IAIK

SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Ferdinand Bachmann

Optimizing Ascon for 32-bit Architectures
Fast Implementations for Xtensa and RISC-V

BACHELOR'S THESIS

Bachelor's degree programme: Computer Science

Supervisors

Maria Eichlseder

Christoph Dobraunig

Institute of Applied Information Processing and Communications

Graz University of Technology

Graz, September 2023

Abstract

With Ascon being selected as the new standard for lightweight cryptography in the NIST Lightweight Cryptography Competition (LWC), we need fast software implementations of Ascon that perform well on existing low-cost and low-power devices.

In this thesis, we present 8 new Ascon assembler implementations for Tensilica Xtensa, 32-bit RISC-V, and the RISC-V bit-manipulation Instruction Set Architecture (ISA) extensions `Zbb`, `Zbkb`, `Zbp`, and `Zbt`. Our implementations achieve a speedup over previous implementations of up to 85.93% on Xtensa, up to 17.54% on RISC-V, and up to 72.42% on RISC-V with ISA extensions.

Our results show that assembler implementations can be much faster than C implementations, especially in situations of high register pressure. We also show that the availability of certain instructions has a high impact on the performance of cryptographic algorithms, especially funnel shift instructions, and to a lesser degree, rotation instructions and instructions that accelerate bit permutations. We expect that assembler implementations of other cryptographic algorithms using similar techniques will yield similar performance improvements.

Keywords: lightweight cryptography · Ascon · Xtensa · RISC-V

1 Introduction

Today, it is increasingly common to find electronic devices and appliances that communicate wirelessly with each other or over the internet. These devices usually contain low-cost and low-power processors that are not well suited to run current standard cryptographic algorithms. The Lightweight Cryptography Competition (LWC) [NIS23] is a cryptographic competition started in 2019 by the US National Institute of Standards and Technology (NIST) with the goal of standardizing cryptographic algorithms for such constrained devices. Ascon, a family of Authenticated Encryption with Associated Data (AEAD) and hashing algorithms designed by Dobraunig, Eichlseder, Mendel, and Schläffer [DEMS21], is the selected standard for LWC [TMC+23] and primary choice for lightweight applications in the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [Ber19b; Ber19a]. Low-cost and low-power devices usually use readily available implementations of existing 8-, 16-, or 32-bit architectures. On such devices, hardware accelerators for Ascon do not exist, so fast software implementations of Ascon are needed.

Related Work. Dobraunig et al. [DEMS23] created the Ascon reference implementations and created and collected several optimized C and assembler implementations for various architectures. Campos et al. [CJL+20] created optimized RISC-V assembler implementations for a variety of LWC candidates, including Ascon, and found an improved description of the Ascon substitution layer.

Contribution. In this thesis, we provide an analysis and optimization of the individual operations needed to implement Ascon, as well as fast concrete software implementations for Tensilica Xtensa and 32-bit RISC-V, as well as for the RISC-V bit-manipulation Instruction Set Architecture (ISA) extensions `Zbb`, `Zbkb`, `Zbp`, and `Zbt`. We provide an overview over optimized implementations of byte order swapping, 64-bit rotations, bit-interleaving, and the Ascon permutation, structured by which types of instructions are needed to implement them efficiently. We present 8 new optimized assembler implementations of the Ascon members Ascon-128a, Ascon-128, Ascon-80pq, Ascon-Hasha, Ascon-Xofa, Ascon-Hash, and Ascon-Xof that achieve performance improvements over previous implementations of up to 85.93% on Xtensa, up to 17.54% on RISC-V, and up to 72.42% on RISC-V with ISA extensions.

Outline. The rest of this thesis is structured as follows: Chapter 2 reviews the necessary cryptographic background. Chapter 3 summarizes the instruction sets of Tensilica Xtensa, 32-bit RISC-V, and the RISC-V bit-manipulation ISA extensions. Chapter 4 presents our optimized implementations. Chapter 5 presents our evaluation methodology and results. In Chapter 6, we conclude our work with some closing thoughts.

2 Cryptographic Background

2.1 Authenticated Encryption

Authenticated Encryption schemes aim to provide three basic cryptographic properties:

- Confidentiality: An adversary should not be able to gain any knowledge about the plaintext without possession of the key. Additionally, an adversary should not be able to gain any knowledge about the key even if they have access to arbitrary plaintext-ciphertext pairs.
- Integrity: The intended recipient should be able to verify that the message was delivered completely and without modifications.
- Authenticity: The intended recipient should be able to verify that the message was created by an authorized sender. An adversary should not be able to craft a message that will be accepted by the recipient without possession of the key.

One way to achieve these properties is to combine an encryption scheme that only provides confidentiality, but not integrity or authenticity, with a Message Authentication Code (MAC) that provides integrity and authenticity. The MAC computes an authentication tag that can be verified before decryption [Aum17].

An alternative way to achieve this is using a cipher that provides all three properties in one algorithm that produces both a ciphertext and an authentication tag. In some applications, it is useful to authenticate additional unencrypted data that is stored alongside or sent together with the ciphertext. Authenticated encryption ciphers that allow this are called Authenticated Encryption with Associated Data (AEAD) ciphers.

The generic interface for AEAD is

$$\mathbf{E}(K, A, M, N) = (C, T)$$
$$\mathbf{D}(K, A, C, T, N) = M \text{ or } \perp,$$

where K is the key, M is the plaintext message, A is the associated data, N is the nonce, a number that has to be chosen differently for every encryption operation with the same key, C is the ciphertext, and T is the authentication tag. The encryption operation produces a ciphertext C and an authentication tag T. The decryption operation results in either the plaintext message M or the error result $\perp$ if the authentication tag is incorrect.

2.2 The Sponge and Duplex Sponge Constructions

The Sponge and Duplex Sponge constructions are methods of constructing cryptographic functions with variable-length input and arbitrary output length based on a permutation P operating on a fixed number of b bits called the *width* of the sponge. A sponge consists of a b-bit state that is split into two parts of r and c bits called the *rate* and the *capacity*.

The Sponge construction was originally developed by Bertoni et al. [BDPV07] as a reference for security properties of hash function designs, but can also be used as a building block to build symmetric primitives such as hash functions, stream ciphers, MACs, or AEAD ciphers [BDPV11; BDPV12].

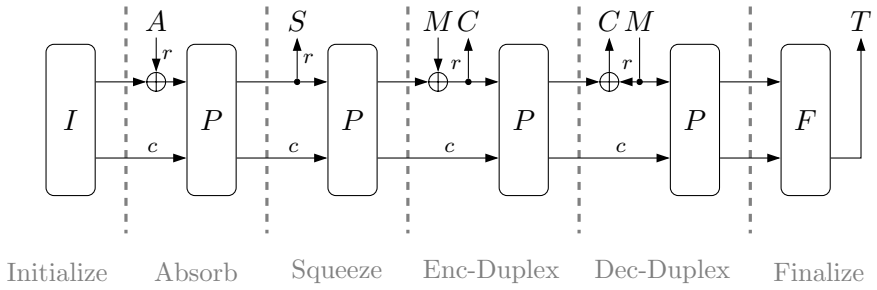


Figure 2.1: Sponge Operations

The initial state of the sponge depends on the specific application, and both keyed and unkeyed forms of initialization exist. For hash functions the initial state is usually a constant, while for MACs or AEAD ciphers, initialization usually involves the key [BDPV11].

The sponge state is usually operated on in phases during which data is processed in blocks of r bits separated by applications of the permutation P. Common sponge operations include *Absorbing*, where r bits of input data are XORed to the rate of the sponge, *Squeezing*, where the r-bit rate is emitted as output data, and various *Duplex* operations that combine absorbing and squeezing (see Figure 2.1).

Ascon uses two variants of a duplex operation in its AEAD ciphers: For *Enc-Duplex*, the input block is XORed to the rate and the new rate before permutation is emitted as output data. For *Dec-Duplex*, the input block is XORed to the rate to produce the output block, then the rate is replaced with the input block before permutation [DEMS21].

Since data is processed by the sponge operations in blocks of r bits, the input data needs to be padded to a multiple of the block size r. Ascon uses a padding scheme where a single 1 bit is always appended, followed by as many 0 bits as needed to reach a multiple of the block size r [DEMS21].

For applications of sponge functions that produce a tag, such as MACs or AEAD ciphers, there is also a finalization phase that computes the tag from the final state of the sponge (see Figure 2.1).

2.3 Ascon

Ascon is a suite of AEAD ciphers, hash functions, and hash functions with arbitrary output length with a design based on the Duplex Sponge construction. It consists of the AEAD ciphers Ascon-128, Ascon-128a, and Ascon-80pq, the hash functions Ascon-Hash and Ascon-Hasha, and the extendable output functions Ascon-Xof and Ascon-Xofa. All schemes in the Ascon suite provide 128-bit security and are based on the same 320-bit permutation [DEMS21].

Ascon has been selected as the primary choice for lightweight applications in Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) [Ber19b][Ber19a] and is the selected standard for Lightweight Cryptography Competition (LWC) [TMC+23].

2.3.1 State and Recommended Parameter Sets

All members of the Ascon suite are built on a sponge with a state size of 320 bits and using the same permutation. The main difference between the members of the suite lies in the size of the sponge rate r (which is also the block size of the cipher or hash), the key size, the number of permutation rounds used for initialization and finalization (p^a), and the number of permutation rounds used between blocks of data (p^b).

The first r bits of the sponge state are referred to as the rate, while the remaining c bits of the sponge state are referred to as the capacity.

For the description of the permutation, the 320-bit state is split into five 64-bit words x_0 through x_4 in big-endian byte order, which means the first byte of the 320-bit state is the most significant byte of x_0 and the last byte of the state is the least significant byte of x_4.

	Bit size of					Rounds	
Name	key	nonce	tag	rate r	capacity c	p^a	p^b
Ascon-128	128	128	128	64	256	12	6
Ascon-128a	128	128	128	128	192	12	8
Ascon-80pq	160	128	128	64	256	12	6

Figure 2.2: Recommended parameter sets for Ascon AEAD ciphers

	Bit size of			Rounds	
Name	hash	rate r	capacity c	p^a	p^b
Ascon-Hash	256	64	256	12	12
Ascon-Hasha	256	64	256	12	8
Ascon-Xof	*any*	64	256	12	12
Ascon-Xofa	*any*	64	256	12	8

Figure 2.3: Recommended parameter sets for Ascon hash functions

All AEAD ciphers in the Ascon suite use a nonce and tag size of 128 bits. The two hash functions Ascon-Hash and Ascon-Hasha use a hash size of 256 bits, while the two extendable output functions Ascon-Xof and Ascon-Xofa can have an arbitrary output length. The recommended parameters are summarized in figures 2.2 and 2.3.

2.3.2 Authenticated Encryption

All AEAD ciphers in the Ascon suite use the same encryption and decryption modes. The encryption and decryption modes consist of 4 phases: The Initialization phase, the Associated Data phase, the Encryption or Decryption phase, and the Finalization phase.

