RISC-V Galois Field ISA Extension for Non-Binary Error-Correction Codes and Classical and Post-Quantum Cryptography

Yao-Ming Kuo, Francisco García-Herrero, Oscar Ruano, and Juan Antonio Maestro, *Senior Member, IEEE*

Abstract—Due to the recent advances in new communication standards, such as 5G New Radio and beyond 5G, and in quantum computing and communications, new requirements for integrating processors into nodes have appeared. These requirements are meant to provide flexibility in the network to reduce operational costs and support diversity in services and load balancing. They are also designed to integrate both new and classical algorithms into efficient and universal platforms, execute specific operations, and attend to tasks with lower latency. Furthermore, some cryptographic algorithms (classical and post-quantum), which are essential to portable devices, share the same arithmetic with error-correction codes. For example, Advanced Encryption Standard (AES), elliptic curve cryptography, Classic McEliece, Hamming Quasi-Cyclic, and Reed-Solomon codes use $GF(2^m)$ arithmetic. As this arithmetic is the basis of many algorithms, a versatile RISC-V Galois field ISA extension is proposed in this work. The RISC-V instruction set extension is implemented and validated using SweRV-EL2 1.3 on a Nexys A7 FPGA. In addition, a five-times acceleration is achieved for AES, Reed-Solomon codes, and Classic McEliece (post-quantum cryptography) at the expense of increasing the logic utilization by 1.27%.

Index Terms—RISC-V, ISA, galois field arithmetic, cryptography, error-correction codes

1 Introduction

In recent decades, with the popularization of small portable devices for applications such as the Internet of Things (IoT) [1], Industry 4.0, and CubeSats [2], and the emergence of communication network architectures like 5G New Radio (5GNR) [3], there have arisen more interconnected network nodes with minimal computing power for data acquisition. They are generally required to have low power consumption due to their battery life and small size, but low latency applications also need to be supported.

Traditionally, all information is sent to a central authority that process the data, as with cloud computing [4]. However, with the increase in portable devices, data processing is becoming inefficient due to the latency and saturation generated in the data center.

New methods of implementing the network's architecture have appeared to replace the current paradigm, the most popular of which is edge computing [5]. This methodology mainly consists of processing part of the device's information in either edge or fog nodes, alleviating the network's load. In this way, the throughput of the devices is improved far more than with cloud computing.

Furthermore, the mobile communication network tends to offer new services, such as artificial intelligence and holographic messages [6], that require data rates that 5G cannot provide. To reduce traffic, portable devices that formerly had a general-purpose processor must compute specific algorithms with very low latency (from 0.1 in previous standards to less than 1 ms or even hundreds of microseconds) and lower frame error rate (from 10^{-5} in 4G to 10^{-6} in 5GNR). Apart from this, the trend towards virtualizing traditional hardware functions [7] to reduce development and equipment costs has motivated GPU- and FPGA-based platforms to implement these solutions. In addition, the dynamic reconfigurability and in-field programming features of FPGAs [8] make them attractive for next generation communications, which rely on different standardized protocols and heterogeneous architectures.

To meet the aforementioned requirements, processors must be more efficient in executing specific operations and attending to tasks with lower latency. Since most portable devices have a general-purpose processor, it is not possible to run specific applications efficiently. Therefore, some solutions must be found to improve performance [9]. One option is the customization of the processor to identify the algorithms that are used most frequently in each case. For example, cryptography and error-correction codes are present in almost every portable device today, and many of these algorithms are based on finite field arithmetic [10]. Because of this, the acceleration of finite field arithmetic $GF(2^m)$ proceeds to have a significant role [11], encoding

- *Yao-Ming Kuo is with ARIES Research Center, Antonio de Nebrija University, 28040 Madrid, Spain. E-mail: ykuo@ieee.org.*
- *Francisco García-Herrero, Oscar Ruano, and Juan Antonio Maestro are with the Department of Computer Architecture and Automatics, Computer Science Faculty, Complutense University of Madrid, 28040 Madrid, Spain. E-mail: {francg18, oruano, jamaestro}@ucm.es.*

Manuscript received 3 Nov. 2021; revised 20 Apr. 2022; accepted 3 May 2022. Date of publication 12 May 2022; date of current version 10 Feb. 2023. (Corresponding author: Yao-Ming Kuo.) Recommended for acceptance by J. Hormigo. Digital Object Identifier no. 10.1109/TC.2022.3174587

and decoding in a communication channel and detecting errors in the transmitted data (BCH, Reed-Solomon (RS) codes [12]). It is also used in asymmetric cryptography, such as elliptical curve cryptography (ECC) [13], which is extensively used in the authentication process to exchange private keys or symmetric cryptography inside a secure communication channel as Advanced Encryption Standard (AES) [14].

Furthermore, post-quantum cryptography (PQC) algorithms [15] have been developed in recent years with the increase in research on quantum computing [16]. Some of the survivors of the third round of the National Institute of Standards and Technology's (NIST) PQC competition [17] use $GF(2^m)$ arithmetic (e.g., Classic McEliece [18], Rainbow [19], HQC [20], and GeMSS [21]). These algorithms require high computing power to generate the keys and either encrypt or decrypt the data, becoming the main bottleneck in processing for small devices, such as IoT end nodes and Cubesats.

Traditionally, there are three ways to optimize a processor. The first is to add coprocessors to the system to perform those specific operations. The second is to expand the base instructions of the computer architecture. The third and final method is to make a hybrid system [22] between the first and second methods, adding coprocessors and specific instructions depending on the case. We focus on the second solution, expanding the RISC-V base instruction set (RV32I) due to its readiness for integration within a core and its low area utilization.

This work proposes two approaches to a flexible instruction set for RV32 cores capable of accelerating any algorithm based on finite field arithmetic $GF(2^m)$, thus improving processor performance [23]. We do not focus on rigid approaches that can only accept a specific reduction polynomial. This work is oriented to portable devices, typically with a lightweight, general-purpose processor as a central processing unit (CPU) with a limited and basic instruction set. Additionally, our hardware architectures are designed to fit into the pipeline without deteriorating the critical path of the CPU. In this way, the maximum operating frequency is maintained, which leads to the comparison of clock cycles throughout the article.

A tradeoff between the base instruction set and specific custom instructions can be found for applications that require high flexibility and, at the same time, acceleration in the CPU execution time. For example, a portable device generally contains different error-correction codes and cryptographic standards or proprietary ciphers.

The rest of the paper is organized as follows. Section 2 describes the mathematical basis of $GF(2^m)$, and previous related works. Then, Section 3 details the contributions of this work. Afterwards, Section 4 introduces the instruction set extension. Section 5 shows the simulations and experimental results in a RISC-V System on Chip (SoC). Finally, Section 6 presents the conclusion of this work.

2 Background

This section is divided into three subsections. The first describes the fundamentals of finite field arithmetic, the second subsection details the previous related works, and the last enumerates the contributions of this work.

2.1 Finite Field Arithmetic

In this subsection, a quick review of the concepts related to Galois field operators is conducted (for more details, refer to [24]) while focusing on $GF(2^m)$. Our goal in this subsection is to explain the basics to the reader. In this way, the RISC-V ISA extension proposed in this article can be better understood.

This field is an extension of GF(2), where the constituent elements are zero and one. All finite fields have a unit element (α^0), a zero element ($\alpha^{-\infty}$), a primitive element (α), and at least one irreducible polynomial $p(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1x + p_0$. The primitive element α is the root of the irreducible polynomial and generates all the $GF(2^m)$ nonzero elements.

