electronics

MDPI

Article

Hardware Acceleration for RLNC: A Case Study Based on the Xtensa Processor with the Tensilica Instruction-Set Extension

Javier Acevedo [1], Robert Scheffel [2], Simon Wunderlich [1], Mattis Hasler [2], Sreekrishna Pandi [1], Juan Cabrera [1], Frank H. P. Fitzek [1], Gerhard Fettweis [2] and Martin Reisslein [3,*]

[1] 5G Lab Germany, Deutsche Telekom Chair of Communication Networks, TU Dresden, 01062 Dresden, Germany; javier.acevedo@mailbox.tu-dresden.de (J.A.); simon.wunderlich@tu-dresden.de (S.W.); sreekrishna.pandi@tu-dresden.de (S.P.); juan.cabrera@tu-dresden.de (J.C.); frank.fitzek@tu-dresden.de (F.H.P.F.)

[2] 5G Lab Germany, Vodafone Chair Mobile Communication Systems, TU Dresden, 01062 Dresden, Germany; robert.scheffel1@tu-dresden.de (R.S.); mattis.hasler@tu-dresden.de (M.H.); gerhard.fettweis@tu-dresden.de (G.F.)

[3] School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85287, USA

* Correspondence: reisslein@asu.edu; Tel.: +1-480-965-8593

† This article is based upon work supported by in part by the Free State of Saxony through funds from the European Commission for the Atto3-D Project and the German Research Foundation (DFG) within the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

Received: 8 August 2018; Accepted: 5 September 2018; Published: 8 September 2018

Abstract: Random linear network coding (RLNC) can greatly aid data transmission in lossy wireless networks. However, RLNC requires computationally complex matrix multiplications and inversions in finite fields (Galois fields). These computations are highly demanding for energy-constrained mobile devices. The presented case study evaluates hardware acceleration strategies for RLNC in the context of the Tensilica Xtensa LX5 processor with the tensilica instruction set extension (TIE). More specifically, we develop TIEs for multiply-accumulate (MAC) operations for accelerating matrix multiplications in Galois fields, single instruction multiple data (SIMD) instructions operating on consecutive memory locations, as well as the flexible-length instruction extension (FLIX). We evaluate the number of clock cycles required for RLNC encoding and decoding without and with the MAC, SIMD, and FLIX acceleration strategies. We also evaluate the RLNC encoding and decoding throughput and energy consumption for a range of RLNC generation and code word sizes. We find that for GF(2^8) and GF(2^{16}) RLNC encoding, the SIMD and FLIX acceleration strategies achieve speedups of approximately four hundred fold compared to a benchmark C code implementation without TIE. We also find that the unicore Xtensa LX5 with SIMD has seven to thirty times higher RLNC encoding and decoding throughput than the state-of-the-art ODROID XU3 system-on-a-chip (SoC) operating with a single core; the Xtensa LX5 with FLIX, in turn, increases the throughput by roughly 25% compared to utilizing only SIMD. Furthermore, the Xtensa LX5 with FLIX consumes roughly three orders of magnitude less energy than the ODROID XU3 SoC.

Keywords: application-specific instruction-set processor (ASIP); flexible-length instruction extension (FLIX); galois field; hardware acceleration; multiply-accumulate (MAC) operations; random linear network coding (RLNC); single instruction multiple data (SIMD)

1. Introduction

Random linear network coding (RLNC) [1–3] is an increasingly popular coding method for complex, chaotic, or lossy communication networks. RLNC reduces the number of transmissions

needed to achieve a prescribed resilience level (probability of packet delivery) and has the potential to reduce transmission delays. RLNC is well suited for wireless data transmissions [4–19], data storage [20–23], and content distribution [24–30]. The practical usage of RLNC for the wide range of communication applications in emerging cyber-physical systems (CPS) and the internet of things (IoT) as introduced by fifth generation (5G) wireless systems requires high RLNC encoding and decoding throughput while consuming only low amounts of energy. A general strategy for boosting computing performance while lowering energy consumption is to employ an application-specific instruction-set processor (ASIP) instead of a general-purpose fixed instruction-set processor. The ASIP approach typically achieves speedups of up to three orders of magnitude, while reducing power consumption to a fraction of a standard fixed instruction-set processor, such as a reduced instruction set computer (RISC) processor [31–33]. While the computing aspects of RLNC encoding and decoding on fixed instruction-set processors have been extensively studied, to the best of our knowledge, the RLNC encoding and decoding performance on ASIP systems has not previously been examined in detail. We conduct a case study with a specific contemporary ASIP example, namely the Tensilica Xtensa LX5 processor with tensilica instruction-set extension (TIE) [34] to quantify the RLNC performance of a contemporary ASIP in comparison to a fixed instruction set processor.