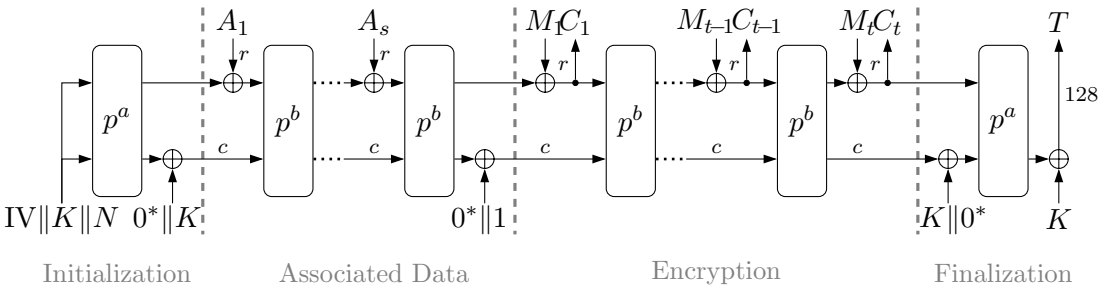


Figure 2.4: Ascon AEAD encryption [DEMS21]

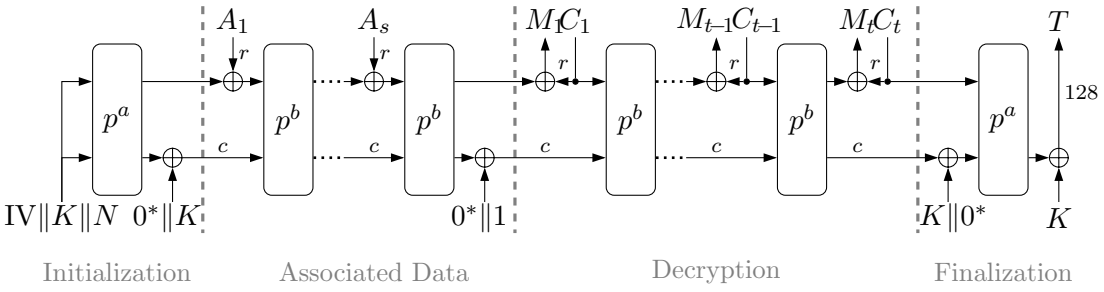


Figure 2.5: Ascon AEAD decryption [DEMS21]

In the initialization phase, the first 32 bits of the state are initialized to an IV consisting of the key size k, the rate r, the initialization and finalization round number a, and the intermediate round number b, each written as 8-bit integers. The next 160 bits of the state are initialized to the key K, padded on the left with zeros. The final 128 bits of the state are initialized to the nonce N. Then, the state is permuted using a rounds of the permutation. Finally, the key is XORed again into the least significant k bits of the capacity c.

In the Associated Data phase, the associated data A is absorbed into the rate r in blocks, separated by b rounds of the permutation. The associated data is padded with a single 1 bit, followed by as many 0 bits as are needed to reach the next block size. After

the associated data and the padding is absorbed, a single 1 bit is XORed to the least significant bit of the capacity to separate the Associated Data phase from the next phase.

In the Encryption phase, the plaintext message M is processed in blocks using the Enc-Duplex operation (see Section 2.2), separated by b rounds of the permutation and yielding the ciphertext C. The plaintext is padded the same way as the associated data in the previous phase. A diagram of the ASCON AEAD encryption mode can be seen in Figure 2.4.

In the Decryption phase, the ciphertext C is processed in blocks using the Dec-Duplex operation (see Section 2.2), separated by b rounds of the permutation and yielding the plaintext. A diagram of the ASCON AEAD decryption mode can be seen in Figure 2.5.

After the last block of plaintext or ciphertext is processed, no permutation is performed.

In the finalization phase, the key is XORed into the most significant k bits of the capacity c, followed by a rounds of the permutation. Then, the key is again XORed to the least significant k bits of the capacity c. The least significant 128 bits of the capacity c then form the authentication tag T. During decryption, decryption succeeds only if the calculated tag T matches the tag T' received with the ciphertext.

2.3.3 Hashing

All hash and extensible output functions in the ASCON suite use the same hashing mode. The hashing mode consists of three phases: The Initialization phase, the Absorbing phase, and the Squeezing phase.

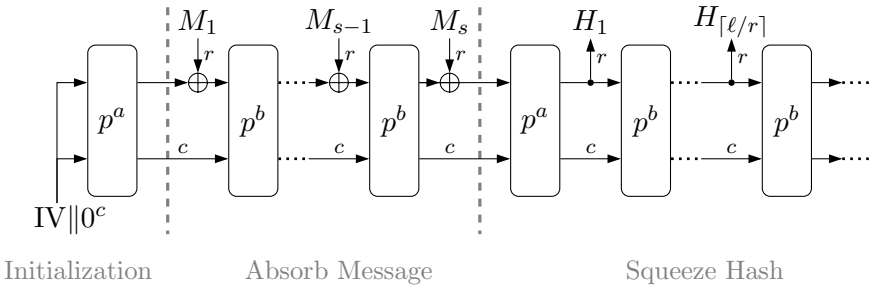


Figure 2.6: ASCON Hashing [DEMS21]

In the initialization phase, the first 64 bits of the state are initialized to an IV consisting of the constant 0, the rate r, the initialization round number a, and the difference between the initialization and intermediate round numbers $a - b$, each written as 8-bit integers, followed by the hash output length h written as a 32-bit big-endian integer. For the extensible output functions ASCON-XOF and ASCON-XOFA, h is set to 0. Then, the state is permuted using a rounds of the permutation. Since the IV is constant, the result of the initialization phase can be precomputed.

In the Absorbing phase, the message is absorbed into the rate r in blocks, separated by b rounds of the permutation. The message is padded the same way as the associated data in the AEAD modes.

After the last block of the message is processed, a rounds of the permutation are performed instead of b rounds.

In the Squeezing phase, the hash value is squeezed from the rate r in blocks, separated by b rounds of the permutation.

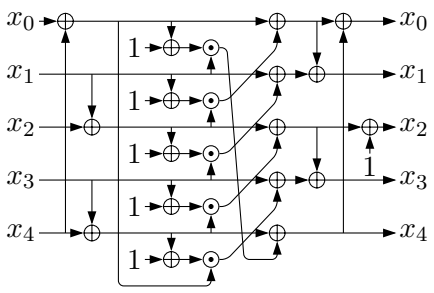
2.3.4 Permutation

The Ascon permutation is the core primitive from which the Ascon suite is built. The Ascon permutation is a 320-bit permutation constructed using multiple rounds of a Substitution Permutation Network (SPN). In each round of the permutation, three operations are performed: A round constant addition, a substitution layer, and a linear diffusion layer.

Starting Round	p^{12}				p^8		p^6					
Constant c_r	`f0`	`e1`	`d2`	`c3`	`b4`	`a5`	`96`	`87`	`78`	`69`	`5a`	`4b`

Figure 2.7: Ascon round constants

For the round constant addition, an 8-bit round constant is XORed to the least significant bits of x_2. The round constants for Ascon can be seen in Figure 2.7. Depending on the number of rounds used, a different round constant is used in the first round, while the round constant of the last round is always `4b`. Note that the next round constant can be calculated by subtracting `0f` from the previous round constant.



(a) Ascon substitution layer

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

(b) Ascon linear layer

Figure 2.8: Ascon substitution layer and linear layer [DEMS21]

The substitution layer consists of a 5-bit Sbox that is applied in parallel to all 64 bits of the state words x_0 through x_4. A bit-sliced implementation of the substitution layer can be seen in Figure 2.8a.

The linear diffusion layer consists of five linear functions Σ_i that Xor rotated copies of the state words to themselves. Each Σ_i rotates the state words by a different amount. A definition of the linear layer can be seen in Figure 2.8b.

3 Target Architectures

In this thesis, we focus mainly on optimized implementations of Ascon for the 32-bit Xtensa and RISC-V architectures. However, most of the implemented optimizations are quite general and may apply to a larger variety of 32-bit architectures. This chapter describes the instruction sets and register model of the aforementioned architectures.

Section 3.1 describes common operations supported by both Xtensa and RISC-V that we consider typical for 32-bit architectures and their notation as used in this thesis. Sections 3.2 and 3.3 describe details specific to the Xtensa and RISC-V architectures.

3.1 Common Architecture Details

Both the Xtensa and 32-bit RISC-V Instruction Set Architecture (ISA) are Reduced Instruction Set Computing (RISC) architectures. They are both register architectures with at least 16 general purpose registers and three register operands per instruction. Both architectures implement a common subset of instructions common in 32-bit RISC architectures, although with slight differences in the availability of immediate operands [Cad22; And19]. This similarity allows general reasoning about optimized assembler implementations that apply to both architectures and likely also to similar 32-bit architectures. This section describes this common subset, some of the differences, and the notation used for such operations in this thesis.

Figure 3.1 shows a non-exhaustive summary of common instructions between Xtensa and RISC-V. The common subset consists of addition, subtraction, bitwise And, Or, and Xor, logical and arithmetic shift, instructions for loading constants, moving values between registers, loading and storing values to and from memory, conditional and unconditional branch instructions, and subroutine call and return instructions. Xtensa and RISC-V differ slightly in which instructions support immediate operands. This is signified by a subscript $_{\text{i}}$ next to instructions which support immediate operands. RISC-V has more support for immediate operands with arithmetic and bitwise instructions, while Xtensa has support for immediate operands in conditional branches.

Some instructions not in the common subset that are nevertheless useful in implementing optimized cryptographic algorithms are additionally listed in Figure 3.2. These instructions include bitwise rotation, funnel shifts (double-width shift instructions), bitwise instructions that negate their second operand, dedicated byte order swap instructions, and bit permutation instructions. Xtensa only supports funnel shifts and bitwise rotations using the `src` instruction. The base RISC-V instruction set has no support for any of these instructions, but support is provided by various RISC-V instruction set extensions (see Section 3.3.3).

Xtensa	RISC-V	Symbol	Description
`add`$_i$	`add`$_i$	$\boxplus$	Addition
`sub`	`sub`$_i$	$\boxminus$	Subtraction
`and`	`and`$_i$	$\odot$	Bitwise And
`or`	`or`$_i$	$\oplus$	Bitwise Or
`xor`	`xor`$_i$	$\oplus$	Bitwise Xor
`sll`$_i$	`sll`$_i$	$\ll$	Logical shift left
`srl`$_i$	`srl`$_i$	$\gg$	Logical shift right
`sra`$_i$	`sra`$_i$		Arithmetic shift right
`movi`	`li`		Load immediate
`mov`	`mv`		Move between registers
`l8ui`	`lbu`		8-bit unsigned indirect load
`l32i`	`lw`		32-bit indirect load
`s8i`	`sb`		8-bit indirect store
`s32i`	`sw`		32-bit indirect store
`beq`$_i$	`beq`		Branch if equal
`bne`$_i$	`bne`		Branch if not equal
`bltu`$_i$	`bltu`		Branch if unsigned less than
`bgeu`$_i$	`bgeu`		Branch if unsigned greater than or equal
`j`	`j`		Unconditional jump
`call0`	`jal`		Subroutine call
`ret`	`jalr`		Subroutine return

Figure 3.1: Common 32-bit RISC instructions

Xtensa	RISC-V	Extension	Symbol	Description
`src`	`ror`$_i$	(`Zbb`)	$\ggg$	Rotate right
`src`	`fsr`$_i$	(`Zbt`)	$\gggg$	Funnel shift right
	`andn`	(`Zbb`)		Bitwise And with negated operand
	`orn`	(`Zbb`)		Bitwise Or with negated operand
	`rev8`	(`Zbb`)		Byte order swap
	`zip`	(`Zbkb`)		Interleave even and odd bits in register
	`unzip`	(`Zbkb`)		Separate even and odd bits in register
	`pack`	(`Zbkb`)		Pack low halves of two registers into one register
	`packu`	(`Zbp`)		Pack high halves of two registers into one register

Figure 3.2: Other useful 32-bit RISC instructions

3.2 Xtensa

The Xtensa ISA is the instruction set used on the Xtensa LX series of processors by Tensilica, Inc. Xtensa is used in the Espressif ESP32 line of microcontrollers.