There are different ways to represent the elements in $GF(2^m)$; some of the approaches can be exponential, while others can be in polynomial form. In the exponential form, the parts are defined as powers of α; that is

$$GF(2^m) = \{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2m-2}\}. \tag{1}$$

Meanwhile, the polynomial representation has the following form:

$$P(\alpha) = a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0; \quad a_i \in GF(2), 0 \leq i \leq m-1. \tag{2}$$

Polynomial representation is beneficial for performing arithmetic operations. The definitions of addition and multiplication of finite fields are given below.

2.1.1 GF Addition

Consider two elements of $a(x)$ and $b(x)$ in (3), both belonging to the field $GF(2^m)$.

$$\begin{aligned} a(x) &= a_{m-1}x^{m-1} + \cdots + a_1x + a_0 \\ b(x) &= b_{m-1}x^{m-1} + \cdots + b_1x + b_0 \\ a_i \wedge b_i &\in GF(2), 0 \leq i \leq m-1. \end{aligned}$$

The sum $s(x)$ in (4) can be implemented as the *XOR* operation of each of its coefficients. Therefore, the result belongs to the same field

$$s(x) = (a_{m-1} \oplus b_{m-1})x^{m-1} + \cdots + (a_0 \oplus b_0). \tag{4}$$

2.1.2 GF Multiplication

There are different ways to multiply two polynomials in $GF(2^m)$. This paper focuses on two-step multiplication. As its name implies, this method separates multiplication into two steps: carry-less multiplication and polynomial reduction.

The first step is carry-less multiplication. The product $d(x)$ of the polynomials $a(x)$ and $b(x)$ is a polynomial of degree $2m-2$. This operation can be represented in matrix form as

$$\begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \\ d_m \\ d_{m+1} \\ \vdots \\ d_{2m-2} \end{pmatrix} = \begin{pmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{pmatrix}. \tag{5}$$

After the carry-less multiplication, the next step is polynomial reduction based on an irreducible polynomial $f(x)$. In modular reduction, $c(x) = d(x) mod\ f(x)$, whereby the degree of $d(x)$ is reduced by the degree of the irreducible polynomial $f(x)$, resulting in a degree less than $m-1$. The matrix form of the polynomial reduction is shown in (6)

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & r_{0,0} & \cdots & r_{0,m-2} \\ 0 & 1 & \cdots & 0 & r_{1,0} & \cdots & r_{1,m-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & r_{m-1,0} & \cdots & r_{m-1,m-2} \end{pmatrix} \begin{pmatrix} d_0 \\ \vdots \\ d_{m-1} \\ d_m \\ \vdots \\ d_{2m-2} \end{pmatrix}. \tag{6}$$

The matrix R in (6) depends exclusively on the irreducible polynomial $f(x)$. The coefficients r can be calculated as follows:

$$r_{j,i} = \begin{cases} f_j; j = 0, \ldots, m-1; i = 0 \\ r_{j-1,i-1} + r_{m-1,i-1}; j = 0, \ldots, m-1; i = 1, \ldots, m-2 \end{cases}. \tag{7}$$

2.1.3 GF Inversion

Different methodologies can be applied to calculate the $GF(2^m)$ inversion. For example, this operation can be precomputed and stored in a look-up table (LUT). However, this method would take a significant amount of silicon area to implement if m is not small. To address this, Zhang [25] proposed different schemes to compute this operation using multipliers, squares, and quartics. One of the schemes in Zhang's work is shown below.

For $\alpha \in GF(2^m)$, $\alpha^{-1} = \alpha^{2^m-2}$. The complexity of α^{2^m-2} computation can be reduced by sharing common terms

$$\alpha^{-1} = \alpha^{a^m-2} = \alpha^2 \cdot \alpha^{2^2} \cdots \alpha^{2^{(m-1)}}. \tag{8}$$

Here we can see that the inversion can be implemented by repeated squaring and multiplying, as illustrated in Fig. 1. This topology requires $m-1$ square operators and $m-2$ multipliers over $GF(2^m)$.

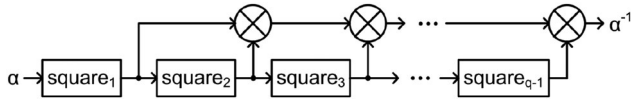

Fig. 1. Implementation architecture of inversion over $GF(2^m)$ [25].

2.2 Previous Related Works

RISC-V is a completely open ISA that is freely available to academia and industry [26]. In addition, the high availability of RISC-V solutions with permissive licenses makes this architecture attractive for any implementation. Moreover, the main principle of RISC-V ISA design is keeping the instruction set simple, micro-architecture agnostic, and technology independent. Therefore, a RISC-V ISA extension is proposed in this manuscript, and related works by other authors are presented in this subsection.

The RISC-V community has proposed a scalar cryptographic extension [27] that accelerates various cryptographic algorithms, such as AES [28], SHA-256, SHA-512, SM3, and SM4. Although it achieves considerable acceleration, this proposal does not contemplate PQC algorithms or error-correction codes. As a result, the instruction set is not flexible and cannot accelerate other algorithms or proprietary ciphers.

On the other hand, Stoffelen [29] proposed different methods to optimize the RISC-V assembly code from AES, ChaCha, and Keccak. Additionally, the current study analyzed the possibility of adding an ISA extension to improve these cryptographic algorithms.

Regarding the acceleration of PQC algorithms in RISC-V, different authors have proposed architectures or coprocessors [30], [31] and acceleration by extending the instruction set [32].

Finally, a finite field $GF(q)$ ISA extension was proposed by Alkim [33] to accelerate lattice-based PQC cryptography (Kyber, NewHope). The same can be performed for the $GF(2^m)$ fields to accelerate code-based PQC, error-correction codes, and basic operations of classical cryptography.

3 Contributions

According to the new challenges in network requirements and applications mentioned in Section 1, a wide range of algorithms uses $GF(2^m)$ arithmetic. For instance, non-binary error-correction codes are employed in communication, cryptography is used to encrypt and decrypt incoming data, and proprietary ciphers are based on this arithmetic. Therefore, the contribution of this work is the definition of a tiny $GF(2^m)$ ISA extension in RISC-V architecture, in which the logic utilization and performance are essential inputs to consider in relation to the direct impact in classical and post-quantum cryptography and non-binary error-correction codes. Furthermore, this work is intended for lightweight, general-purpose processors whose area and power consumption requirements limit the implementation of a coprocessor within the system.

The RISC-V ISA extension proposed in this work is a solution between the RISC-V base ISA and the scalar cryptographic K extension [27], resulting in an intermediate performance between the two. However, it is more flexible in terms of the protocols that it can process because the K extension implements dedicated hardware for each particular algorithm. It should also be noted that the cryptography K extension does not support post-quantum algorithms or non-binary error-correction codes.