We describe the development of the application-specific TIEs for RLNC encoding and decoding. Specifically, we develop application-specific TIEs for the matrix multiplication and matrix inversion over Galois fields to accomplish the RLNC encoding and decoding. We consider a range of TIE-based hardware acceleration strategies ranging from elementary multiply-accumulate (MAC) units for matrix multiplication to the parallelization of computations with the single instruction multiple data (SIMD) and flexible length instruction extension (FLIX) strategies. We thoroughly evaluate the performance of the developed ASIP system with the range of TIEs. In particular, we compare the Tensilica Xtensa LX5 with the range of TIEs in terms of hardware area (die size) as well as the number of compute cycles required for the RLNC encoding and decoding and the corresponding achieved throughput levels. We also evaluate the energy consumption. The performance comparisons with a state-of-the-art fixed instruction set processor indicate roughly tenfold throughput increases while reducing the consumed energy by roughly three orders of magnitude with the studied ASIP system.

This article is organized as follows: Section 2 gives the background on the main areas relating to this case study, namely, the mathematics as well as the computational aspects of RLNC encoding and decoding using Galois fields. Section 2 also briefly describes the main features of the Xtensa LX5 processor and the TIE design flow. Section 3 introduces the developed TIEs for the MAC operation, and the optimizations using the SIMD and FLIX strategies. Section 4 presents the evaluation methodology which includes the evaluation of the hardware area (die size), the required compute cycles and the achieved throughput for RLNC encoding and decoding as well as the consumed energy. Moreover, Section 4 presents the evaluation results for the different Xtensa LX5 TIEs as well as comparisons with a fixed instruction-set benchmark processor. Finally, Section 5 summarizes the conclusions of this case study on RLNC encoding and decoding with an ASIP vs. a fixed instruction-set processor and outlines future research directions.

2. Background and Related Work

2.1. Mathematics of RLNC

RLNC [3,35] combines source (original data) symbols through linear combinations in the Galois field ($GF(q)$) to create coded (redundant) symbols. Source symbols can be represented as row vectors of $GF(q)$ elements. In a common communication scenario, such as Ethernet, with a maximum transfer unit of 1500 bytes and a Galois field size of $q = 2^8$, each Ethernet frame is represented by $m = 1500$ elements of GF (2^8). Shorter frames require zero-padding at the end of each vector.

A given set of g source packets can be protected by r redundant coded packets during network transport, whereby the number r, $r \geq 0$, of redundant coded packets can be adjusted according to the

expected losses during network transport. For RLNC encoding, a coefficient matrix $\mathbf{C}$ with $g + r$ rows and g columns is multiplied by the source data (symbol) matrix $\mathbf{M}$ to obtain the coded symbol matrix $\mathbf{X}$ as

$$\mathbf{X} = \mathbf{CM} \tag{1}$$

Each coded symbol (x) is represented as a row of matrix $\mathbf{X}$, which is transmitted along with its corresponding coefficient (row) vector c of matrix $\mathbf{C}$ as a coded packet.

The receiver collects the received coded symbols in matrix $\bar{\mathbf{X}}$ and the received coding vectors in matrix $\bar{\mathbf{C}}$. When at least g linearly independent coded packets have arrived, the receiver conducts the RLNC decoding by inverting matrix $\bar{\mathbf{C}}$ and multiplying by matrix $\bar{\mathbf{X}}$ to reconstruct the original data (symbol) matrix $\mathbf{M}$:

$$\mathbf{M} = \bar{\mathbf{C}}^{-1}\bar{\mathbf{X}} \tag{2}$$

Equations (1) and (2) indicate that the RLNC encoding and decoding have matrix multiplication in common; moreover, RLNC decoding requires the matrix inversion of $\bar{\mathbf{C}}$ before multiplication.