3.2.1 Architecture

The Xtensa ISA is a 32-bit RISC instruction set with 16 and 24-bit instruction words and up to three register operands per instruction [Cad22].

The architecture supports addressing 16 general-purpose registers numbered `a0` to `a15` and a special-purpose register `sar` used for holding the shift amount for shift and rotate instructions. `a0` is used as to hold the return address and `a1` holds the stack pointer, which leaves 14 freely usable registers for computation.

The Xtensa architecture has a "register window" feature that allows to change which 16 registers out of a larger register file are accessible to the program. This is used to reduce the amount of loads and stores needed to save register values between function calls. The window can be shifted by 0, 4, 8, or 12 registers at a time, which makes it possible to efficiently use the larger register file if some functions in the call stack don't need to use a lot of registers.

The architecture also has a hardware-assisted looping feature, which can be used to execute loops that iterate a known number of times at the same speed that an unrolled version of the loop would execute at.

3.2.2 Instruction Set

The Xtensa Instruction Set contains a moderate number of instructions for different purposes, most of which are not relevant to the implementation of cryptographic algorithms such as Ascon. For this reason, we will focus on a relevant subset of the available instructions and highlight interesting additions or omissions that have to be worked around.

The arithmetic and bitwise instructions `add`, `sub`, `and`, `or`, and `xor` each have three operands (one destination and two source operands) and can operate on any of the 16 general-purpose registers. Shifted addition and subtraction instructions, where the second operand is shifted left by a constant are also available: `addx2`, `addx4`, `addx8`, `subx2`, `subx4`, `subx8`.

Notable is the omission of immediate variants of all arithmetic instructions except addition: `addi` and `addmi` can be used to add signed 8-bit immediates, where the latter shifts the immediate left by 8, allowing 16-bit constants to be added via a two-instruction sequence. Due to the omission of immediate variants of bitwise operations bitwise negation can only be done by loading -1 into a register and using the `xor` instruction.

The `movi` instruction can be used to load 12-bit signed immediates into a register, all other immediates need to be loaded from a constant pool in memory using the relative load instruction `l32r`.

The left shift (`sll`) and right shift (arithmetic `sra` and logical `srl`) instructions take two register operands (destination and source), the shift amount is taken from the `sar` special register. For constant shift amounts, immediate variants exist (`slli`, `srai`, `srli`). The shift amount register can be set using the `ssr` and `ssai` instructions, setting the shift amount from a register or immediate respectively.

Rotate instructions do not exist, but the funnel shift instruction `src`, which shifts a concatenated pair of registers to the right and stores the lower 32-bit half of the result can be used to implement rotations by concatenating a register to itself. A funnel shift can also be used to efficiently implement rotations wider than register width, such as the 64-bit rotations used in Ascon. No immediate variant for `src` exists, the shift amount is always taken from `sar`.

Loads from and stores to memory through pointers and a constant offset can be performed using `l8ui`, `l16ui`, `l32i` (load instructions) and `s8i`, `s16i`, `s32i` (store instructions).

Conditional branches take two register arguments and a constant branch offset and perform the comparison in the same instruction. There are no separate compare instructions. The available branch conditions are equal (`beq`), not equal, (`bne`), signed less than `blt`, signed greater or equal `bge`, and unsigned variants of the former (`bltu`, `bgeu`). Immediate variants of these instructions also exist (`beqi`, `blti`, `bgei`, `bltui`, `bgeui`).

The zero overhead looping feature can be used by using the `loop` instruction for loops shorter than 256 bytes of instructions, writing to the hardware registers controlling the looping feature directly, or using the `floop` and `floopend` assembler macros, which automatically choose the correct way of setting up the loop.

3.3 RISC-V

RISC-V is an open ISA designed at University of California, Berkeley. Many open source and proprietary implementations exist. RISC-V is also used in the Espressif ESP32-C3 microcontroller.

3.3.1 Architecture

The RISC-V specification defines 32-, 64-, and 128-bit RISC instruction sets with 32-bit instruction words and up to three register operands. RISC-V is composed of a base instruction set and a set of instruction set extensions [And19]. In this thesis, we mainly focus on the 32-bit base instruction set (RV32I), with some references to useful instructions from the bit manipulation and scalar cryptography extensions (see Section 3.3.3).

The RISC-V architecture allows addressing 32 registers `x0` to `x31`. The `zero` (`x0`) register is hard-wired to the value 0. The `ra` (`x1`) register is used for function return addresses. The `sp` (`x2`) register is used as a stack pointer. The `gp` (`x3`) and `tp` (`x4`) registers point to global and thread-local storage. Registers `a0` to `a7` (`x10` to `x17`) are used to pass function arguments, where registers `a0` and `a1` are also used to pass return

values. Registers `s0` to `s11` (`x8` to `x9` and `x18` to `x27`) are callee-saved registers. Registers `t0` to `t6` (`x5` to `x7` and `x28` to `x31`) are temporary registers not preserved across calls.

Aside from the `zero`, `ra`, `sp`, `gp`, and `tp` registers, this leaves 27 freely usable registers for computation, 15 of which can be used without saving registers to the stack.

3.3.2 Base Instruction Set

The RV32I base instruction set is a compact instruction set containing only integer operations.

The arithmetic and bitwise instructions `add`, `sub`, `and`, `or`, `xor` each have three operands (one destination and two source operands) and can operate on any of the 32 registers. Each of these instructions except `sub` has an immediate variant (`addi`, `andi`, `ori`, `xori`) with a 12-bit signed immediate as its second operand.

The `lui` instruction loads a 20-bit unsigned immediate into the upper bits of a register, allowing to load arbitrary 32-bit constants using a 2 instruction `lui` and `addi` sequence.

The left shift (`sll`) and right shift (arithmetic `sra` and logical `srl`) instructions take three operands (destination, source register, and shift amount register). For all three shift instructions, immediate variants exist that allow specifying a constant shift amount (`slli`, `srai`, `srli`). The base instruction set notably lacks both rotate and funnel shift instructions.

Loads from and stores to memory through pointers and a constant offset can be performed using `lb`, `lbu`, `lh`, `lhu`, `lw` (load instructions) and `sb`, `sh`, `sw` (store instructions).

Conditional branches take two register arguments and a constant branch offset and perform the comparison in the same instruction. The available branch conditions are equal (`beq`), not equal (`bne`), signed less than `blt`, signed greater or equal `bge`, and unsigned variants of the former (`bltu`, `bgeu`). Immediate variants of these instructions do not exist, constants must be loaded into a register before performing the branch.

The `slt` instruction performs a signed less than comparison and stores the resulting boolean value into a register. The `sltu` instruction performs an unsigned less than comparison instead. Immediate variants of these instructions are also available (`slti`, `sltiu`).

3.3.3 Instruction Set Extensions

To serve different use cases for the architecture, the RISC-V specification defines a set of instruction set extensions that define new instructions or specify specific behaviour for existing instructions. RISC-V extensions are usually named with an uppercase letter, such as `F` for the standard extension for single-precision floating point, or with the uppercase letter `Z` followed by lowercase letters, such as the `Zicsr` standard extension for control and status registers [And19].

The Compressed Instruction extension version 2.0 [And19] defines a number of alternative encodings for often-used instructions with 16-bit instruction words to reduce code size. It also relaxes the alignment requirement of instructions to 16-bit alignment,

allowing compressed and regular encodings to be mixed freely. The `C` extension is part of the base ISA and reduces code size by approximately 25-30%.

The Bit-Manipulation extension version 1.0.0 [RIS21] contains a number of instructions useful for low-level bit manipulation, which are often needed for implementing optimized cryptographic operations. The full standard extension `B` has not been defined yet, but parts of the extension have been ratified. The currently ratified bit manipulation extensions are the `Zba` extension for address generation, the `Zbb` extension for basic bit manipulation, the `Zbc` extension for carry-less multiplication, and the `Zbs` extension for working with individual bits in a register. Most relevant for this thesis are the `andn`, `orn`, `ror`, and `rev8` instructions from `Zbb`.

The draft version 0.93 of the Bit-Manipulation extension [Wol21] also contained the `Zbp` extension for bit permutations and the `Zbt` extension for "ternary" bit manipulation instructions that have a third source register operand. The `Zbp` extension contains instructions useful for performing bit-interleaving, such as the `pack`, `packu`, `zip`, and `unzip` instructions. The `Zbt` extension contains the `fsr` funnel shift instruction.

Some of the bit manipulation instructions missing from the Bit-Manipulation extension version 1.0.0 have been ratified as part of the Scalar Cryptography extension version 1.0.1 [Mar22], along with specialized instructions for some cryptographic algorithms. The full standard extension `K` has not been defined yet, but parts of the extension have been ratified. The `Zbkb` extension contains, among others, the instructions `pack`, `zip`, and `unzip` from the draft version of `Zbp`, but not the `packu` instruction. The Scalar Cryptography extension also defines the `Zkt` extension, which guarantees that a large subset of non-control-flow instructions in RISC-V execute with a latency independent from the data processed, reducing side channels from potential optimizations such as when the data is zero.

RISC-V implementations presented in this thesis use the base RISC-V 32-bit ISA and the `C` extension for compressed instructions. Some Ascon implementations presented in this thesis also use instructions from the `Zbb`, `Zbkb`, `Zbp`, or `Zbt` extensions to show their relevance for optimized implementations of cryptographic algorithms.

4 Implementation

In this chapter, we describe our optimized implementations of Ascon for Xtensa and RISC-V in detail. First, we motivate our choice of implementing Ascon in Assembler, then we discuss optimized implementations of common operations in Ascon. Finally, we describe our implementation of the Ascon permutation and the encryption and hashing modes in detail.

4.1 Choice of Programming Language

Optimized implementations of cryptographic algorithms focused on either execution speed or code size are often written in compiled languages like C. In C, a major factor for execution speed and code size are optimizations done by the compiler. Therefore, optimized implementations of cryptographic algorithms in C have to be written with the underlying machine in mind and tailored to the optimizations done by the compiler.

One common optimization performed by the compiler is inlining, where the compiler copies the body of a called function into the calling function. This allows the compiler more register allocation freedom, since it no longer has to move values into specific registers or memory to conform to the function calling convention. Inlining removes the function call overhead, but can sometimes drastically increase code size.

In Assembler, we have much more control over register allocation and argument passing. Functions that are private to the optimized implementation don't have to conform to the calling convention, and we can choose any registers to pass the arguments in. This can reduce the function call overhead to be almost as low as with inlining, without duplicating the code at all call sites of the function.

This reduction of function call overhead is especially important for Ascon, since the compiler would need to move the sponge state to and from memory at every call to or return from a subroutine like the permutation.