Furthermore, our ISA extension is a complement to Alkim's GF(q) ISA extension [33] since some algorithms use $GF(q)$ arithmetic (i.e., Kyber, NewHope) and others use $GF(2^m)$. We have identified that cryptography and error-

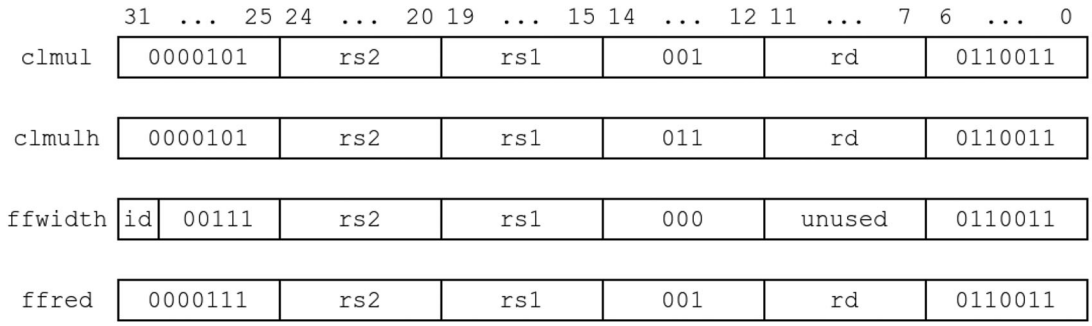

Fig. 2. The instruction format for the variant 1.

correction codes share the same operations, such as bit manipulation (i.e., rotations, permutations, and carry-less multiply) and finite field arithmetic. These operations can be defined in the instruction set in order to accelerate a wider range of algorithms. Therefore, we explore in this article an extension of the instruction set to accelerate algorithms based on $GF(2^m)$ arithmetic.

In summary, the proposed ISA extension is capable of processing the following algorithms:

- Error-correction codes (e.g., non-binary LDPC, BCH, RS codes, etc.)
- Pre-quantum cryptography (e.g., AES, ECC, etc.)
- PQC (e.g., McEliece, HQC, etc.)
- Proprietary ciphers based on $GF(2^m)$ arithmetic

The maximum finite field size that can be computed in a single instruction is limited by the bit width of the processor's architecture. In this case, the ISA extension can support field sizes of up to 32 since the width of RV32I general-purpose registers is 32. However, flexible $GF(2^m)$ hardware architectures like the one described in [34] can be found in order to execute various primitive polynomials using the same logic.

On the other hand, it is possible to calculate larger Galois fields by using composite fields [35], thereby separating them into smaller fields. Additionally, although PQC Rainbow [19] uses tower-field representation, it is possible to map into an isomorphic field and increase computation speed [36].

In brief, the proposed ISA extension supports any polynomial size. When the polynomial degree is higher than the bit width of the general-purpose registers (32 bits in the case of RV32I), it is possible to split the calculation into smaller fields and calculate using composite fields [35].

4 Proposed ISA Extension

This section describes two approaches of the ISA extension proposed in this work for RISC-V processors.

The first variant is for lightweight, general-purpose processors since these processors generally have a limited instruction set and a small footprint. Therefore, the goal is to reuse existing instructions and add only what is necessary. Additionally, the hardware should be flexible enough to accept any primitive polynomial.

The second variant is oriented towards applications where the area utilization is not a critical requirement. In this way, fast and dedicated hardware architectures can be used to maximize the processor's operating frequency.

4.1 Variant 1 - Flexible and Small Footprint

The operation added in this approach is the multiplication of finite fields since the addition of two numbers in $GF(2^m)$ is only an *XOR* operation and is already defined in RV32I. The multiplication is performed in three different instructions (See Section 2.1.2): carry-less multiplication (*CLMULH* and *CLMUL*) and polynomial reduction (*FFRED*). The carry-less multiplication requires two instructions since multiplying two 32-bit numbers results in a 64-bit number, and the size of the RV32 registers is 32 bits.

To correctly multiply two numbers in $GF(2^m)$, it is also necessary to indicate the irreducible polynomial and the degree to the processor. Therefore, additional instruction is required to pass these parameters (*FFWIDTH*). These parameters are stored in two internal registers at the input of the reduction module. We keep these two values in an internal register of the ALU since four source registers are needed to perform finite field operations. At the same time, the RISC-V instruction format only allows for two source registers (*rs1* and *rs2*). The block diagram is shown in Fig. 3.

In some processors, such as SweRV-EL2, carry-less multiplication is already implemented as an extension. Hence, the opcode is kept for a compatibility and resource-sharing issue. Moreover, as we can see here, this instruction set requires flexible multiplier architectures since the hardware must accept any irreducible polynomial. Thus, for example, the architecture proposed in [34] could fit in this case.

We decided not to implement the square function and the inverse of $GF(2^m)$ in this variant by means of adding extra hardware, as they are modules that require a significant degree of logic and can be calculated using finite field multiplication. In more specific computers, these two instructions can be added to improve the system's efficiency at the expense of increasing logic utilization, which is the second variant of this work.

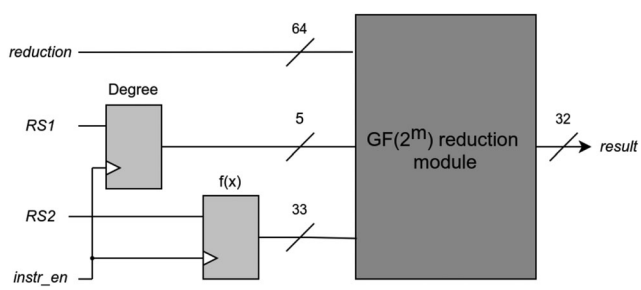

Fig. 3. FFWIDTH internal registers.

TABLE 1
Clock Cycles Required to Compute the Inversion (SweRV-EL2 v1.3)

GF inversion	$GF(2^4)$	$GF(2^8)$	$GF(2^{16})$
SweRV-EL2 [37]	352	1397	6801
variant 1	74	172	384
Reduc. %	78.98%	87.69%	94.35%

Instead, for example, the inverse can be calculated using Zhang's topology [25] described in Section 2.1. As shown in Table 1, the number of clock cycles required to execute the RISC-V standard instruction set was compared to that of the custom instructions proposed by us. It can be seen that a significant reduction of 94.35% in the number of cycles for the $GF(2^{16})$ field was reached. The implementation setup is detailed in Section 5, and the resource utilization is shown in Table 7.

Fig. 2 shows the formats of the custom instructions. As we can see here, the carry-less multiplication keeps the RISC-V extension B format.

The parameters that receive and return the instructions are:

- *FFWIDTH*: In *rs1*, it receives the degree of the polynomials, and in *rs2*, it receives the irreducible polynomial. These two parameters are stored in internal parameters, determined by the field *id*, which is helpful when different tasks run in the core, making a context switch. Additionally, since RISC-V registers are 32 bits, the most significant bit of the irreducible polynomial is assumed to be one when the degree of the input polynomials is 32.
- *FFRED*: It receives the polynomial to be reduced as a parameter. In *rs1*, it receives the high part, and in *rs2*, the low part of the polynomial. This instruction returns the reduced polynomial $c(x)$.
- *CLMULH & CLMUL*: The parameters are the same as extension B.

An example of the multiplication of finite fields using the custom instructions is shown in Fig. 4. This example is a part of the AES code, which employs the following reducing polynomial for multiplication

$$f(x) = x^8 + x^4 + x^3 + x + 1. \tag{9}$$

This primitive polynomial belongs to $GF(2^8)$. Therefore, the degree is eight, and the coefficients can be represented in an 8-bit binary form, each bit representing one of the

```
li      a5, 8       % Polynomial degree
li      a4,283      % Primitive poly
ffwidth a5,a5,a4
...
lbu     a4,0(a0)    % Operand A
lbu     a7,0(a1)    % Operand B
clmul   a4,a4,a7    % CL multiplication
li      a5,0
ffred   a7,a5,a4    % Reduction
```

Fig. 4. GF multiplication for AES (variant 1).

primitive polynomial coefficients. As seen in (10), the coefficients can also be expressed in hexadecimal or decimal form

$$100011011_2 = 11B_{16} = 283_{10}. \tag{10}$$

The number 283 is the value loaded into the second instruction of Fig. 4, representing the primitive polynomial in base 10.

