounds, in Kyber. Furthermore, symmetric primitives in Kyber outside of those standardized by NIST can also be investigated. The Grostl [Gauravaram et al., 2009] and Haraka [Kölbl et al., 2016] hash algorithms can benefit from AES hardware acceleration, which can further improve Kyber speeds beyond those measured for Kyber-90s while having a smaller memory footprint due to the exclusion of the costly SHA-256 and SHA-512 round constants.

Additionally, digital signatures also need to be investigated. One option is to use KEM-based authentication [Wiggers et al., 2023] for applications in which messages can be sent back and forth instead of traditional digital signatures. This approach enables reuse of hardware accelerated Kyber, but requires multi-round-trip communication, which is not suitable for all applications, such as authenticating binaries stored in non-volatile memory, for which a traditional digital signature must be used.

The NIST PQC competition also comprises digital signatures, for which 3 algorithms have been chosen for standardization: Dilithium [NIST, 2023a], Falcon [Fouque et al., 2018] and SPHINCS [NIST, 2023c]. Dilithium is closely related to Kyber, where the same techniques for Kyber optimizations as described in this work could still be applied in Dilithium, but would likely lead to smaller gains than in the Kyber case. The SIMD elements in the *XKyber* extension stem from the fact that Kyber polynomial coefficients are 12 bit wide, where 2 coefficients can fit in a single word of a 32 bit RISC-V processor. Dilithium polynomial coefficients are 23 bit wide, where only a single coefficient can fit in a word. Furthermore, larger bit widths for intermediary results in the Barrett reduction would need to be accommodated into the data path and Ibex temporary registers, which would lead to a worse performing ISE at a higher area cost.

Falcon has the smaller public key and signature size, which at a first glance might be make it the most attractive option for digital signatures in a resource-constrained embedded context. However, it requires the use of constant time floating-point operations, which is hard to implement in software-emulated floating point operations, discouraging its use in the context in question, seeing as hardware floating point operations are unlikely to be available. Finally, SPHINCS shows the biggest signature size. For the aforementioned binary authentication from non-volatile memory example, this would not be significant, seeing as non-volatile memory space is not a constraint. Being a hash-based scheme, its performance is heavily reliant on the underlying hash functions performance. In this case, its integration with hardware accelerated Haraka and use alongside KEM-based authentication seems promising.

# REFERENCES


Abdulrahman, A., Hwang, V., Kannwischer, M. J., and Sprenkels, A. (2022). Faster Kyber and Dilithium on the Cortex-M4. Cryptology ePrint Archive, Paper 2022/112. https://eprint.iacr.org/2022/112, February 2024.

Albrecht, M. R., Hanser, C., Hoeller, A., Pöppelmann, T., Virdia, F., and Wallner, A. (2018). Implementing RLWE-based Schemes Using an RSA Co-Processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):169–208. https://doi.org/10.13154/tches.v2019.i1.169-208.

Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., and Petri, R. (2020). ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V. Cryptology ePrint Archive, Paper 2020/049. https://eprint.iacr.org/2020/049, February 2024.

Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehl, D. (2022). CRYSTALS-Kyber – Submission to round 3 of the NIST post-quantum project. https://pq-crystals.org/kyber/resources.shtml, February 2024.

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 (TACO)*, 14(2):1–25. https://doi.org/10.1145/3085572.

Banerjee, U., Ukyab, T. S., and Chandrakasan, A. P. (2019). Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61. https://doi.org/10.13154/tches.v2019.i4.17-61.

Barrett, P. (2020). Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology (CRYPTO)*, pages 311–323. https://doi.org/10.1007/3-540-47721-7_24.

Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G. V., Keer, R. V., and Viguier, B. (2023). TurboSHAKE. Cryptology ePrint Archive, Paper 2023/342. https://eprint.iacr.org/2023/342, February 2024.

Bertoni, G., Daemen, J., Peeters, M., van Assche, G., and van Keer, R. (2012). Keccak Implementation Overview, version 3.2. Technical report, Keccak Team. 59p., https://keccak.team/files/Keccak-implementation-3.2.pdf.

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 *Cryptology*

*and Network Security Conference (CANS)*, pages 526–545. https://doi.org/10.1007/978-3-030-65411-5_26.

Cheng, H., Großschädl, J., Marshall, B., Page, D., and Pham, T. (2022). RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):193–237. https://doi.org/10.46586/tches.v2023.i1.193-237.

Chips Alliance (2023). Rocket Core. https://github.com/chipsalliance/rocket, February 2024.