For this reason we choose to write our optimized Ascon implementations in Assembler. The flexibility of Assembler allows creating implementations that optimize for both execution speed and code size.

4.2 Endianness

The Ascon specification defines Ascon in terms of big-endian arithmetic and byte order. However, many current architectures, including Xtensa and RISC-V, are little-endian architectures. Therefore, Ascon implementations on such architectures need to convert the byte order of all loaded state words to little-endian before doing any calculations, and convert them back to big-endian before storing the results.

```
slli t0,  x,  24
srli t1,  x,  24
or   t0, t0,  t1
srli t1,  x,  16
andi t1, t1, 255
slli t1, t1,   8
or   t0, t0,  t1
srli  x,  x,   8
andi  x,  x, 255
slli  x,  x,  16
or    x,  x,  t0
```

(a) shift-and-Or

```
slli t0,  x,  24
srli  x,  x,   8
or    x,  x,  t0
srli t0,  x,  16
xor  t0, t0,   x
andi t0, t0, 255
xor   x,  x,  t0
slli t0, t0,  16
xor   x,  x,  t0
```

(b) shift-and-Xor

```
rori  x,  x,   8
srli t0,  x,  16
xor  t0, t0,   x
andi t0, t0, 255
xor   x,  x,  t0
slli t0, t0,  16
xor   x,  x,  t0
```

(c) rotate-and-Xor

```
srli t0,  x, 16
fsri t0,  x, t0, 8
fsri t0, t0, t0, 8
fsri  x, t0,  x, 8
```

(d) funnel shift

Figure 4.1: Optimized implementations of byte order swapping

A common idiom for swapping the byte order is to use a sequence of And, shift, and Or operations to mask out individual bytes and shift them to their swapped positions. This results in an implementation that needs at least 11 operations and 2 temporary registers, as shown in Figure 4.1 (a). The same operation can be implemented more efficiently by using shift and Xor operations to swap two bytes at the same time, which results in a 9 operation implementation and only 1 temporary register (b). When rotate instructions are available, this can be reduced to 7 operations (c). When the architecture used supports funnel shifts, such as Xtensa or the RISC-V `Zbt` extension, byte order swapping can be implemented in only 4 operations (d) [Cad22].

The C compiler builtin `__builtin_bswap32` will expand to a function call to the library function `__bswapsi2` in libgcc, which uses a different implementation depending on the architecture and compiler version. On Xtensa, the compiler uses the shift-and-Or implementation (a) up until GCC 10, and the funnel shift implementation (d) in GCC 11 and later. On RISC-V, GCC uses the shift-and-Or implementation (a) for the base instruction set and the dedicated byte order swap instruction `rev8` for Instruction Set Architecture (ISA) extensions that support it. This means that beside from the additional overhead of a function call, `__builtin_bswap32` is suboptimal on Xtensa before GCC 11 and on RISC-V without ISA extensions on all compiler versions.

For our Ascon implementations, we use the optimized shift-and-Xor implementation for the RISC-V base ISA (b), the funnel shift implementation for Xtensa (d), and the dedicated byte order swap instruction `rev8` on RISC-V with ISA extensions.

4.3 Alignment

On many CPU architectures, memory addresses must be aligned to a multiple of the register size when loading from or storing to memory. On architectures where unaligned accesses to memory are allowed, they often result in a performance penalty.

Our Ascon implementation assumes unaligned accesses to memory are allowed. The associated data, plaintext, and ciphertext buffers should be aligned to 4 bytes when using our implementation. This ensures unaligned accesses occur only when loading or storing the authentication tag for messages who's length is not a multiple of the rate.

4.4 64-bit Rotation

The Ascon permutation uses 64-bit right rotations in its linear layer, which are not native operations on 32-bit architectures like Xtensa and RISC-V. On such architectures, wide rotations must be constructed from multiple instructions.

```
slli t0, xh, 32-N
slli t1, xl, 32-N
srli xh, xh, N
srli t0, xl, N
or   xh, xh, t1
or   xl, xl, t0
```

(a) shift-and-Or

```
slli t0, xh, 32-N
slli t1, xl, 32-N
srli xh, xh, N
srli t0, xl, N
xor  xh, xh, t1
xor  xl, xl, t0
```

(b) shift-and-Xor

```
fsri t0, xh, xl, N
fsri xh, xl, xh, N
mv   xl, t0
```

(c) funnel shift

```
rori xe, xe, (N+1)/2
rori xo, xo, N/2
```

or, for odd N:

```
rori t0, xe, (N+1)/2
rori xe, xo, N/2
mv   xo, t0
```

(d) bit-interleaved

Figure 4.2: Implementation of 64-bit right rotation by N on 32-bit architectures

A common implementation of 64-bit right rotation is to use two left and right shift operations, and combining the results with two Or operations. This results in an implementation that needs 6 operations and 2 temporary registers, as shown in Figure 4.2 (a). The Or operations can be replaced with Xor operations, since the results of the left and right shift operations never overlap (b). The shift-and-Xor implementation is more useful for implementing Ascon than the shift-and-Or implementation, since the Xor operations commute with the Xor operations in the Ascon linear layer.

When the architecture supports funnel shifts, such as Xtensa or the RISC-V `Zbt` extension, 64-bit right rotation can be implemented in only 3 operations and 1 temporary register (c). The third operation is only used to move the temporary result into a different register, which can be elided in most use cases through careful register allocation, leading

to this implementation effectively needing only 2 operations, a third of the shift-and-Or implementation (a).

When 32-bit rotate instructions are available, but funnel shifts are not available, 64-bit rotation can also be realized using a technique called *bit-interleaving*. For bit-interleaving, the 64-bit word needs to be stored in registers in bit-interleaved form, meaning one of the registers contains all of the bits at even positions (`xe`) and one register contains all of the bits at odd positions (`xo`) instead of the usual low and high halves (`xl` and `xh`). The 64-bit rotate can then be realized by performing two 32-bit rotates by half the amount on the even and odd halves. When rotating by an odd amount, the two registers must additionally be swapped, leading to an implementation needing 2 or 3 operations and 0 or 1 temporary registers (d). The third operation can often be elided similar to the funnel shift implementation.

Converting to and from bit-interleaved representation is an expensive operation (see Section 4.5), so it is only worth doing when the conversion needs to only be done once. When rotate instructions are not available, bit-interleaving is not worth doing since emulating the rotates with shifts and Or or Xor operations does not give an advantage over implementations (a) or (b) but still incurs the high cost of converting to and from bit-interleaved representation.

Implementations (a), (b), and (c) only work for rotate amounts less than 32. For rotation amounts 32 and higher one has to additionally swap the destination registers `xh` and `xl`, which effectively performs an additional rotation by 32 bits, and rotate instead by the original rotation amount minus 32.

For our Ascon implementations, we use the shift-and-Xor implementation for the RISC-V base ISA (b), the funnel shift implementation for Xtensa and RISC-V with the `Zbt` extension (c), and the bit-interleaved implementation for RISC-V with the `Zbb`, `Zbkb`, or `Zbp` extensions. The bit-interleaved implementation is not viable on the RISC-V base ISA and Xtensa, since the RISC-V base ISA does not have rotate instructions, and Xtensa has funnel shift instructions, which are a better alternative.

4.5 Bit-Interleaving

Bit-interleaving is a technique for accelerating rotation operations where words are stored in bit-interleaved representation, meaning the word is separated into its even and odd halves (`xe` and `xo`) instead of its low and high halves (`xl` and `xh`). Converting to and from bit-interleaved representation is an expensive operation on most architectures since it involves many separate swaps of groups of bits.

Separating a 32-bit word into its even and odd halves is also called the "perfect outer unshuffle" [War13] and can be implemented by successively exchanging groups of bits. The same can be implemented for 64-bit words by performing a perfect outer unshuffle on both halves and then exchanging the high 16 bits of the low half with the low 16 bits of the high half. The inverse can be performed by doing the operations in reverse order.

```
srli t0, x,  off
xor  t0, t0, x
and  t0, t0, mask
xor  x,  x,  t0
slli t0, t0, off
xor  x,  x,  t0
```

(a) `bitxchg` operation

```
srli t0, xl, 16
xor  t0, t0, xh
slli t0, t0, 16
xor  xl, xl, t0
srli t0, t0, 16
xor  xh, xh, t0
```

(b) `halfxchg` operation

```
li t1, 0x22222222
bitxchg xl, t1, 1
bitxchg xh, t1, 1
li t1, 0x0C0C0C0C
bitxchg xl, t1, 2
bitxchg xh, t1, 2
li t1, 0x00F000F0
bitxchg xl, t1, 4
bitxchg xh, t1, 4
li t1, 0x0000FF00
bitxchg xl, t1, 8
bitxchg xh, t1, 8
halfxchg xl, xh
```

(c) bit-interleave 64-bit

```
li t1, 0x22222222
bitxchg xl, t1, 1
bitxchg xh, t1, 1
li t1, 0x0C0C0C0C
bitxchg xl, t1, 2
bitxchg xh, t1, 2
li t1, 0x000F000F
bitxchg xl, t1, 12
bitxchg xh, t1, 12
li t1, 0x000000FF
bitxchg xl, t1, 24
bitxchg xh, t1, 24
halfxchg xl, xh
```

(d) swap byte order and bit-interleave 64-bit

Figure 4.3: Implementation of bit-interleaving on 32-bit architectures

Figure 4.3 shows different implementations of bit-interleaving 64-bit words on 32-bit architectures. (a) shows the bit exchange operation by Warren [War13], (b) shows the exchanging of the high and low halves of two registers, (c) shows an implementation of 64-bit bit-interleaving in 58 operations, and (d) shows a variant of the same implementation that also performs a byte order swap at the same time at no additional cost.

```
unzip t0, xl
unzip xh, xh
pack  xl, t0, xh
rori  t0, t0, 16
rori  xh, xh, 16
pack  xh, t0, xh
```

(e) bit-interleave 64-bit on RISC-V `Zbkb`

```
unzip t0, xl
unzip xh, xh
pack  xl, t0, xh
packu xh, t0, xh
```

(f) bit-interleave 64-bit on RISC-V `Zbp`

Figure 4.4: Implementation of bit-interleaving using RISC-V ISA extensions

Using the bit permutation instructions from the RISC-V bit-manipulation extensions `Zbkb` and `Zbp` bit-interleaving can be performed much faster. Figure 4.4 shows an implementation of bit-interleaving 64-bit words in 6 instructions for `Zbkb` (e) and an implementation in 4 instructions for `Zbp`.

4.6 Permutation

The Ascon permutation is the core primitive of the Ascon cipher suite and is therefore the most important target for optimization. In this section, we present our optimized assembler implementations of the permutation for Xtensa, the RISC-V base ISA, and for the RISC-V extensions `Zbb`, `Zbkb`, `Zbp`, and `Zbt`.

4.6.1 State and Notation

The 320-bit state of Ascon requires 10 registers of storage on 32-bit architectures. The two halves of the five 64-bit state words `x0` through `x4` are each stored in 2 registers (e.g. `x0l` and `x0h` for the low and high halves of `x0`). Our implementations require 4 additional registers. One for storing the round constant, which is also used as a loop control variable, and three for storing intermediate results. This is just enough to still fit into the general purpose register file of both Xtensa and RISC-V, allowing the permutation to be implemented without accesses to memory. To avoid unnecessary memory accesses, our implementation expects the state and initial round constant in registers when the permutation is called, and will return the permuted state in those same registers, ignoring the platform calling convention, which usually has only one or two registers for returning values. For better readability we provide register diagrams for each stage of the permutation. An example of such a diagram can be seen in Figure 4.5.