The *FFWIDTH* instruction appears only for the first time to tell the hardware the degree and the irreducible polynomial, which are both stored in internal registers. Additionally, this instruction receives a value in the result in register *a5* since the format of *FFWIDTH* corresponds to an operation instruction in RISC-V. This instruction is not used in any other part of the code.

The register *rs1* passed to *FFRED* is zero because AES uses polynomials that belong to $GF(2^8)$, and the bits 63-32 will always be zero after carry-less multiplication. In this case, the compiler used the instruction *LI*, assigning a logical zero to *rs1* instead of using *CLMULH* (see Fig. 4).

4.2 Variant 2 - Traditional Definition

The instructions added to this approach are shown in Fig. 5. The main difference from the first approximation is that we have instructions for each operation and primitive polynomial in $GF(2^m)$ arithmetic.

Additionally, the ISA is more rigid because it has an *idx* field that selects the appropriate finite field hardware already implemented in the core. For example, the $GF(2^m)$ multipliers that could fit this instruction set are the architectures proposed by Mastrovito [38], as both are swift but at the expense of area utilization.

It can be seen in Fig. 5 that the instructions receive the operands in *rs1* and *rs2*, and the result is returned in *rd*. Finally, the field *idx* selects the corresponding hardware. The details of each instruction are listed below:

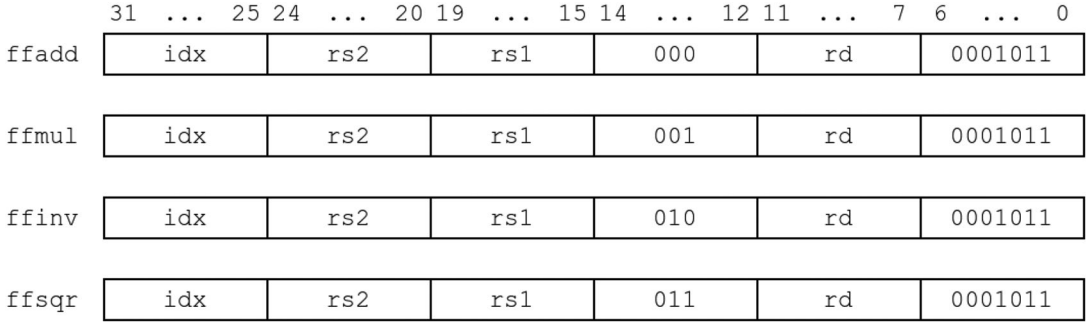

Fig. 5. The instruction format for the variant 2.

```
lbu     a4,0(a0)    % Operand A
lbu     a7,0(a1)    % Operand B
ffmul   a4,a4,a7    % GF multiplication
```

Fig. 6. GF multiplication for AES (variant 2).

- *FFADD*: This instruction performs the addition of $GF(2^m)$. It receives the operands in *rs1* and *rs2*, and the result is returned in *rd*. The field *idx* selects the hardware implementation for each particular primitive polynomial.
- *FFMUL*: This instruction performs the multiplication of $GF(2^m)$. It receives the operands in *rs1* and *rs2*, and the result is returned in *rd*. The field *idx* has the same function described before.
- *FFINV*: This instruction performs the inversion of $GF(2^m)$. It receives the operands in *rs1* and *rs2*, and the result is returned in *rd*. The field *idx* has the same function described before.
- *FFSQR*: This instruction performs the squares of $GF(2^m)$. It receives the operands in *rs1* and *rs2*, and the result is returned in *rd*. The field *idx* has the same function described before.

Following the example of the multiplication of finite fields presented in the previous subsection, the code is shown in Fig. 6. As seen in the figure, the number of instructions required to perform a GF operation is lower than in variant one.

5 Simulation and Implementation Results

The custom instructions are implemented and validated with Verilator v4.032 using the SweRV-EL2 v1.3 core, with the extension Zbc [37] enabled. This extension includes the built-in carry-less multiplier.

The first step is adding the logic in the decoding stage to recognize the opcode of the custom instructions. SweRV-EL2 has a dedicated script [37] written in Perl, which automatically generates the additional logic required in the decoding stage based on the custom ISA formats described in a text file using the Espresso logic minimizer [39]. The text file must be carefully modified according to the custom ISA format; otherwise, the logic generated by the script is wrong, and the core will not be able to recognize the added instructions.

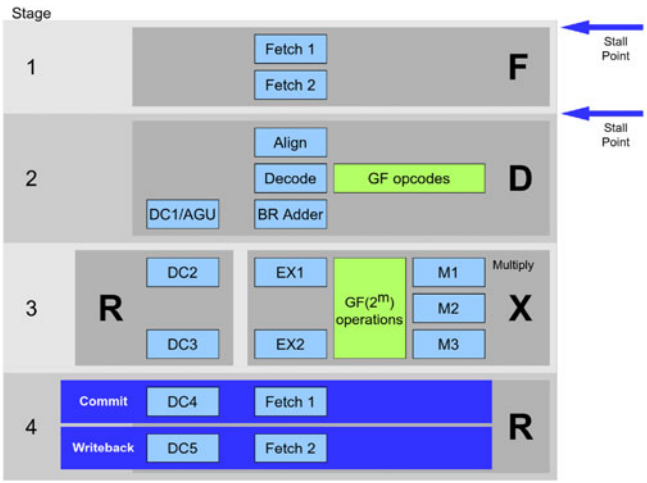

Fig. 7. SweRV EL2 Core Pipeline [37].

```
#define CUSTOM_CODES

static uint8_t Multiply(uint8_t x, uint8_t y)
{
#ifdef CUSTOM_CODES
  uint32_t imm_result,result;
  asm volatile
            (
                "clmul   %[z], %[x], %[y]\n\t"
                : [z] "=r" ((uint32_t)imm_result)
                : [x] "r" ((uint32_t)x), [y] "r" ((uint32_t)y)
            );
  asm volatile
            (
                "ffred   %[z], %[x], %[y]\n\t"
                : [z] "=r" ((uint32_t)result)
                : [x] "r" ((uint32_t)0), [y] "r" ((uint32_t)imm_result)
            );
  return (uint8_t) result;
#else
  uint8_t p = 0, i = 0, hbs = 0;
    for (i = 0; i < 8; i++) {
        if (y & 1) {
            p ^= x;
        }
        hbs = x & 0x80;
        x <<= 1;
        if (hbs) x ^= 0x1b;
        y >>= 1;
    }
    return (uint8_t)p;
#endif
}
```

Fig. 8. GF multiplication example using inline assembly approach (AES).

Then, the corresponding logic is added in the execution stage. Since carry-less multiplication and binary multiplication share the same module within the core, the polynomial reduction module is also implemented in the same block. The block diagram of the SweRV-EL2 core is shown in Fig. 7.

Once the logic is implemented, the assembly module (binutils) is modified. Thus, the toolchain can recognize the opcodes of the custom instructions using the inline assembly approach, by which the GF operators written in C are replaced with their corresponding instructions in assembly language. Consequently, it is possible to run tests of different algorithms using the C language and compare the efficiency among the RV32IMC and the custom ISA extension. For example, the GF multiplication for AES using this approach is shown in Fig. 8. It can be seen here that the assembly code is inserted in the C function, and the GF multiplication is performed using two instructions (clmul and ffred) for variant one. The testbenches used in this work are included in [40] under the same structure as [37]. It should be noted that the C codes of each algorithm are based on [41], [42], and [43].