Aside from the computation delay for RLNC encoding and decoding, the latency introduced by RLNC depends on the employed network coding approach. There are two main approaches for network coding for low-latency communications: coding based only on Exclusive OR (XOR) operations, see, e.g., [36–42], and coding based on feedback from the receivers to the sender, see,e.g., [43–48]. The XOR approach corresponds to network coding for binary Galois fields, GF(2). Conversely, we considered RLNC with general Galois fields, GF(q), since they provide optimal throughput and have a negligible linear dependency of the coding coefficient vectors when the dimension of the field is sufficiently large, e.g., 8 or 16 [49,50]. Feedback-based approaches adapt various network coding parameters to establish a closed-loop feedback system between the encoder and decoder [51,52]. In this study, we considered the simpler batch-based network coding approach that does not require feedback. The batch-based approach has been considered for a wide range of communication scenarios, such as delay-tolerant networks in [53], video delivery networks in [54,55], industrial networks in [56,57], and some complex sensor networks in [58]. Recent studies have adapted the batch positions and sizes to conform to different sliding window policies in an effort to reduce the RLNC communication protocol delay [59–64].

2.2. Computation of RLNC

Systems with parallelization require multiple processors to accelerate the RLNC computations [65–72]. High-performance applications exploit the parallelization provided by general purpose computation on graphics processing units (GPGPUs) [73]. However, the implementation of those applications is not an easy task, since GPU code is developed independently from the main central processing unit (CPU) application. In large-scale computing systems, such as desktop computers and servers with thousands of GPU threads, the data transfer delays between the main CPU and the GPU memory can be amortized due to the vast processing power from the numerous GPU threads. On the other hand, small-scale systems, such as IoT nodes and smartphones, typically have only small numbers of GPU threads [74]. Therefore, on IoT nodes and smartphones, the data transfer delay to the GPU threads typically cannot be amortized with the moderately increased processing on the few available GPU threads.

Advances in semiconductor technology and programming schemes have led to mobile systems with heterogeneous multi-core processors [75]. These processors can execute general instruction sets to exploit data parallelization, such as SIMD and FLIX, enabling innovative network coding techniques. One example of those extended functionalities is parallelized progressive network coding (PPNC) with SIMD instructions [76] which statically partitions the coefficient matrix $\bar{\mathbf{C}}$ and the coded symbol matrix $\bar{\mathbf{X}}$ vertically to threads running in parallel on the multiple cores.

RLNC computation on a fixed instruction-set Raspberry PI (Raspberry Pi Foundation, Cambridge, United Kingdom) testbed was studied in Refs. [77,78]. Related energy saving strategies were studied

in Refs. [79,80]. In particular, Arrobo and Gitlin [79] compared cooperative network coding with a cooperative diversity coding approach. Fiandrotti et al. [80] introduced band codes, a variant of network coding with controlled packet degree distribution. However, to the best of our knowledge, the encoding throughput increases and the reductions of consumed energy that can be achieved by computing the RLNC encoding and decoding with an ASIP have not yet been quantified. We note that a circuit-level very large scale integration (VLSI) implementation of RLNC was examined in Ref. [81]. In contrast, we considered a contemporary ASIP, namely, the Tensilica Xtensa LX5 (Cadence Design Systems, Inc., San Jose, CA, USA) processor with TIEs [34,82], which avoids the complexities and costs of specific VLSI designs.

Efficient calculations over Galois fields were heavily investigated for Rijndael encryption [83–86] and elliptic curve cryptography [87–89] which form the basis for data encryption mechanisms, such as the advanced encryption standard (AES). AES does not use regular multiplication, but rather, involves constant multipliers for mix-column transformation [90]. In contrast, RLNC requires multiplications with arbitrary values. Therefore, the proposed AES computing architectures cannot be directly used for RLNC.

2.3. Xtensa LX5 Processor and TIE Design Flow

The Xtensa LX5 core features two 128-bit local load/store units to allocate the data of the basic registers, the application-specific registers, and the application-specific states. The Xtensa LX5 can thus simultaneously load two 128-bit values from memory. Additionally, the Xtensa LX5 has five stages of pipelining: instruction fetching, register access, execute, data-memory access, and write-back. Additionally, there is a seven-stage pipeline extension with extra instruction fetching and data-memory access stages, which provides access to the processor's main memory. The Xtensa LX5 has a 64-bit instruction register to hold the basic RISC and the application-specific instruction-sets from local instruction memory buffers.