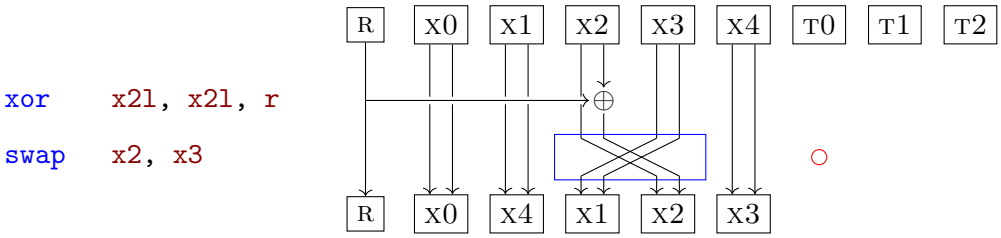


Figure 4.5: Permutation Register Diagram Example

The left side of the diagram shows the assembler instructions, while the right side of the diagram shows a data flow graph. Each column in the data flow graph represents a register. The register pairs for each of the 5 state words `x0` through `x4` are drawn closer together and have a shared label to save space in the diagram. For these register pairs, the left column represents the high bits (e.g. `x0h`) and the right column represents the low bits (e.g. `x0l`). Simple operations such as the Xor instruction in the first line of Figure 4.5 are drawn as a single symbol in the data flow graph. Operands to these operations are noted with arrows from the register they read from to the operation that needs them. Complex operations that consist of multiple instructions or operate on multiple registers at the same time, such as the swap operation in the second line of Figure 4.5 are not written out in full, but are instead represented by a blue box around the registers they act on and a red circle on registers used as temporaries. The details of these complex operations are shown in a separate diagram.

4.6.2 Constant Addition and Loop Control

Every round, a round constant c_r is XORed to the low half of the state word x_2. The round constants in ASCON have been chosen in such a way that for each round, the next round constant c_{r+1} is always $c_r - 15$. Additionally, the round constant for the last round is always `0x4b`. Therefore the initial round constant is enough information to derive all round constants in the permutation and to derive the number of rounds.

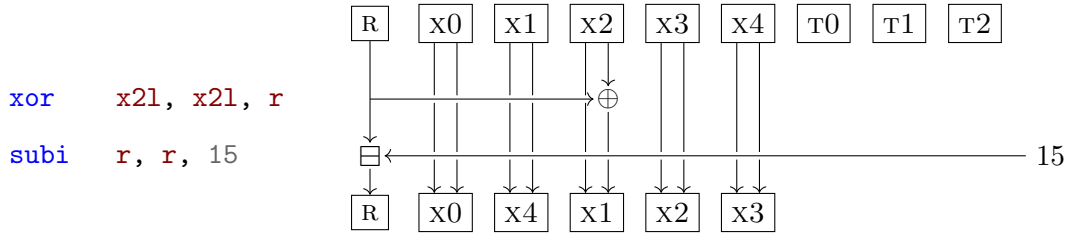


Figure 4.6: Constant Addition Register Diagram

This allows implementing ASCON without the need for a round constant lookup table and without the need for separate round number and round constant variables. After each round, the current round constant is decreased by 15 and then compared against the final round constant. On Xtensa, we additionally make use of a hardware-assisted looping feature that requires the number of iterations to be supplied in advance. The hardware keeps track of the current iteration internally, so the register used for passing the iteration count is free for other uses after setting up the loop. This means we can use one of the temporary registers for passing the round count to the permutation and do not need another register. Figure 4.6 shows a register diagram of the constant addition and constant update after each round.

For bit-interleaved implementations, a round constant lookup table is still needed, since the even and odd bits of the round constants don't follow such a simple pattern. Additionally, separate round constants have to be XORed to the even and odd halves of x_2. For those implementations, the variable `r` is replaced by a pointer to the round constant lookup table that is advanced by two bytes every round.

4.6.3 Substitution Layer

Our implementation of the substitution layer is based on the 17 operation and 3 temporary register SBOX implementation by Campos et al. [CJL+20]. The implementation notably results in the state being rotated right by two registers: `x0` ends up in the register for `x2`, `x1` in `x3`, `x2` in `x4`, `x3` in `x0`, and `x4` in `x1`. In order to simplify reordering the state back into the original order, Campos et al. choose to place `x3` in a temporary register instead.

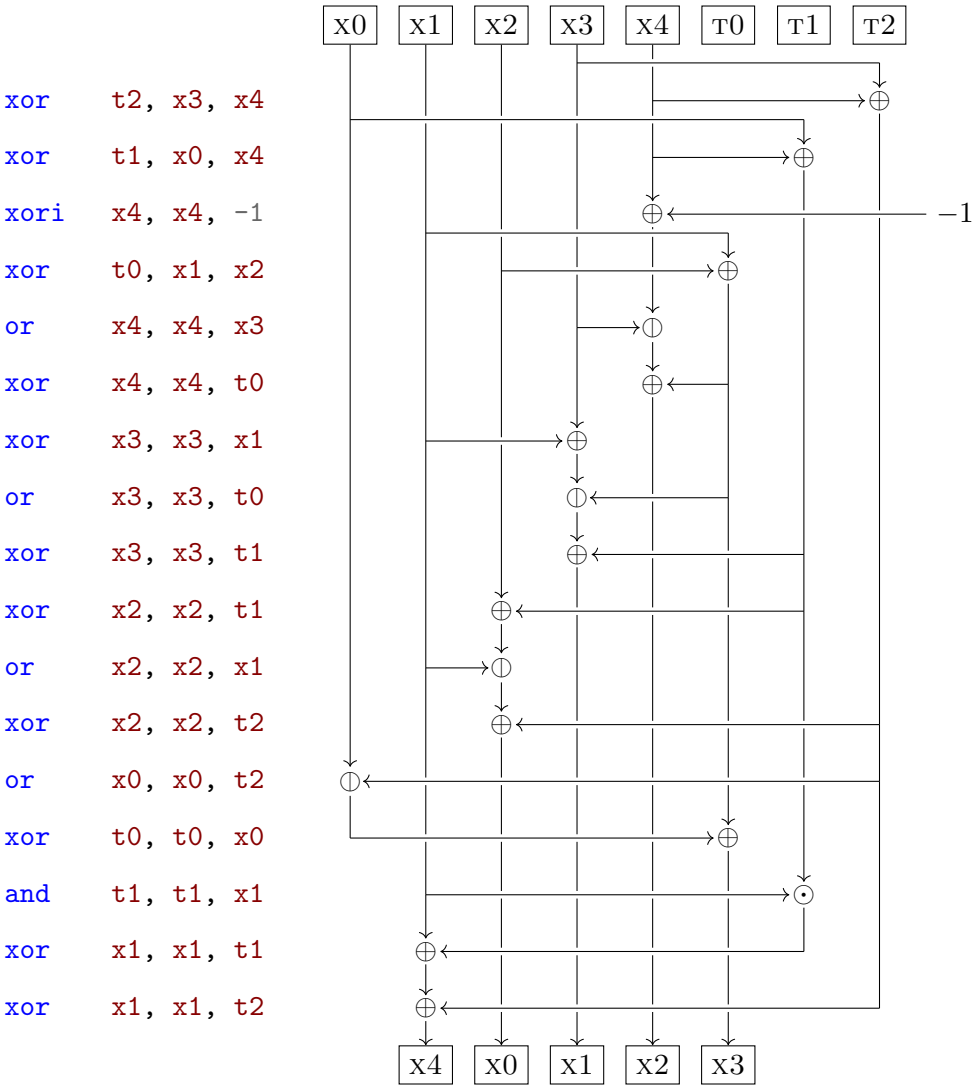


Figure 4.7: Modified Campos et al. SBOX Register Diagram

The Campos et al. substitution layer is not directly usable on Xtensa, since Xtensa does not have a NOT or XOR with immediate instruction. The Campos et al. substitution layer contains two NOT operations. Emulating a NOT instruction requires a two instruction sequence and an extra temporary register. However, one of the two NOT instructions

is part of an A And (Not B) sequence, which can be rewritten as (A And B) Xor A. The implementation can also be reordered to free up a temporary register so that the Not instruction can be emulated without needing an extra temporary register. This results in an 18 instruction implementation on Xtensa. Figure 4.7 shows our modified implementation of the Campos et al. substitution layer.

On RISC-V with ISA extensions, both Not instructions can be optimized further using the `andn` and `orn` instructions, resulting in a 15 instruction implementation.

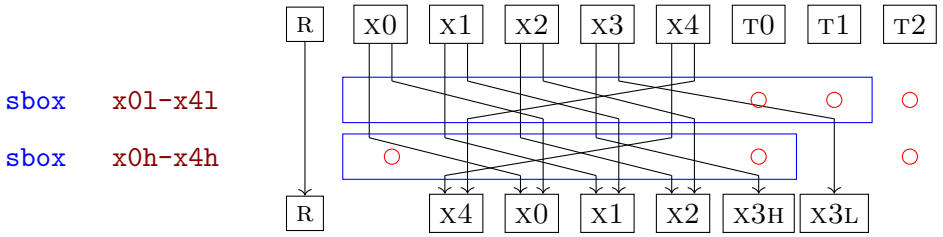


Figure 4.8: Substitution Layer Register Diagram

The Sbox operation needs to be applied twice, once on the lower halves of the state words (`x0l-x4l`) and once on the upper halves (`x0h-x4h`). For the low half, we choose `t0`, `t1`, and `t2` as temporary registers for the Sbox operation, with the result for `x3l` being placed in `t1`. For the high half, we choose `t0`, `t2`, and the now unused register `x0l` as temporary registers, with the result for `x3h` being placed in `t0`. This can be seen in Figure 4.8. This reuse of registers that become unused in previous operations as temporaries allows our implementation of the full substitution layer to use only 3 temporary registers, whereas Campos et al. used 5 temporary registers in total for their substitution layer since they used dedicated temporary registers for the results for `x3l` and `x3h`.

4.6.4 Linear Layer

Our implementation of the linear layer is again based on the implementation by Campos et al. In addition to applying the linear diffusion functions Σ_i of the Ascon linear layer, the implementation also has to undo the rotation of state words by the substitution layer. This is done by successively moving the state words back into their original register in the order `x0`, `x2`, `x4`, `x1`, `x3`.