In this work, a performance evaluation for AES, Reed-Solomon codes, and McEliece348864 was conducted. The simplified diagram of the design flow meant to integrate a custom instruction set is shown in Fig. 9. It should be noted that the topology of the $GF(2^m)$ hardware architecture is not relevant in the evaluation since we are estimating the number of clock cycles required for each ISA extension. Additionally, our hardware architectures are designed to fit into the pipeline without deteriorating the critical path of the CPU. In this way, the maximum operating frequency is maintained.

The number of instructions of each proposed ISA extension can be directly determined by simulation, where the configuration details are explained in Section 5.1. On the other hand, the logic utilization and timing are described in Section 5.5.

5.1 Simulation Setup

The SweRV-EL2 core has four predefined configurations that can be selected when the CPU model is built: default,

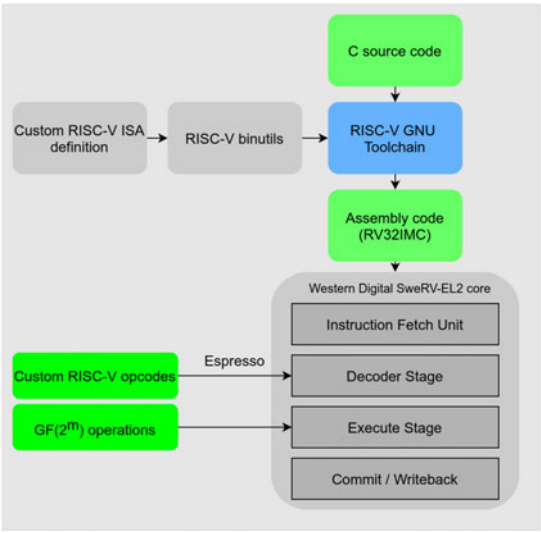

Fig. 9. Custom ISA design flow and validation.

default_ahb, typical_pd and high_perf. The details are shown in Table 3. Depending on these configurations, the core could be used for low-power or high-performance computing, resulting in more logic utilization. In this work, we used the typical_pd configuration to create a power-optimized CPU.

At this stage, to compare the clock cycles for different algorithms, we built a Verilator model consisting of a SweRV core with an AXI bus connected with the virtual memory. Then, the algorithms in the C language were compiled using RISC-V GNU toolchain, and the binary was generated for each test.

Finally, the binary was pre-loaded in this memory before the simulation started. All these setup guidelines can be found in the SweRV Github repository [37], and the modified SweRV core is publicly available [40] since it is a fork of the original repository. In addition, there are various git branches in the modified repository, which contain different versions of the core:

- *variant1*: SweRV-EL2 variant 1 core with testbenches
- *variant2*: SweRV-EL2 variant 2 core with testbenches
- *aes*: SweRV-EL2 aes [28] core with testbenches

To use a specific version, clone the forked repository [40], and change to the corresponding branch. The folder structures of the original repository [37] are maintained; therefore, all the required documentation regarding setup and simulation can be found there.

Then, the simplified simulation block diagram is shown in Fig. 10. As a result, we can directly determine the performance of the custom ISA by simulation.

5.2 Classical Cryptography: AES

The C code was generated for the different key sizes (AES128, AES192, and AES256) and encryption schemes (CBC, CTR, and ECB), based on the code of TinyAES [41] and using the example test vectors defined by NIST in the "Special Publication 800-38A 2001 ED" [44]. This publication provides recommendations regarding modes of operation to be used with symmetric key block cipher algorithms. In Appendix F, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2022.3174587, of this report, three examples and test vectors are provided for each of the modes for AES.

Next, two other versions were created with the custom instructions replacing the code segments where the GF arithmetic appears. The first version used variant one, and the second version used variant two.

Moreover, another version was created using the RISC-V ISA extension proposed in [28]. A natural conclusion of this comparison is that this instruction set is better than our approach to accelerating AES since it has exclusive hardware and does only one thing. In contrast, the approaches described in this manuscript are more generic and can improve the performance of $GF(2^m)$ arithmetic-based algorithms, such as RS codes, Classic McEliece, HQC, etc.

Then, they were compiled with the following flags:
-O3 -fomit-frame-pointer -fPIC -no-pie

Table 2 shows the number of clock cycles required for the base and custom instructions for each encryption method.

For example, for AES256, the clock cycles needed for CBC encryption was 285 245, while using variant one only needed 53 396. Therefore, it can be seen that for variant one, a significant reduction of 81.28% can be reached in this mode, while using variant two required 48 763 clock cycles, which translates into a decrease of 82.90%. Thus, variant two is slightly better than the first one; while more than one instruction is necessary to perform the GF multiplication in variant one (*CLMUL*, *CLMULH*, and *FFRED*), variant two only uses one (*FFMUL*). Additionally, variant two has the GF inversion included in the hardware, whereas the operator must be executed by software in variant one.

Then, Table 4 compares the number of clock cycles in variant one, variant two, and the work in [28]. Comparing our work with Marshall's proposal, it is clear that his work achieved a more significant reduction in clock cycles for all the cases since the instruction set was only for processing AES and had dedicated hardware for this particular algorithm. For instance, the number of instructions for AES CBC encryption is reduced by 81.28% for variant 1, 82.90% for variant 2, and 97.76% for [28]; in other words, the ISA extension proposed in [28] is reduced by 16.48% compared to variant 1, and 14.86% compared to variant 2. Instead, our proposals are universal and can improve the performance of any $GF(2^m)$ arithmetic-based algorithms, not only AES.

Finally, it was observed that the code size reduction was greater than 35% for all cases (variant one and two).

5.3 Non-Binary Error-Correction Code: Reed-Solomon

The same procedure was conducted for the Reed-Solomon code performance comparison, based on the code written by Minsky [42] and using its test benches. A program in C code was created with the encoding and decoding routine for RS(255,247) and RS(255,239). Then, another two versions were created with the custom instructions (variants one and two), replacing the multiplication of finite fields. The same SweRV-EL2 configurations for AES were kept and compiled with the same GCC flags. Table 5 shows the number of clock

TABLE 2
Number of Clock Cycles Required Under the Same Maximum Clock Frequency for AES (RV32IMC versus Custom)

AES128	CBC Encryption	CBC Decryption	CTR Encryption	CTR Decryption	ECB Encryption	ECB Decryption
SweRV-EL2 [37]	197,920	198,240	198,208	198,197	50,641	50,726
Variant 1	38,328	39,303	39,033	38,995	10,854	11,011
Reduc. %	80.63%	80.17%	80.31%	80.33%	78.57%	78.29%
Variant 2	35,136	36,431	35,853	35,818	10,049	10,291
Reduc. %	82.25%	81.62%	81.91%	81.93%	80.16%	79.71%
AES192	CBC Encryption	CBC Decryption	CTR Encryption	CTR Decryption	ECB Encryption	ECB Decryption
SweRV-EL2 [37]	242,573	242,695	242,839	242,828	62,572	62,661
Variant 1	47,016	48,019	47,637	47,617	13,764	13,939
Reduc. %	80.62%	80.21%	80.38%	80.39%	78.00%	77.75%
Variant 2	43,013	44,530	43,738	43,719	12,772	13,055
Reduc. %	82.27%	81.65%	81.99%	82.00%	79.59%	79.17%
AES256	CBC Encryption	CBC Decryption	CTR Encryption	CTR Decryption	ECB Encryption	ECB Decryption
SweRV-EL2 [37]	285,245	285,593	285,439	285,425	72,331	72,416
Variant 1	53,396	54,548	54,054	54,036	14,462	14,660
Reduc. %	81.28%	80.90%	81.06%	81.07%	80.01%	79.76%
Variant 2	48,763	50,394	49,444	49,431	13,303	13,631
Reduc. %	82.90%	82.35%	82.68%	82.68%	81.61%	81.18%

cycles required to encode and decode the RS(255,247) and RS(255,239) codes.