The design flow for the Xtensa LX processor family consists of three stages: functional C/C++ code, TIE creation, and TIE compilation. The flow starts with the profiling of the user-defined functional application (reference) code written in C/C++ language. We implemented the RLNC encoding and decoding algorithms directly in a reference C code, without a specific C code optimization. We deemed C code optimization to be unnecessary, as the RLNC encoding and decoding algorithms can be directly implemented through elementary C programming, and the focus of this case study was on hardware acceleration strategies. In order to detect hotspots in the code, the Xtensa C compiler can rank code regions by frequency, determine the loops that can be vectorized, generate dataflow graphs, and perform operation counts of every code region. Alternatively, the hotspots can be manually detected through inspection and the intuitive understanding of computing functionality. The TIE code creation maps the functional behaviors of the detected hotspots to a set of TIE code instructions. For this case study, we detected the hotspots through Xtensa's automatic profiling tool. From the obtained profiling data, we derived the hotspots and decided where to apply the TIE workflow to achieve accelerations. The thus derived hotspots confirmed the hotspots that we anticipated from our intuitive understanding of the computing functionality. We then proceeded to develop hand-crafted acceleration solutions, i.e., we wrote the TIE code manually without the assistance of an automatic code generation tool. Since the TIE code resembles Verilog code, the programmed instructions, also known as operations, maintain the register-transfer level of abstraction. It is noteworthy that the TIE code describes functionalities through TIE functions rather than hardware structures, i.e., the Xtensa design flow does not require hardware modules to be created, e.g., by writing Verilog code; instead, built-in functions, such as TIEMUX and TIEADD, are specified to operate directly with the data.

The TIE compiler, in conjunction with the Xtensa C compiler, generates a set of software tools, also known as function intrinsics, that tune the designer-defined TIE instructions with the processor's default setup. We conducted the verification with the automated instruction set simulator (ISS). More specifically, the ISS verifies whether the results of the modified application code, which

contains the custom TIE instructions, match the results of the user-defined functional application code. If no discrepancies (errors) arise during the verification, the TIE compiler generates a hardware implementation of the modified application code as a synthesizable Verilog optimized for the Xtensa core.

3. Hardware Acceleration for RLNC through TIEs

3.1. Accelerating Matrix Multiplication with Multiply-Accumulate (MAC) Hardware

An elementary matrix multiplication implementation uses three nested loops [91]. The inner loop executes multiply-add operations. A coefficient value of the coefficient matrix **C** (which does not change in the inner loop) and a source packet value of the source matrix **M** are multiplied using Galois field operations. The product is then "added" to the result within the Galois field, which corresponds to an XOR operation.

We observed the following potential for optimization: First, the core operation is a multiply-add operation in a Galois field using a static coefficient. This operation is simple enough to be implemented in a MAC hardware unit. Second, the multiply-add operation is performed at consecutive memory locations, therefore allowing SIMD operations, as further pursued in Section 3.2.

Multiply-accumulate (MAC) architectures are hardware units that execute integer-multiply, integer-accumulate, and miscellaneous register instructions. Integrating the multiply and accumulate (add) operations into a single instruction improves the accuracy (since only a single rounding operation is required at the end of each combined multiply-add operation) and reduces the hardware cost. The combined instruction is also known as fused multiply-add (FMA) instruction. The multiplication and addition operations can be performed in parallel during one processor clock cycle, ensuring high performance. While the multiplier is computing the result of two registers, the accumulator can store the result of the previous multiplication operation by adding its value to the product of the former two registers [92,93].

MAC architectures are common in multiple digital signal processing functions, such as the finite fourier transform (FFT) and infinite impulse response (IIR), since most of those functions require the computation of the sum of the products of a series of consecutive multiplications. However, in this case study, we applied this architecture to matrix-matrix multiplication using Galois fields in which the rows of the coefficient matrix **C** were multiplied with the columns of the source matrix **M** to generate each element of the coded symbol matrix **X**.

The Galois field multiplication of two polynomials (8-bit values) in GF(2^8), a and b can be based on the Russian Peasant Multiplication algorithm [94]. In particular, the final product polynomial (p) is calculated using an XOR operation with the irreducible polynomial of grade 8, $x^8 + x^4 + x^3 + x + 1$, which is used for the shifting operation of the operand a illustrated on the right side in Figure 1. In particular, the multiplication consists of two parts: computing the values of a to be used in the algorithm and the calculation of the product (p). If the most significant bit of a after a preceding shifting stage ($s-1$) is a one, i.e., if $a^{(s-1)}[7] = 1$, then the grade of the polynomial a must be reduced in shifting stage s through an XOR operation with the irreducible polynomial. Otherwise, i.e., if $a^{(s-1)}[7] = 0$, then the values of a remain unaltered. The evaluation of the product (p) accumulates the values of a after the various shifting stages which are selected according to the values of b.