Our implementation again uses fewer temporary registers than Campos et. al, since we reuse the registers that become free as temporaries for the next Σ_i.

```
slli dh, xl, 32-r0
srli t0, xh, r0
xor  dh, dh, t0
slli t0, xl, 32-r1
xor  dh, dh, t0
srli t0, xh, r1
xor  dh, dh, t0
slli dl, xh, 32-r0
srli t0, xl, r0
xor  dl, dl, t0
slli t0, xh, 32-r1
xor  dl, dl, t0
srli t0, xl, r1
xor  dl, dl, t0
xor  dl, dl, xl
xor  dh, dh, xh
```

(a) shift-and-xor

```
rori de, xo, (r0-r1)/2
rori do, xe, (r0-r1)/2
xor  de, de, xo
xor  do, do, xe
rori de, de, (r1-1)/2
rori do, do, (r1+1)/2
xor  de, de, xe
xor  do, do, xo
```

for `r0`, `r1` odd

```
rori de, xo, (r0-1)/2
xor  de, de, xe
rori do, xe, (r0+1)/2
xor  do, do, xo
rori xe, xe, r1/2
rori xo, xo, r1/2
xor  de, de, xe
xor  do, do, xo
```

for `r1` even

(b) bit-interleaved

```
fsri dl, xl, xh, r0
fsri dh, xh, xl, r0
xor  dl, dl, xl
xor  dh, dh, xh
fsri t0, xl, xh, r1
fsri xh, xh, xl, r1
xor  dl, dl, t0
xor  dh, dh, xh
```

(c) funnel shift

Figure 4.9: Linear Layer implementations

The linear diffusion functions Σ_i each consist of two 64-bit rotations by different constant amounts and two 64-bit Xor operations. Figure 4.9 shows our implementations for different implementations of 64-bit rotation. For the RISC-V base ISA, we use the shift-and-Xor implementation of 64-bit rotation (a). This results in 16 instructions, since each 64-bit Xor needs 2 operations and each 64-bit rotation needs 6 operations.

On RISC-V with the `Zbb`, `Zbkb`, or `Zbp` extension, we use the bit-interleaved implementation (b). This results in only 8 instructions, since each 64-bit rotation can be implemented in only two rotation operations. There are two variants of this implementation that have to be used when the rotation amounts are both odd, or when at least one of the rotation amounts is even.

On RISC-V with the `Zbt` extension, we use the funnel-shift implementation of 64-bit rotation. This results in only 8 instructions, since each 64-bit rotation only needs two operations with funnel shift available. On Xtensa, we also use the funnel-shift implementation. The Xtensa implementation needs 10 instructions, however, since rotation amounts for funnel shifts need to be set using a separate instruction on Xtensa. One of these extra instructions can be elided, since both Σ_4 and Σ_1 are implemented using a rotation by 7 (mod 32), so the rotation amount register only has to be set once for both.

```
linear x0, x2, 19, 28
linear x2, x4, 1, 6
linear x4, x1, 7, 41
linear x1, x3, 61, 39
linear x3, t0, t1, 10, 17
```

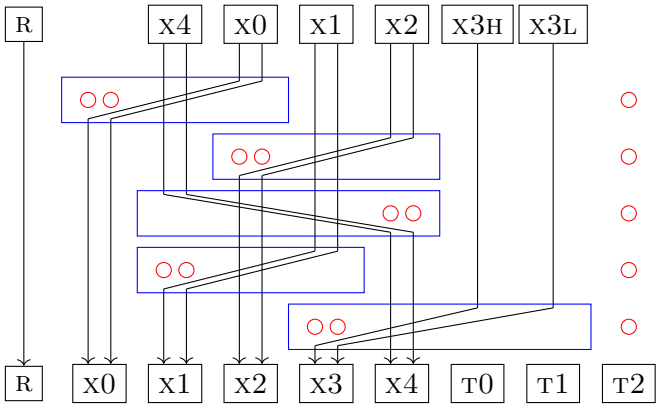


Figure 4.10: Linear Layer Register Diagram

Figure 4.10 shows the order in which the Σ_i functions are applied to move the state words back into their original positions and to ensure at least three registers are always available for use as temporaries.

4.6.5 Round Diagram

A diagram for a full round of the Ascon permutation can be seen in Figure 4.11. As can be seen in the diagram, the state reordering performed by the Campos et al. Sbox is reverted by the linear layer and every state word ends up back in the correct register.

```
xor    x2l, x2l, r
sbox   x0l-x4l
sbox   x0h-x4h
linear x0, x2, 19, 28
linear x2, x4, 1, 6
linear x4, x1, 7, 41
linear x1, x3, 61, 39
linear x3, t0, t1, 10, 17
subi   r, r, 15
```

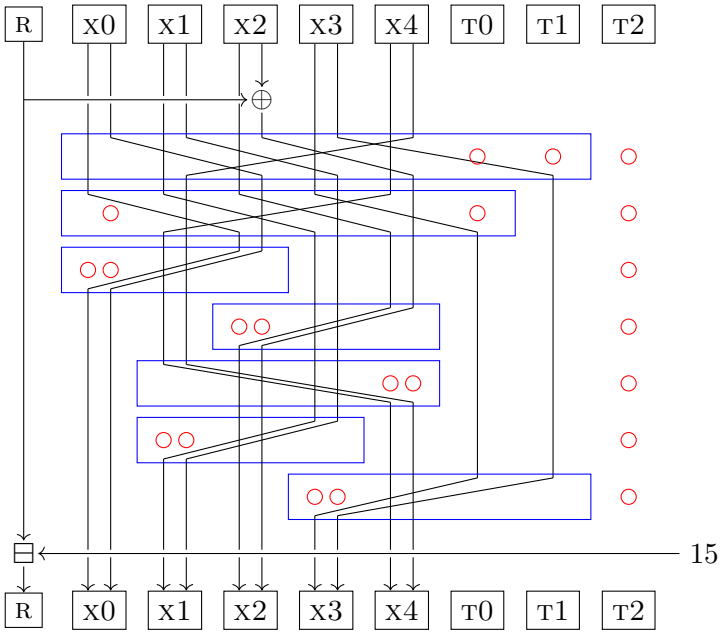


Figure 4.11: Ascon Permutation Round Diagram

4.7 Encryption and Hashing Modes

For implementing the encryption and hashing modes we focused our optimizations on the variable length data processing phases of the mode, i.e. absorbing the associated data and duplexing the plaintext or ciphertext for the encryption modes, and absorbing the message and squeezing the hash for the hashing mode.

The required variables that must be kept in registers for these phases are the rate part of the Ascon state (`x0` and `x1` for the Ascon-128a encryption modes, only `x0` for all other modes and variants), the input and/or output pointers, and the length of the block of data. To reduce code size, we use the same subroutine `ascon_duplex` for the associated data, encryption, and decryption phases of the encryption modes. For this we need to add an additional variable "mode" that differentiates between the phases. The key can be stored in memory, since it is only needed for initialization and finalization. On Xtensa, the key is always stored in memory, since there aren't enough free registers available to keep it in registers. On RISC-V, the key is stored fully in registers, with

the exception of Ascon-80pq, where the extra 32 bits of the key need to be stored in memory as well.

This results in up to 8 registers being used for the phase state in the worst case (the plaintext or ciphertext phase in Ascon-128a), leaving 6 free registers for temporary values and other state on Xtensa. This means the capacity part of the Ascon state needs to be stored on the stack between calls of the permutation. On RISC-V the state can be kept in registers permanently due to the high number of general purpose registers available.

During the absorbing and duplex phases, each block of data that is loaded from the input needs to be byte-swapped, since both Xtensa and RISC-V are little-endian architectures. This is accomplished by the subroutine `ascon_rev8`, which swaps the byte order of one block of data. Parameters for this subroutine are passed using the remaining free registers, to avoid needing to store the state to memory. For bit-interleaved implementations, this subroutine is called `ascon_to_bi32_rev8`, since it also needs to convert to bit-interleaved representation.

During the squeezing and duplex phases, each block of data needs to be byte-swapped again to convert back to big-endian byte order. We again call the subroutine `ascon_rev8`, though for some implementations this subroutine needs to be inlined to free up an additional temporary register due to register pressure. For bit-interleaved implementations, the inverse of bit-interleaving needs to be performed, which is implemented in the subroutine `ascon_from_bi32_rev8`.

At the end of each data processing phase, the final block needs to be padded to a full block size, and a single 1 bit has to be added to signify the end of the block. Previous implementations have implemented this by iterating the bytes in the block and using bit operations to place the data bytes and the padding bit into the correct bit-position of the correct variable (see the `opt64` implementation by Dobraunig et al. [DEMS23]). We implement the padding by copying the final block to a one block large stack buffer, storing the padding bit using a store byte operation, and then loading the block into registers the same way as for the previous blocks. This results in smaller code and more performance for data that is not a multiple of the block size.

The initialization, finalization, and sequencing of the phases is implemented in the function `ascon_core`. The function follows the platform calling convention to facilitate it being called from C and uses the subroutines defined above. For the decryption mode, this function also compares the calculated authentication tag with the stored tag in the ciphertext, and returns whether the decryption succeeded.

5 Evaluation

In this chapter, we describe our evaluation environment and methodology, we list the implementations we benchmarked and evaluated, we report on our benchmarking results, and discuss our findings.

5.1 Evaluation Environment and Methodology

We evaluated our implementations on three different target platforms:

- **ESP32:** The Espressif ESP32 is a series of microcontrollers popular for use in Internet of Things (IoT) applications based on the Tensilica Xtensa LX6 CPU architecture. The board we used is the ESP32-DevKitC-V4, which contains an ESP32-D0WDQ6 CPU running at 240 MHz.

  We use the ESP32 as a reference platform for evaluating our Xtensa implementations.

- **ESP32-C3:** The Espressif ESP32-C3 is a newer series of microcontrollers popular for use in IoT applications based on the RISC-V RV32IMC CPU architecture. The board we used in the ESP32-C3-DevKitM-1, which contains an ESP32-C3FN4 CPU running at 160 MHz.

  We use the ESP32-C3 as a reference platform for evaluating our RISC-V implementations.

- **riscvOVPsim+:** The riscvOVPsim+ simulator developed by Imperas Software is a configurable instruction-level RISC-V simulator available for free after registration. riscvOVPsim+ supports simulating many RISC-V Instruction Set Architecture (ISA) extensions, including the `Zbb` extension from RISC-V Bit-Manipulation version 1.0.0, `Zbp` and `Zbt` from RISC-V Bit-Manipulation version 0.92 (which does not have functional differences to 0.93 for the instructions we are using), as well as the `Zbkb` extension from RISC-V Scalar Cryptography version 1.0.1.

  We use riscvOVPsim+ as a reference platform for evaluating our implementations that use RISC-V ISA extensions.

We used our own custom-made benchmarking framework *cryptobench*[1] to measure our implementations on the target platforms. The framework consists of a C application running on the target, which receives commands from a Python application running on the host. The framework is inspired by work by Renner, Pozzobon, and Mottok [RPM22]

[1] `https://gitlab.tugraz.at/247B03DB02C337C2/lwc-cryptobench`

on benchmarking US National Institute of Standards and Technology (NIST) Lightweight Cryptography Competition (LWC) candidates and supports testing the same candidate implementations.

The target C application is compiled separately for each evaluated implementation to rule out interferences between implementations. The protocol with the host supports receiving requests for encryption, decryption, or hashing with a specific key, authenticated data, message or ciphertext, as well as requests with random inputs of a specific length. The target then performs the specified operation once to ensure relevant code is cached, and then measures the cycle count of performing the operation a specified number of times. The total cycle count and the output is reported back to the host. The protocol also supports a benchmarking command that allows measuring the performance of different message lengths in a single command and allows taking multiple samples that are reported back to the host separately.