For example, 151 681 clock cycles were needed for RS (255,247) decryption using the RISC-V base instructions, while only 22 648 cycles were required for variant one and 21 220 for variant two. In other words, a reduction of 85.07% can be achieved with variant one and 86.02% with variant two. As is the case in AES, variant two is slightly better because it uses only one instruction for the $GF(2^m)$ multiplication.

TABLE 3
SweRV-EL2 Predefined Target Configurations [37]

Target	Description
default	Default configuration. AXI4 bus interface
default_ahb	Default configuration, AHB-Lite bus interface
typical_pd	No ICCM, AXI4 bus interface
high_perf	Large BTB/BHT, AXI4 interface

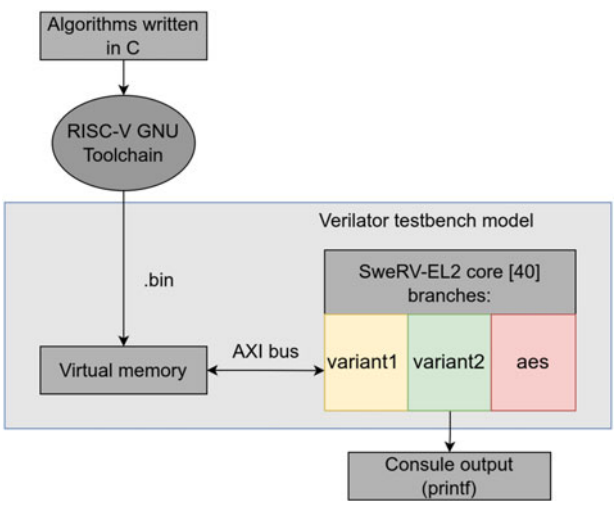

Fig. 10. Simulation setup block diagram.

5.4 Post-Quantum Cryptography: Classic McEliece

The same methodology was used to estimate the performance of the Classic McEliece post-quantum algorithm, using the official source code submitted to NIST [43]. McEliece348864 was evaluated in this work since this set of parameters results in Classic McEliece's lightest algorithm. Additionally, as the generation of public and private keys requires many clock cycles and exceeds the computing power of this core, the keys were pre-computed and stored in the memory of the CPU.

Table 6 shows the number of cycles required to encrypt and decrypt this post-quantum algorithm. Using variant one reduced the encryption by 0.57% and the decryption by 72.06% since the Classic McEliece encryption does not use $GF(2^m)$ multiplication. Hence, there is no performance improvement in encryption.

Nevertheless, the number of clock cycles needed for decryption (176 M cycles) was much more significant than for encryption (4.7 M cycles), making the reduction in decryption more important for this algorithm.

Then, it can be seen that variant two is better than the first approach, as it required 44 M clock cycles instead of 49 M. In other words, a reduction of 72.06% was achieved for variant one and 74.92% for variant two.

Moreover, it should be noted that if only [28] is implemented in the core and we need to process another $GF(2^m)$

TABLE 4
Number of Clock Cycles Required for AES256-CBC

AES256	CBC Encryption	CBC Decryption
SweRV-EL2 [37]	285,245	285,593
Variant 1	53,396	54,548
Reduc. %	81.28%	80.90%
Variant 2	48,763	50,394
Reduc. %	82.90%	82.35%
AES ISA [28]	6,390	7,739
Reduc. %	97.76%	97.29%

Comparison between RV32I, Variant 1, Variant 2, and Ref. [28].

TABLE 5
Number of Clock Cycles Required Under the Same Maximum Clock Frequency for RS Codes

	RS(255,247)		RS(255,239)	
	Encode	Decode	Encode	Decode
SweRV-EL2 [37]	154,003	151,681	300,831	303,289
Variant 1	29,006	22,648	58,660	45,237
Reduc. %	81.17%	85.07%	80.50%	85.08%
Variant 2	27,586	21,220	56,332	42,336
Reduc. %	82.09%	86.02%	81.27%	86.04%

TABLE 6
Number of Clock Cycles Required Under the Same Maximum Clock Frequency for McEliece 348864

McEliece 348864	Encryption	Decryption
SweRV-EL2 [37]	4,710,737	176,180,492
Variant 1	4,684,084	49,222,882
Reduc. %	0.57%	72.06%
Variant 2	4,331,765	44,181,361
Reduc. %	8.04%	74.92%

arithmetic, there is no additional speed up for other algorithms compared to variants one and two. For example, a [28] modified core can accelerate AES significantly, but it requires 176 M clock cycles to execute the Classic McEliece decryption algorithm. Meanwhile, variant one only needs 49 M, and variant two needs 44 M.

Finally, to the best knowledge of the authors, all of the PQC results in RISC-V are for accelerating $GF(q)$-based PQC [33] (e.g., Kyber, NewHope), not $GF(2^m)$-based PQC. Therefore, the comparison cannot be made since they use different arithmetics. On the other hand, the methods employed to speed up $GF(q)$ do not provide any advantage to the algorithms based on $GF(2^m)$.

5.5 Logic Utilization and Timing

In order to implement the designs in an FPGA (Nexys A7), the SweRV-EL2 core was integrated into SweRVolf [45] SoC, which consists of the SweRV CPU with a boot ROM, AXI4 interconnect, UART, SPI, RISC-V timer, and GPIO.

Different SoC versions were created using Vivado: one with the RISC-V base instructions, and the others with variant one, variant two, and AES ISA [28] with the extension Zbc [37] enabled. This extension includes the built-in carry-less multiplier.

For variant one, the polynomial reduction (*FFRED*) was implemented based on the topology presented in [34] since it is a flexible polynomial architecture. Additionally, the carry-less multiplication (*CLMULH* and *CLMUL*) was not necessary to implement because it was already implemented in SweRV-EL2 [37].

For variant two, the operations were implemented for each algorithm (i.e., AES, Reed-Solomon codes, and McEliece) based on fast but rigid architectures [38]. In this way, we implemented three multipliers, inversion, and squares for these algorithms since they have different primitive polynomials and degrees. Moreover, the inversion was precomputed and stored in an LUT for each particular primitive polynomial.

For the AES ISA [28] integration, the operations were implemented using the Verilog RTL provided by the authors [28].

Then, they were implemented onto a Nexy A7 FPGA device, the results of which were then extracted. Table 7 shows the logic utilization using the standard instruction set (RVIMC), variant one, variant two, and AES ISA [28]. It can be seen that the SweRV-EL2 core uses 26 375 LUTs, while variant one uses 26 710 and variant two uses 26 957. Thus, we have only a 1.27% increment in logic utilization for variant one and 2.21% for variant two. As we can see here, the logic increment was relatively low compared to the number of clock cycles that we can reduce by using these custom instructions. For example, we decreased the number of clock cycles for variant one by around 80% at the expense of an increment of 1.27% in logic utilization.