Since the evaluation of product p depends on the values of a after the various shifting stages and the successive accumulation of the various stages (s) of the $p^{(s)}$ values, the multiplication computation cannot be parallelized. However, the multiplexing functionality can be implemented in a TIE with 15 multiplexers. The seven multiplexers on the left side of Figure 1 evaluate whether the a value must be reduced (sifted) or not. The eight multiplexers on the right accumulate the product value of p. In particular, the first accumulation multiplexer at the top-right of Figure 1 takes in the unshifted a value, whereas, the seven subsequent accumulation multiplexers on the right side of Figure 1 take in

shifted versions of a. Thus, only seven multiplexers are required for the shifting of a, even though there are eight multiplexers that accumulate the product values.

The MAC unit can be designed in three different operand-size configurations: 8-bit only MAC, 16-bit only MAC, as well as 8-bit and 16-bit MAC. The 8-bit MAC consists of 15 multiplexers (routing 8-bit values), as illustrated in Figure 1. Analogously, the 16-bit MAC consists of 31 multiplexers (routing 8-bit values). The 8-bit and 16-bit MAC is composed by two multiplexing sections: one 8-bit MAC section and one 16-bit MAC section. These two MAC sections have the underlying structure of the 16-bit MAC unit and thus require twice the number of multiplexers of the 16-bit MAC unit, i.e., 62 multiplexers are required to implement this flexible 8-bit and 16-bit MAC. The selection of 8-bit or 16-bit operation is carried out by a 1-bit flag register.

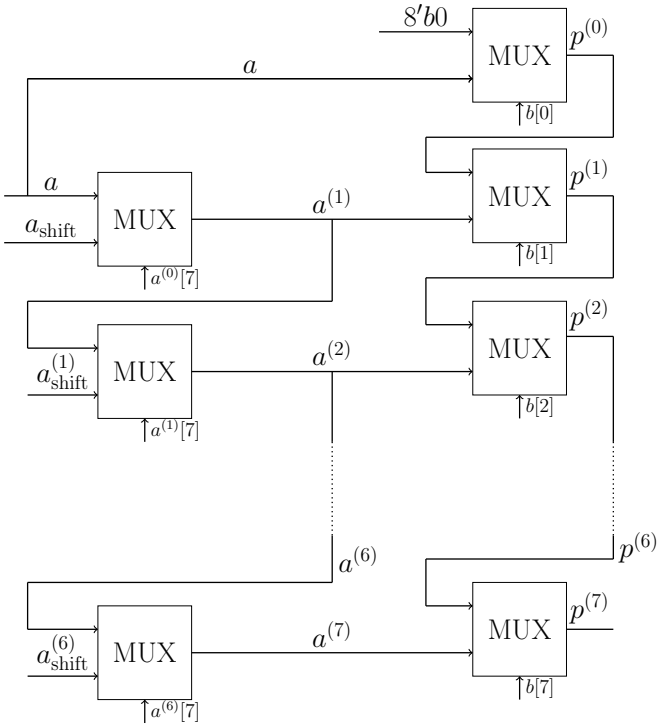


Figure 1. Block diagram of the multiplier: Two 8-bit operands a and b are multiplied by the Russian Peasant Multiplication algorithm [94] in the Galois field (GF(2^8)) to obtain the product (p). The seven multiplexers on the left control the shifting of the operand (a); the shifting in a given stage (s) is controlled by the most significant (7th) bit of operand a after the preceding $(s-1)$ shifting stage, i.e., by the bit $a^{(s-1)}[7]$. The eight multiplexers on the right accumulate the outcome of the product (the input "8'b0" denotes eight bits with value zero), with the final product (p) provided by $p^{(7)}$.

3.2. SIMD

Single instruction multiple data (SIMD) is an optimization technique based on loop vectorization that exploits the alignment of parallel data. In order to increase the computation performance and reduce hardware costs, the SIMD architecture exploits consecutive and aligned memory accesses to achieve a high vectorization efficiency [95]. Thus, by using SIMD, an operation can be simultaneously performed on multiple data operands if the result of the operation on one operand does not affect the result of this operation on the other (parallel) operands. Figure 2 depicts the block diagram of the SIMD multiplier. For the 8-bit operand configuration, 16 source registers are multiplied in parallel by a coefficient in 16 MAC units. The structure of each of those MAC units is identical to the MAC unit in Figure 1, since the arithmetic operation is in GF(2^8).