The host Python application first compiles the implementation to be tested, and flashes it onto the target device. The host then tests the correctness of the algorithm by asking the target to evaluate the known answer tests from the NIST LWC submission. The performance of the algorithm is measured by using the benchmarking command with two message lengths: 16 bytes (128 bits) to test short messages, and 32 kilobytes to test long messages. The Python application supports outputting a human readable summary of the benchmark results, as well as in JavaScript Object Notation (JSON) to facilitate automated processing of the results.

5.2 Implementations

We evaluated and benchmarked 11 implementations for the Ascon members Ascon-128a, Ascon-128, Ascon-80pq, Ascon-Hasha, Ascon-Xofa, Ascon-Hash, and Ascon-Xof. We evaluated two C implementations from the Ascon implementation repository [DEMS23], one C and assembler implementation for 32-bit RISC-V by Campos et al. [CJL+20], and 8 new assembler implementations, two for Xtensa, one for the RISC-V 32-bit base ISA, and 5 for various RISC-V ISA extensions. A summary of the tested implementations can be seen in Figure 5.1, with our implementations marked with an asterisk (*). The column labeled "Strategy" denotes the operations used to implement the Ascon linear layer, as well as "b.i." if the implementation uses bit-interleaved representation internally. For C implementations, the strategy was assessed by reviewing the generated assembly code. For RISC-V implementations that use ISA extensions, the "Notes" column also lists notable instructions used from the extension.

Name	Architecture	Strategy	Notes
	ESP32		
`c_opt64_lowsize`	Xtensa LX6	shift	[DEMS23]
`c_opt64`	Xtensa LX6	shift	[DEMS23]
`asm_xtensa_bi32_ror*`	Xtensa LX6	b.i., `ror`	
`asm_xtensa_fsr*`	Xtensa LX6	`fsr`	
	ESP32-C3		
`c_opt64_lowsize`	RISC-V 32-bit	shift	[DEMS23]
`c_opt64`	RISC-V 32-bit	shift	[DEMS23]
`asm_rv32_campos`	RISC-V 32-bit	shift	[CJL+20]
`asm_rv32_shift*`	RISC-V 32-bit	shift	
	riscvOVPsim+		
`c_opt64_lowsize`	RISC-V 32-bit	shift	[DEMS23]
`c_opt64`	RISC-V 32-bit	shift	[DEMS23]
`asm_rv32_campos`	RISC-V 32-bit	shift	[CJL+20]
`asm_rv32_shift*`	RISC-V 32-bit	shift	
`asm_rv32_Zbb_shift*`	+`Zbb`	shift	with `andn`, `orn`, `ror`, `rev8`
`asm_rv32_Zbb_bi32_ror*`	+`Zbb`	b.i., `ror`	with `andn`, `orn`, `ror`, `rev8`
`asm_rv32_Zbkb_bi32_ror*`	+`Zbkb`	b.i., `ror`	with `zip`, `unzip`, `pack`
`asm_rv32_Zbp_bi32_ror*`	+`Zbp`	b.i., `ror`	with `packu`
`asm_rv32_Zbt_fsr*`	+`Zbt`	`fsr`	with `fsr`

Figure 5.1: Evaluated Implementations

5.3 Results

We evaluated and benchmarked all 11 implementations for the Ascon members Ascon-128a, Ascon-128, Ascon-80pq, Ascon-Hasha, Ascon-Xofa, Ascon-Hash, and Ascon-Xof on the target platforms ESP32, ESP32-C3, and riscvOVPsim+. We benchmarked each implementation separately for long inputs and short inputs.

Name	Ascon-128a	Ascon-128 Ascon-80pq	Ascon-Hasha Ascon-Xofa	Ascon-Hash Ascon-Xof
		ESP32		
c_opt64_lowsize	98.76 c/B	141.53 c/B	178.04 c/B	260.14 c/B
c_opt64	85.01 c/B	127.49 c/B	156.95 c/B	230.07 c/B
asm_xtensa_bi32_ror*	70.87 c/B	100.09 c/B	115.47 c/B	166.54 c/B
asm_xtensa_fsr*	51.01 c/B	77.18 c/B	95.68 c/B	139.24 c/B
		ESP32-C3		
c_opt64_lowsize	78.35 c/B	112.81 c/B	140.88 c/B	205.97 c/B
c_opt64	70.83 c/B	102.90 c/B	129.44 c/B	194.27 c/B
asm_rv32_campos	70.13 c/B	n/a	n/a	n/a
asm_rv32_shift*	66.30 c/B	97.35 c/B	124.00 c/B	183.34 c/B
		riscvOVPsim+		
c_opt64_lowsize	76.18 c/B	110.36 c/B	138.66 c/B	202.70 c/B
c_opt64	68.85 c/B	101.10 c/B	128.15 c/B	189.69 c/B
asm_rv32_campos	68.79 c/B	n/a	n/a	n/a
asm_rv32_shift*	64.79 c/B	94.85 c/B	121.17 c/B	179.72 c/B
asm_rv32_Zbb_shift*	58.65 c/B	87.59 c/B	114.91 c/B	171.46 c/B
asm_rv32_Zbb_bi32_ror*	54.71 c/B	74.45 c/B	84.87 c/B	122.40 c/B
asm_rv32_Zbkb_bi32_ror*	41.44 c/B	61.19 c/B	78.23 c/B	115.77 c/B
asm_rv32_Zbp_bi32_ror*	40.94 c/B	60.69 c/B	77.98 c/B	115.52 c/B
asm_rv32_Zbt_fsr*	38.62 c/B	57.56 c/B	74.85 c/B	111.39 c/B

Figure 5.2: Performance in cycles per byte, for long inputs (32 kB)

For evaluating the performance on long inputs, we took 32 samples of encrypting and decrypting 32 kilobytes of data 4 times. The resulting cycle count for each sample is divided by 4 to get an average number of cycles for a single encryption or decryption. Figure 5.2 shows the average performance of each implementation for each platform and for each Ascon member in processor cycles per byte. The implementations are sorted by descending performance for Ascon-128a for each target platform. Benchmarking results for encryption and decryption are averaged together since they don't differ significantly, and the Ascon members Ascon-128 and Ascon-80pq, Ascon-Hasha and Ascon-Xofa, as well as Ascon-Hash and Ascon-Xof are also grouped together for the same reason. Results for each Ascon member are available separately, but are omitted for brevity.

Name	Ascon-128a	Ascon-128 Ascon-80pq	Ascon-Hasha Ascon-Xofa	Ascon-Hash Ascon-Xof
		ESP32		
c_opt64	918.61 c/B	874.50 c/B	564.61 c/B	760.94 c/B
c_opt64_lowsize	398.83 c/B	441.23 c/B	588.36 c/B	793.44 c/B
asm_xtensa_bi32_ror*	293.27 c/B	310.49 c/B	442.69 c/B	595.77 c/B
asm_xtensa_fsr*	214.51 c/B	239.00 c/B	365.68 c/B	496.25 c/B
		ESP32-C3		
c_opt64	1548.91 c/B	942.02 c/B	902.54 c/B	7484.96 c/B
c_opt64_lowsize	320.40 c/B	354.51 c/B	472.86 c/B	635.43 c/B
asm_rv32_campos	303.58 c/B	n/a	n/a	n/a
asm_rv32_shift*	273.79 c/B	301.60 c/B	471.46 c/B	649.71 c/B
		riscvOVPsim+		
c_opt64_lowsize	310.37 c/B	344.30 c/B	461.14 c/B	621.14 c/B
asm_rv32_campos	285.71 c/B	n/a	n/a	n/a
c_opt64	277.21 c/B	307.99 c/B	431.01 c/B	584.95 c/B
asm_rv32_shift*	265.59 c/B	292.46 c/B	459.70 c/B	635.20 c/B
asm_rv32_Zbb_shift*	240.77 c/B	268.84 c/B	435.08 c/B	604.58 c/B
asm_rv32_Zbb_bi32_ror*	222.52 c/B	231.87 c/B	325.01 c/B	437.51 c/B
asm_rv32_Zbkb_bi32_ror*	172.71 c/B	190.46 c/B	298.51 c/B	411.01 c/B
asm_rv32_Zbp_bi32_ror*	170.84 c/B	189.27 c/B	297.51 c/B	410.01 c/B
asm_rv32_Zbt_fsr*	160.77 c/B	178.84 c/B	285.08 c/B	394.58 c/B

Figure 5.3: Performance in cycles per byte, for short inputs (16 B)

For evaluating the performance on short inputs, we took 32 samples of encrypting and decrypting 16 bytes of data of data 32 times. The resulting cycle count for each sample is divided by 32 to get an average number of cycles for a single encryption or decryption. Figure 5.3 shows the average performance of each implementation for each platform and for each Ascon member in processor cycles per byte. The implementations are sorted and grouped the same way as in the diagram for long inputs, with some reversals in the order due to differing performance. Results for each Ascon member are available separately, but are omitted for brevity.

5.4 Discussion

Figures 5.4 and 5.5 show the performance difference in percent between each implementation and the fastest previous implementation for each Ascon member and for each target platform. The fastest previous implementation is marked "ref".

Name	Ascon-128a	Ascon-128 Ascon-80pq	Ascon-Hasha Ascon-Xofa	Ascon-Hash Ascon-Xof
		ESP32		
c_opt64_lowsize	+13.93 %	+9.92 %	+11.84 %	+11.56 %
c_opt64	ref	ref	ref	ref
asm_xtensa_bi32_ror*	−19.95 %	−27.38 %	−35.93 %	−38.15 %
asm_xtensa_fsr*	−66.65 %	−65.19 %	−64.04 %	−65.23 %
		ESP32-C3		
c_opt64_lowsize	+10.50 %	+8.79 %	+8.12 %	+5.68 %
c_opt64	+0.99 %	ref	ref	ref
asm_rv32_campos	ref	n/a	n/a	n/a
asm_rv32_shift*	−5.77 %	−5.70 %	−4.39 %	−5.96 %
		riscvOVPsim+		
c_opt64_lowsize	+9.69 %	+8.39 %	+7.58 %	+6.42 %
c_opt64	+0.08 %	ref	ref	ref
asm_rv32_campos	ref	n/a	n/a	n/a
asm_rv32_shift*	−6.19 %	−6.59 %	−5.76 %	−5.55 %
asm_rv32_Zbb_shift*	−17.29 %	−15.43 %	−11.52 %	−10.63 %
asm_rv32_Zbb_bi32_ror*	−25.75 %	−35.79 %	−51.00 %	−54.97 %
asm_rv32_Zbkb_bi32_ror*	−66.01 %	−65.23 %	−63.80 %	−63.85 %
asm_rv32_Zbp_bi32_ror*	−68.04 %	−66.59 %	−64.33 %	−64.21 %
asm_rv32_Zbt_fsr*	−78.12 %	−75.65 %	−71.20 %	−70.30 %

Figure 5.4: Performance difference in percent, for long inputs (32 kB)

On average, all implementations perform 4.76 times worse ($\sigma = 5.05$) on short inputs compared to long inputs. Some slowdown is expected, since the cost of initialization and finalization is not offset by the cost of processing the data for short messages. The C implementation `c_opt64` has the largest difference between short and long message performance, performing 6.14 times worse on short messages on ESP32, and 19.13 times worse on ESP32-C3. This is likely caused by the `c_opt64` implementation having a large amount of instruction cache misses that result in very poor short message performance, but comparably good long message performance.