Daemen, J. and Rijmen, V. (1998). AES Proposal: Rijndael. 45p., https://web.archive.org/web/20070203204845/https://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf, February 2024.

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.

Di Matteo, S., Gerfo, M. L., and Saponara, S. (2023). VLSI Design and FPGA Implementation of an NTT Hardware Accelerator for Homomorphic SEAL-Embedded Library. *IEEE Access*, 11:72498–72508. http://dx.doi.org/10.1109/ACCESS.2023.3295245.

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.

ETH Zurich (2016). PULP platform. https://github.com/pulp-platform/pulpino, February 2024.

Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al. (2018). Falcon: Fast-Fourier lattice-based compact signatures over NTRU. https://falcon-sign.info/, February 2024.

Fritzmann, C., Sharif, U., Mueller-Gritschneder, D., Reinbrecht, C. R. W., Schlichtmann, U., and Sepúlveda, M. J. (2019). Towards Reliable and Secure Post-Quantum Co-Processors based on RISC-V. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1148–1153. https://doi.org/10.23919/DATE.2019.8715173.

Fritzmann, T., Sigl, G., and Sepúlveda, M. J. (2020). RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280. https://doi.org/10.13154/tches.v2020.i4.239-280.

García-Morchón, Ó., Kumar, S. S., and Sethi, M. (2019). Internet of Things (IoT) Security: State of the Art and Challenges. *RFC*, 8576:1–50. https://doi.org/10.17487/RFC8576.

Gauravaram, P., Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., and Thomsen, S. S. (2009). Grøstl-a SHA-3 candidate. 42p., https://perso.uclouvain.be/fstandae/source_codes/hash_atmel/specs/groestl.pdf, February 2024.

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.

Gewehr, C., Moura, N., Luza, L., Bernardon, E., Calazans, N., Garibotti, R., and Moraes, F. G. (2024). Hardware Acceleration of Authenticated Encryption with Associated Data via RISC-V Instruction SetExtensions in Low Power Embedded Systems. In *Latin American Symposium on Circuits and Systems (LASCAS)*, pages 1–5.

Greconici, D. (2020). KYBER on RISC-V. Master's thesis, Radboud University. 69p., https://www.ru.nl/publish/pages/769526/denisa_greconici.pdf.

Huang, J., Zhang, J., Zhao, H., Liu, Z., Cheung, R. C. C., Koç, u. K., and Chen, D. (2022). Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636. http://dx.doi.org/10.46586/tches.v2022.i4.614-636.

Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R. C. C., Koc, C. K., and Chen, D. (2023). Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *CoRR*, abs/2309.00440:1–15. https://arxiv.org/abs/2309.00440.

Intel (2017). TinyCrypt Cryptographic Library. https://github.com/intel/tinycrypt, February 2024.

Karabulut, E. and Aysu, A. (2020). RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 26–32. https://doi.org/10.1109/fpl50879.2020.00016.

Kölbl, S., Lauridsen, M. M., Mendel, F., and Rechberger, C. (2016). Haraka v2 - Efficient Short-Input Hashing for Post-Quantum Applications. Cryptology ePrint Archive, Paper 2016/098. https://eprint.iacr.org/2016/098, February 2024.

Lee, J., Kim, W., Kim, S., and Kim, J.-H. (2022). Post-Quantum Cryptography Coprocessor for RISC-V CPU Core. In *International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–2. https://doi.org/10.1109/iceic54506.2022.9748834.

lowRISC (2017). OpenTitan project. https://github.com/lowRISC/opentitan, February 2024.

lowRISC (2018). Ibex RISC-V Core. https://github.com/lowRISC/ibex, February 2024.

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://doi.org/10.46586/tches.v2021.i1.109-136.

Marshall, B., Page, D., and Pham, T. (2021). Implementing the Draft RISC-V Scalar Cryptography Extensions. In *Hardware and Architectural Support for Security and Privacy*, pages 1–8. https://doi.org/10.1145/3458903.3458904.

Miteloudi, K., Bos, J., Bronchain, O., Fay, B., and Renes, J. (2023). PQ.V.ALU.E: Post-Quantum RISC-V Custom ALU Extensions on Dilithium and Kyber. Cryptology ePrint Archive, Paper 2023/1505. https://eprint.iacr.org/2023/1505, February 2024.

Nannipieri, P., Matteo, S. D., Zulberti, L., Albicocchi, F., Saponara, S., and Fanucci, L. (2021). A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access*, 9:150798–150808. https://doi.org/10.1109/access.2021.3126208.