Comparing the hardware increment in the execute stage instead of comparing the logic utilization of the whole system demonstrates that the SweRV-EL2 requires 3,832 LUTs, while variant one uses 4,092 and variant two uses 4,407. This results in an increment of 6.78% for variant one and

TABLE 7
Logic Utilization for SweRV-EL2 Core

SweRV-EL2 core (Freq. = 25 MHz)	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	DSP
SweRV-EL2 [37]	26,375	9,051	486	73	4
Variant 1	26,710	9,084	478	127	4
Inc. %	1.27%	0.36%	-1.65%	73.97%	0%
Variant 2	26,957	9,046	415	127	4
Inc. %	2.21%	-0.06%	-14.61%	73.97%	0%
AES ISA [28]	26,721	9,051	486	73	4
Inc. %	1.31%	0%	0%	0%	0%
SweRV-EL2 Execute stage (Freq. = 25 MHz)	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	DSP
SweRV-EL2 [37]	3,832	516	0	0	4
Variant 1	4,092	554	63	0	4
Inc. %	6.78%	7.36%	-	-	0%
Variant 2	4,407	516	0	0	4
Inc. %	15.01%	0%	-	-	0%
AES ISA [28]	4,120	516	0	0	4
Inc. %	7.52%	0%	-	-	0%

15.01% for variant two in terms of logic utilization. However, these numbers are still insignificant compared to the acceleration achieved using the custom instructions.

Additionally, it can be seen that variant two occupies more area than variant one. This gap widens when more algorithms based on $GF(2^m)$ arithmetic must be processed in the core. Therefore, a variant is chosen depending on how many algorithms a particular application must process.

Finally, the logic utilization presented in this work (i.e., in variants one and two) and the ISA proposed in [28] were compared. Table 7 demonstrates that variant one requires 26 710 LUTs, variant two requires 26 957, and [28] requires 26 721. In other words, the logic utilization is incremented by 1.27% for variant one, 2.21% for variant two, and 1.31% for [28]. Thus, by incrementing the logic utilization in the same proportion, it is possible to speed up more algorithms using the ISA extensions proposed in this work. At the same time, [28] only accelerates AES. In other words, if the acceleration of another $GF(2^m)$-based algorithm is also required in the same core, an additional hardware accelerator must be implemented, resulting in more logic utilization compared to variants one and two.

Regarding the clock frequency, the implementations work at 25 MHz, which is the limit for the SweRV-EL2 core running on the Nexys A7 FPGA. There was no decrease in frequency due to the addition of the extra logic for the custom instructions.

6 Conclusion

New challenges in technical specifications for the nodes that constitute future communication networks (e.g., better frame error rate, lower latency, flexibility, etc.) and the appearance of new security algorithms derived from the rise of quantum computing have led to an interest in increasing the efficiency of general-purpose processors. One of the approaches to solving this problem is adding specific instructions to execute particular applications, such as the use of non-binary error-correction decoders to improve the frame error rate compared to previous standards or cryptography to secure a communication channel in the presence of quantum processors. These algorithms become the main bottleneck in data processing for general-purpose processors, as they generally require high computing power (i.e., the number of CPU clock cycles) to generate their public and private keys.

Finite field $GF(2^m)$ arithmetic has been identified to be used in these applications, such as pre- and post-quantum cryptography and error-correction codes. Therefore, this work proposed an extension of the instructions and was oriented towards applications where it is necessary to process different protocols that use finite field arithmetic.

From the previous section, it can be seen that a significant reduction in the number of clock cycles of more than 77.75% was achieved for AES, 80.50% for Reed-Solomon codes, and 72.06% for Classic McEliece348864 using variant one. Moreover, the performance can be improved by using variant two. Otherwise, the logic utilization was increased by only 1.27% for all the encryption schemes and algorithms.

To summarize, variant two is superior in accelerating algorithms based on $GF(2^m)$ arithmetic, but the logical utilization is slightly higher than variant one. Variant one can reuse CPU carry-less multiplication *(CLMUL)* to compute the first part of $GF(2^m)$ multiplication and only implement the primitive polynomial reduction module. However, since both variants have similar performance, it is possible to choose either of the two options depending on the processor's architecture to which including this improvement is desirable.

Acknowledgments

Yao Ming Kuo has developed his work in this project sponsored by Banco Santander in the frame of the program "Contratos Predoctorales de Formación" of Nebrija University.

References

[1] L. Tan and N. Wang, "Future internet: The Internet of Things," in *Proc. 3rd Int. Conf. Adv. Comput. Theory Eng.*, 2010, pp. V5–376–V5–80.

[2] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka, and R. Twiggs, "CubeSat: A new generation of picosatellite for education and industry low-cost space experimentation," in *Proc. Small Satell. Conf.*, 2000, pp. 1–19. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2000/All2000/32/

[3] T. Huang, W. Yang, J. Wu, J. Ma, X. Zhang, and D. Zhang, "A survey on green 6G network: Architecture and technologies," *IEEE Access*, vol. 7, pp. 175 758–175 768, 2019.

[4] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, Gaithersburg, MD, USA: Special Publication 800-145, NIST, 2011, pp. 1–7. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[6] E. Calvanese Strinati *et al.*, "6G: The next frontier: From holographic messaging to artificial intelligence using subterahertz and visible light communication," *IEEE Veh. Technol. Mag.*, vol. 14, no. 3, pp. 42–50, Sep. 2019.

[7] C. Tarver, M. Tonnemacher, H. Chen, J. Zhang, and J. R. Cavallaro, "GPU-Based, LDPC decoding for 5G and beyond," *IEEE Open J. Circuits Syst.*, vol. 2, pp. 278–290, 2021.

[8] V. Chamola, S. Patra, N. Kumar, and M. Guizani, "FPGA for 5G: Re-configurable hardware for next generation communication," *IEEE Wireless Commun.*, vol. 27, no. 3, pp. 140–147, Jun. 2020.

[9] J. M. McGinthy and A. J. Michaels, "Lightweight Internet of Things encryption using Galois extension field arithmetic," in *Proc. IEEE Int. Conf. Internet Things IEEE Green Comput. Commun. IEEE Cyber Phys. Social Comput. IEEE Smart Data*, 2018, pp. 74–80.

[10] W.-M. Lim and M. Benaissa, "Design space exploration of a hardware-software co-designed $GF(2^m)$ Galois field processor for forward error correction and cryptography," in *Proc. 1st IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synth.*, 2003, pp. 53–58.

[11] Y. Chen, *et al.*, "A programmable Galois field processor for the internet of things," in *Proc. IEEE/ACM 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 55–68.

[12] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. Hoboken, NJ, USA: Wiley, 1999.

[13] L. C. Washington, *Elliptic Curves: Number Theory and Cryptography*. Boca Raton, FL, USA: CRC, 2008.

[14] S. Heron, "Advanced encryption standard (AES)," *Netw. Secur.*, vol. 2009, no. 12, pp. 8–12, 2009.

[15] A. Khalid, S. McCarthy, M. O'Neill, and W. Liu, "Lattice-based cryptography for IoT in a quantum world: Are we ready?," in *Proc. IEEE 8th Int. Workshop Adv. Sensors Interfaces*, 2019, pp. 194–199.

[16] L. Gyongyosi and S. Imre, "A survey on quantum computing technology," *Comput. Sci. Rev.*, vol. 31, pp. 51–71, 2019.