Overall, the performance of the new implementations on short inputs is comparable or better than the performance on long inputs for the Ascon Authenticated Encryption with Associated Data (AEAD) members, but the performance gain is only around half as high on the Ascon hash members of the Ascon family on short inputs. This is likely

due to the simpler Ascon hashing mode being easier to optimize for compilers due to lower register pressure and potentially also partly caused by the fact that we optimized the implementations for Ascon-128a first, and only ported them to the other variants.

On ESP32, the fastest implementation for all Ascon members is `asm_xtensa_fsr`, the funnel shift implementation, followed by `asm_xtensa_bi32_ror`, the bit-interleaved implementation. This is expected, since the funnel shift implementation does not have to pay the additional cost of converting between bit-interleaved and non-bit-interleaved representations. The performance improvement on short inputs is bigger than for long inputs due to the poor performance of `c_opt64` on short inputs, leading to `c_opt64_lowsize` being the comparison implementation for Ascon-128a and Ascon-128 for short inputs.

Name	Ascon-128a	Ascon-128 Ascon-80pq	Ascon-Hasha Ascon-Xofa	Ascon-Hash Ascon-Xof
		ESP32		
`c_opt64`	+56.58 %	+49.54 %	ref	ref
`c_opt64_lowsize`	ref	ref	+4.04 %	+4.10 %
`asm_xtensa_bi32_ror`*	−35.99 %	−42.11 %	−27.54 %	−27.72 %
`asm_xtensa_fsr`*	−85.93 %	−84.61 %	−54.40 %	−53.34 %
		ESP32-C3		
`c_opt64`	+80.40 %	+62.37 %	+47.61 %	+91.51 %
`c_opt64_lowsize`	+5.25 %	ref	ref	ref
`asm_rv32_campos`	ref	n/a	n/a	n/a
`asm_rv32_shift`*	−10.88 %	−17.54 %	−0.30 %	+2.20 %
		riscvOVPsim+		
`c_opt64_lowsize`	+10.68 %	+10.55 %	+6.53 %	+5.83 %
`asm_rv32_campos`	+2.98 %	n/a	n/a	n/a
`c_opt64`	ref	ref	ref	ref
`asm_rv32_shift`*	−4.38 %	−5.31 %	+6.24 %	+7.91 %
`asm_rv32_Zbb_shift`*	−15.13 %	−14.57 %	+0.93 %	+3.25 %
`asm_rv32_Zbb_bi32_ror`*	−24.58 %	−32.83 %	−32.61 %	−33.70 %
`asm_rv32_Zbkb_bi32_ror`*	−60.51 %	−61.71 %	−44.39 %	−42.32 %
`asm_rv32_Zbp_bi32_ror`*	−62.27 %	−62.72 %	−44.87 %	−42.67 %
`asm_rv32_Zbt_fsr`*	−72.42 %	−72.22 %	−51.19 %	−48.25 %

Figure 5.5: Performance difference in percent, for short inputs (16 B)

On ESP32-C3, our implementation `asm_rv32_shift` is the fastest implementation for all Ascon members on long inputs and for the Ascon AEAD members on short inputs. For hashing short inputs, `asm_rv32_shift` is around the same performance as `c_opt64_lowsize` for Ascon-Hasha and slightly slower for Ascon-Hash.

On riscvOVPsim+, the results look very similar for the implementations that also run on ESP32-C3, though there is not as much of a performance breakdown for `c_opt64` due to riscvOVPsim+ not simulating an instruction cache. `asm_rv32_shift` performs slightly worse than on ESP32-C3, but still constitutes an improvement of around 5% on long inputs. The fastest implementations by far are the funnel shift implementation for RISC-V `Zbt` and the bit-interleaved implementations for RISC-V `Zbp`, `Zbkb`, and `Zbb`, achieving a performance improvement on long inputs of up to 78.12% for `Zbt`, and 68.01%, 66.01%, and 54.97% for `Zbp`, `Zbkb`, and `Zbb` respectively. `asm_rv32_Zbb_bi32_ror` performed worse on Ascon AEAD members than for hashing, likely due to the expensive conversion to bit-interleaved representation, which has to be performed twice per block on AEAD members, and only once for hashing.

To summarize, our implementations perform significantly better on long inputs than previous implementations on all platforms. They also perform significantly better on short inputs for Ascon AEAD members, but there is room for improvement in the performance on short inputs in general, and for the Ascon hash members in particular.

6 Conclusion

With Ascon being selected as the new standard for lightweight cryptography in the US National Institute of Standards and Technology (NIST) Lightweight Cryptography Competition (LWC) [TMC+23], we need fast software implementations of Ascon that perform well on existing low-cost and low-power devices. Existing implementations for Xtensa and RISC-V are mostly written in C [DEMS23] or in a mix of C and assembler [CJL+20], resulting in suboptimal performance in cases where the compiler fails to optimize the generated assembly.

In this thesis, we analyzed and presented different assembler implementations of building blocks necessary to implement Ascon, such as byte order swapping, 64-bit rotation operations, bit-interleaving, and the Ascon permutation. We presented 8 new assembler implementations for the Ascon members Ascon-128a, Ascon-128, Ascon-80pq, Ascon-Hasha, Ascon-Xofa, Ascon-Hash, and Ascon-Xof for Xtensa, RISC-V, and the RISC-V Instruction Set Architecture (ISA) extensions `Zbb`, `Zbkb`, `Zbp`, and `Zbt`. We evaluated our implementations on the Espressif ESP32 for Xtensa, the Espressif ESP32-C3 for RISC-V, and the Imperas riscvOVPsim+ simulator for RISC-V with ISA extensions. Our implementations achieved significant performance improvements over previous implementations, achieving up to 85.93% more performance on Xtensa, up to 17.54% on RISC-V, and up to 72.42% on RISC-V with ISA extensions.

Our results show that assembler implementations can be much faster than C implementations, especially in situations of high register pressure such as with Ascon-128a. Compilers also sometimes miss optimizations when certain instruction idioms are unknown to the compiler, such as with the funnel shift on Xtensa, leading to subpar performance on byte order reversal and rotation operations. Our results also show that in simpler cases, compilers are able to compete with hand-crafted assembler implementations, such as with our Ascon-Hash implementation for the RISC-V base ISA. We also showed that the availability of certain instructions has a high impact on the performance of cryptographic algorithms, especially funnel shift instructions on Xtensa and RISC-V `Zbt`, and to a lesser degree, rotation instructions and instructions that accelerate bit permutations such as `zip` and `unzip` on RISC-V `Zbkb` and `Zbp`.

We expect that assembler implementations of Ascon and other cryptographic algorithms using similar techniques will yield similar performance improvements. Future research could evaluate how well optimizations like ours apply to other architectures such as ARM or to other cryptographic algorithms. We expect that ISA extensions like the RISC-V bit-manipulation or scalar cryptography extensions have a lot of potential for optimizing a large variety of cryptographic algorithms.

Acronyms

AEAD	Authenticated Encryption with Associated Data	3, 4, 5, 6, 7, 8, 34, 35, 36
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness	3, 6
IoT	Internet of Things	29
ISA	Instruction Set Architecture	2, 3, 10, 12, 13, 15, 17, 18, 19, 20, 21, 24, 25, 29, 31, 37
JSON	JavaScript Object Notation	30
LWC	Lightweight Cryptography Competition	2, 3, 6, 30, 37
MAC	Message Authentication Code	4, 5
NIST	US National Institute of Standards and Technology	3, 30, 37, 40
RISC	Reduced Instruction Set Computing	10, 11, 12, 13
SPN	Substitution Permutation Network	9

Bibliography

[And19] Krste Asanovi Andrew Waterman. *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*. Version 20191213-Base-Ratified. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (visited on 09/18/2023).

[Aum17] Jean-Philippe Aumasson. *Serious Cryptography. a practical introduction to modern encryption*. eng. 1st edition. No Starch Press, 2017. ISBN: 1-5932-7826-8.

[BDPV07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Sponge functions". In: *ECRYPT hash workshop*. Vol. 2007. 9. 2007. URL: https://keccak.team/files/SpongeFunctions.pdf (visited on 09/23/2023).

[BDPV11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "On the security of the keyed sponge construction". In: *Symmetric Key Encryption Workshop*. Vol. 2011. 2011. URL: https://keccak.team/files/SpongeKeyed.pdf (visited on 09/23/2023).

[BDPV12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the sponge: single-pass authenticated encryption and other applications". In: *Selected Areas in Cryptography: 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers 18*. Springer. 2012, pp. 320–337. DOI: 10.1007/978-3-642-28496-0_19.

[Ber19a] D. J. Bernstein. *CAESAR Submissions*. 2019. URL: https://competitions.cr.yp.to/caesar-submissions.html (visited on 09/18/2023).

[Ber19b] D. J. Bernstein. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. 2019. URL: https://competitions.cr.yp.to/caesar.html (visited on 09/18/2023).

[Cad22] Cadence Design Systems, Inc. *Xtensa® Instruction Set Architecture (ISA) Summary*. 2022. URL: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf (visited on 09/23/2023).

[CJL+20] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoit Viguier. "Assembly or Optimized C for Lightweight Cryptography on RISC-V?" In: *International Conference on Cryptology and Network Security*. Springer. 2020, pp. 526–545. DOI: 10.1007/978-3-030-65411-5_26.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. 2021. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf (visited on 09/18/2023).

[DEMS23] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Reference, highly optimized, masked C and ASM implementations of Ascon*. 2023. URL: https://github.com/ascon/ascon-c (visited on 09/18/2023).

[Mar22] Ben Marshall. *RISC-V Cryptography Extensions Volume I. Scalar & Entropy Source Instructions*. Version 1.0.1. 2022. URL: https://github.com/riscv/riscv-crypto/releases/download/v1.0.1-scalar/riscv-crypto-spec-scalar-v1.0.1.pdf (visited on 09/18/2023).

[NIS23] NIST. *Lightweight Cryptography*. 2023. URL: https://csrc.nist.gov/Projects/lightweight-cryptography/ (visited on 09/18/2023).

[RIS21] RISC-V International. *RISC-V Bit-Manipulation ISA-extensions*. Version 1.0.0-38. 2021. URL: https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0-38-g865e7a7.pdf (visited on 09/18/2023).

[RPM22] Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok. "The Final Round: Benchmarking NIST LWC Ciphers on Microcontrollers". In: *Attacks and Defenses for the Internet-of-Things*. Ed. by Wenjuan Li, Steven Furnell, and Weizhi Meng. Cham: Springer Nature Switzerland, 2022, pp. 1–20. ISBN: 978-3-031-21311-3. DOI: 10.1007/978-3-031-21311-3_1.

[TMC+23] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Lawrence E Bassham, Jinkeon Kang, Noah D Waller, John M Kelsey, and Deukjo Hong. *Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process*. 2023. DOI: 10.6028/NIST.IR.8454.

[War13] Henry S Warren. *Hacker's Delight*. eng. 2nd edition. Addison-Wesley Professional, 2013. ISBN: 0-3218-4268-5.

[Wol21] Claire Wolf. *RISC-V Bitmanip Extension*. Version 0.93. 2021. URL: https://github.com/riscv/riscv-bitmanip/releases/download/v0.93/bitmanip-0.93.pdf (visited on 09/18/2023).