Nişancı, G., Flikkema, P. G., and Yalçın, T. (2022). Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms. *MDPI Cryptography*, 6(3):41:1–41:29. https://doi.org/10.3390/cryptography6030041.

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

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

NIST (2007a). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, National Institute of Standards and Technology. 39p., https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

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

NIST (2015a). Secure Hash Standard (SHS). Technical report, National Institute of Standards and Technology. 36p., https://doi.org/10.6028/nist.fips.180-4.

NIST (2015b). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, National Institute of Standards and Technology. 37p., https://doi.org/10.6028/nist.fips.202.

NIST (2023a). Module-Lattice-Based Digital Signature Standard. Technical report, National Institute of Standards and Technology. 44p., http://dx.doi.org/10.6028/NIST.FIPS.204.ipd.

NIST (2023b). Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical report, National Institute of Standards and Technology. 39p., http://dx.doi.org/10.6028/NIST.FIPS.203.ipd.

NIST (2023c). Stateless Hash-Based Digital Signature Standard. Technical report, National Institute of Standards and Technology. 47p., http://dx.doi.org/10.6028/NIST.FIPS.205.ipd.

NSA (2023). Commercial National Security Algorithm Suite 2.0. National Security Agency. 10p., https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF, February 2024.

Paar, C. and Pelzl, J. (2009). *Understanding cryptography - A Textbook for Students and Practitioners*. Springer. 372p., https://link.springer.com/book/10.1007/978-3-642-04101-3.

Park, J.-Y., Moon, Y.-H., Lee, W., Kim, S.-H., and Sakurai, K. (2022). A Survey of Polynomial Multiplication With RSA-ECC Coprocessors and Implementations of NIST PQC Round3 KEM Algorithms in Exynos2100. *IEEE Access*, 10:2546–2563. https://doi.org/10.1109/ACCESS.2021.3138807.

Plantard, T. (2021). Efficient Word Size Modular Arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518. http://dx.doi.org/10.1109/TETC.2021.3073475.

Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. *RFC*, 8446:1–160. https://www.rfc-editor.org/info/rfc8446.

Rescorla, E., Tschofenig, H., and Modadugu, N. (2022). The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. *RFC*, 9147:1–61. https://www.rfc-editor.org/info/rfc9147.

RISC-V Foundation (2019). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. https://riscv.org/technical/specifications, February 2024.

RISC-V Foundation (2022). RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions, Document Version v1.0.1. https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar, February 2024.

Saarinen, M.-J. O. (2020). A Lightweight ISA Extension for AES and SM4. *CoRR*, abs/2002.07041:1–4. https://arxiv.org/abs/2002.07041.

Saarinen, M.-J. O. (2022). rvkrypto-fips. https://github.com/rvkrypto/rvkrypto-fips, February 2024.

Satriawan, A., Syafalni, I., Mareta, R., Anshori, I., Shalannanda, W., and Barra, A. (2023). Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations. *IEEE Access*, 11:70288–70316. http://dx.doi.org/10.1109/ACCESS.2023.3294446.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332. https://doi.org/10.1137/S0036144598347011.

Stoffelen, K. (2019). Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology (LATINCRYPT)*, page 323–340. https://doi.org/10.1007/978-3-030-30530-7_16.

University of Bristol (2018). SCARV: a side-channel hardened RISC-V platform. https://github.com/scarv/scarv, February 2024.

Waterman, A. (2016). Design of the RISC-V Instruction Set Architecture. Technical report, EECS Department, University of California, Berkeley. 117p., http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

Wiggers, T., Celi, S., Schwabe, P., Stebila, D., and Sullivan, N. (2023). KEM-based Authentication for TLS 1.3. Technical report, Internet Engineering Task Force. 25p., https://datatracker.ietf.org/doc/draft-celi-wiggers-tls-authkem/02/.

Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., and Zeng, X. (2020). VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684. https://doi.org/10.1109/tcsi.2020.2983185.

Zgheib, A., Potin, O., Rigaud, J.-B., and Dutertre, J.-M. (2021). Extending a RISC-V Core with an AES Hardware Accelerator to Meet IoT Constraints. In *International Conference on SMACD and Conference on PRIME*, pages 1–4. https://ieeexplore.ieee.org/document/9547962.

Zodpe, H. and Shaikh, A. (2021). A Survey on Various Cryptanalytic Attacks on the AES Algorithm. *International Journal of Next-Generation Computing*, 12(2):115–123. https://ijngc.perpetualinnovation.net/index.php/ijngc/article/view/202.


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