[17] D. Moody, "Post-quantum cryptography: NIST's plan for the future," in *Proc. 7th Int. Conf. Post-Quntum Cryptogr.*, 2016, pp. 1–16.

[18] D. J. Bernstein *et al.*, "Classic McEliece: Conservative code-based cryptography," *PQCRYPTO Mini-Sch. Workshop*, pp. 1–25, 2018. [Online]. Available: https://troll.iis.sinica.edu.tw/school+forum18/slides/McEliece.pdf

[19] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2005, pp. 164–175.

[20] C. A. Melchor *et al.*, "Hamming quasi-cyclic (HQC)," *NIST PQC Round*, vol. 2, pp. 4–13, 2018.

[21] A. Casanova, J.-C. Faugere, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem, "GeMSS: A great multivariate short signature," Tech. Rep., 2017, pp. 1–4. [Online]. Available: https://hal.inria.fr/hal-01662158/file/doc.pdf
[22] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 11, pp. 2035–2045, Nov. 2007.
[23] Y.-M. Kuo, F. Garcia-Herrero, and J. A. Maestro, "Versatile RISC-V ISA Galois field arithmetic extension for cryptography and error-correction codes," in *Proc. 5th Workshop Comput. Archit. Res. RISC-V*, 2021, pp. 1–6.
[24] J.-P. Deschamps, J. L. Imana, and G. D. Sutter, *Hardware Implementation of Finite-Field Arithmetic*. New York, NY, USA: McGraw-Hill Education, 2009.
[25] Z. Xinmiao, *VLSI Architectures for Modern Error-Correcting Codes*. Boca Raton, FL, USA: CRC, 2016.
[26] A. S. Waterman, *Design of the RISC-V Instruction Set Architecture*. Berkeley, CA, USA: Univ. of California, 2016.
[27] A. Zeh *et al.*, "RISC-V cryptographic extension proposals - Version 0.6.1," RISC-V, Switzerland, Tech. Rep., pp. 1–44, 2020. [Online]. Available: https://lists.riscv.org/g/tech-crypto-ext/attachment/259/0/riscv-crypto-spec-v0.6.1.pdf
[28] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, "The design of scalar AES instruction set extensions for RISC-V," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2021, no. 1, pp. 109–136, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8729
[29] K. Stoffelen, "Efficient cryptography on the RISC-V architecture," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, 2019, pp. 323–340.
[30] G. Xin *et al.*, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2672–2684, Aug. 2020.
[31] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2020, no. 4, pp. 239–280, Aug. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8683
[32] S. Pircher, J. Geier, A. Zeh, and D. Mueller-Gritschneder, "Exploring the RISC-V vector extension for the Classic McEliece post-quantum cryptosystem," in *Proc. 22nd Int. Symp. Qual. Electron. Des.*, 2021, pp. 401–407.
[33] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic: Accelerating Kyber and NewHope on RISC-V," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2020, no. 3, pp. 219–242, Jun. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8589
[34] Y.-M. Kuo, F. Garcia-Herrero, O. Ruano, and J. A. Maestro, "Flexible and area-efficient Galois field arithmetic logic unit for soft-core processors," *Comput. Elect. Eng.*, vol. 99, 2022, Art. no. 107759.
[35] B. Sunar, E. Savas, and C. K. Koc, "Constructing composite field representations for efficient conversion," *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1391–1398, Nov. 2003.
[36] J.-L. Danger *et al.*, "On the performance and security of multiplication in GF(2N)," *Cryptography*, vol. 2, no. 3, 2018, Art. no. 25. [Online]. Available: https://www.mdpi.com/2410–387X/2/3/25
[37] W. D. Corporation, "RISC-V SweRV-EL2 github repository," 2020. [Online]. Available: https://github.com/chipsalliance/Cores-SweRV-EL2
[38] A. Halbutogullari and C. K. Koc, "Mastrovito multiplier for general irreducible polynomials," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 503–518, May 2000.
[39] P. C. McGeer, J. V. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vicentelli, "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1, no. 4, pp. 432–440, Dec. 1993.
[40] W. D. Corporation, "RISC-V SweRV-EL2 github repository fork," 2020. [Online]. Available: https://github.com/kuoyaoming93/Cores-SweRV-EL2
[41] Tiny AES github repository, 2012. [Online]. Available: https://github.com/kokke/tiny-AES-c
[42] Reed-Solomon code github repository, 2018. [Online]. Available: https://github.com/kbrafford/rscode
[43] Classic McEliece round-3 submission package, 2021. [Online]. Available: https://classic.mceliece.org/nist/mceliece-20201010.tar.gz
[44] M. Dworkin, "National institute of standards and technology special publication 800–38A 2001 edition," *Inst. Stand. Technol. Spec. Publ.*, vol. 800, p. 38A, 2001. [Online]. Available: ftp://217.17.192.66/pub/mitarb/lutz/standards/nist/sp800-38a/sp800-38a.pdf
[45] O. Kindgren, "SweRVolf github repository," 2019. [Online]. Available: https://github.com/chipsalliance/Cores-SweRVolf

Yao-Ming Kuo received the MEng degree in electronic engineering from the Universidad Tecnológica Nacional (UTN) in Buenos Aires, Argentina, in 2018. He is currently working toward the PhD degree in ARIES Research Center with the aim of completing his thesis in 2022. He has previously worked as a reasearcher with UTN and as a hardware design engineer with INTI (National Institute of Industrial Technology, Argentina) in the Micro and Nanoelectronics Department. He has designed ASIC, FPGA and Microcontroller based systems for instrumentation sensors and power-save applications with INTI for third party companies. His interest areas include digital signal processing, digital design and implementation, fault-tolerance, and reliability.

Francisco García-Herrero received the BSc degree in telecommunication engineering from the Escuela Politecnica Superior de Gandia, Spain, in 2008, and the MS and PhD degrees in electrical engineering from the Universitat Politècnica de València, Spain, in 2010 and 2013, respectively. He has worked as a lecturer and a researcher with several universities, including the European University Miguel de Cervantes and the Universitat Politècnica de València. He is currently an Associate Professor and a researcher with the Universidad Antonio de Nebrija. His research interests include hardware and algorithmic optimizations of error-control decoders and fault-tolerance electronics in communication and storage systems.

Oscar Ruano received the MSc and PhD degrees in computer engineering from the Universidad Antonio de Nebrija, in 2005 and 2011, respectively. He has served as a lecturer and researcher with the Universidad Nebrija with a research contract supported by the Regional Government of Madrid. He has developed his activity in the Space field, with different research projects on fault tolerance optimization against radiation effects in microelectronic circuits. He is the author of several technical publications, both in journals and international conferences. Also, he has worked with different multinational companies in the IT consultancy field, as Accenture. His areas of interest include digital design, fault-tolerance, and reliability.

Juan Antonio Maestro (Senior Member, IEEE) received the MSc degree in physics and the PhD degree in computer engineering from the Universidad Complutense de Madrid, Madrid, Spain, in 1994 and 1999, respectively. He is currently a full professor with the Universidad Complutense de Madrid, in the computer architecture field. His current activities are oriented to the space industry, with several projects on the protection of digital circuits against the effects of radiation, including microprocessors, memories and auxiliary systems. He also collaborates with institutions as the European Space Agency, Stanford University, University College Dublin or the Harbin Institute of Technology, among others. He is the author of numerous technical publications in journals and international conferences. His areas of interest include computer architecture, digital design, fault-tolerance, reliability, small satellites, and space applications.