As Figure 3 illustrates, we loaded two 128-bit registers that allocated the source values (Line 5) and the encoded values (Line 6). The coefficients were stored in an internal register that can be accessed as a global state locally in the TIE file. Then, we computed those two arrays in 16 parallel multiplications (Line 8). Finally, the resulting values were saved into the preloaded encoded register (Line 10). This procedure is completely equivalent for the decoding operation.

For the configuration with 16-bit operands, eight 16-bit MAC units operate in parallel to conform with the 128-bit LX5 RAM memory line. In our design, the coefficients were stored as a local register in the TIE code file to reduce the usage of fetch and store registers when the encoding and decoding operations were executed.

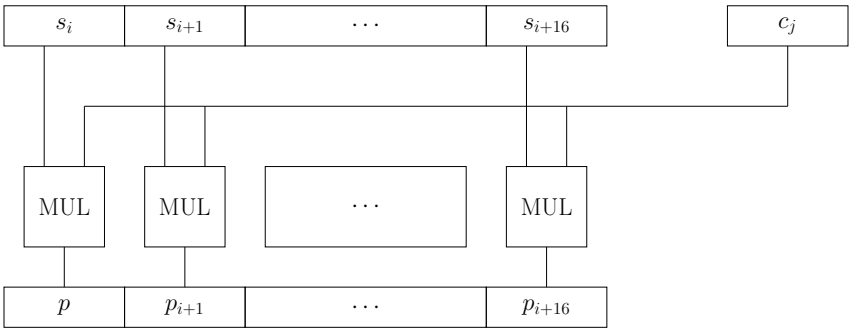


Figure 2. Block diagram of the single instruction multiple data (SIMD) multiplier for 8-bit operands:16 MAC units were employed in parallel to conform to the 128-bit memory interface of the LX5 processor.

```
1  for (round = 0; round < TESTROUNDS; round++) {
2      for (i = 0; i < GENSIZE; i++) {
3          WUR_coeff(coefficients[i]);   /* load coding coefficients */
4          for (j = 0; j < PACKETSIZE; j+=16) {
5              reg_source = tie_load_reg(source, j); /* load 128-bit source reg. ;*/
6              reg_encoded = tie_load_reg(encoded, j);
7                 /* load 128-bit encoded register with uncoded initialization input */
8              tie_16x_b8_mult(reg_source, reg_encoded);
9                 /* perform 128-bit multiplication using 16 8-bit MACs; */
10             tie_store_reg(encoded, j, reg_encoded);
11                /* store result in 128-bit register to memory */
12         }
13     }
14 }
```

Figure 3. SIMD code: Illustration of the transformation of plain C code to a code with tensilica instruction-set extension (TIE) operations.

3.3. Matrix Inversion

We implemented the GF inversion of each value of the coefficient matrix $\bar{\mathbf{C}}$ of Equation (2) in a TIE. Subsequently, the inverted values of the coefficient matrix were multiplied by the encoded symbol matrix $\bar{\mathbf{X}}$ following Equation (2) to complete the RLNC decoding.

A direct GF(2^8) inverse implementation without any Galois field transformation is computationally complex. Many applications use static lookup tables, which require 256 entries for GF(2^8). When it is implemented in hardware, the table lookup is possibly the fastest approach; however, the table lookup requires a large area, especially for large field sizes. A compromise is to transform the elements from GF(2^8) into GF($(2^4)^2$), and to compute the inverse in GF(2^4) by twice looking up a table with 16 elements. The resulting element in GF($(2^4)^2$) can then be transformed back to GF(2^8) with the so-called fast inversion in composite Galois fields [96].

Figure 4 gives the block diagram of the matrix inversion in hardware. An initial 8-bit value a is split into two 4-bit registers, a_0 and a_1. Then, MAC operations in Galois fields are used to evaluate the inverse, which requires table lookups of 16 pre-computed reciprocal values in GF(2^4). The resulting

two 4-bit inverse values, b_0 and b_1, are then joined through elementary bit rearrangements to obtain a single 8-bit value that represents the 8-bit inverse of a.

Similarly to the inversion of an element in the Galois field (GF(2^8)), the computation of an element for GF(2^{16}) can be decomposed into GF($((2^4)^2)^2$), allowing the reuse of structures created for the 8-bit inverse calculation. As a result, the inverse is computed in GF(2^4) by using four times a lookup table with 16 elements. This strategy can be extended to the computation of various elements using large Galois fields, effectively keeping the area consumption low at the expense of increasing the number of computation cycles.

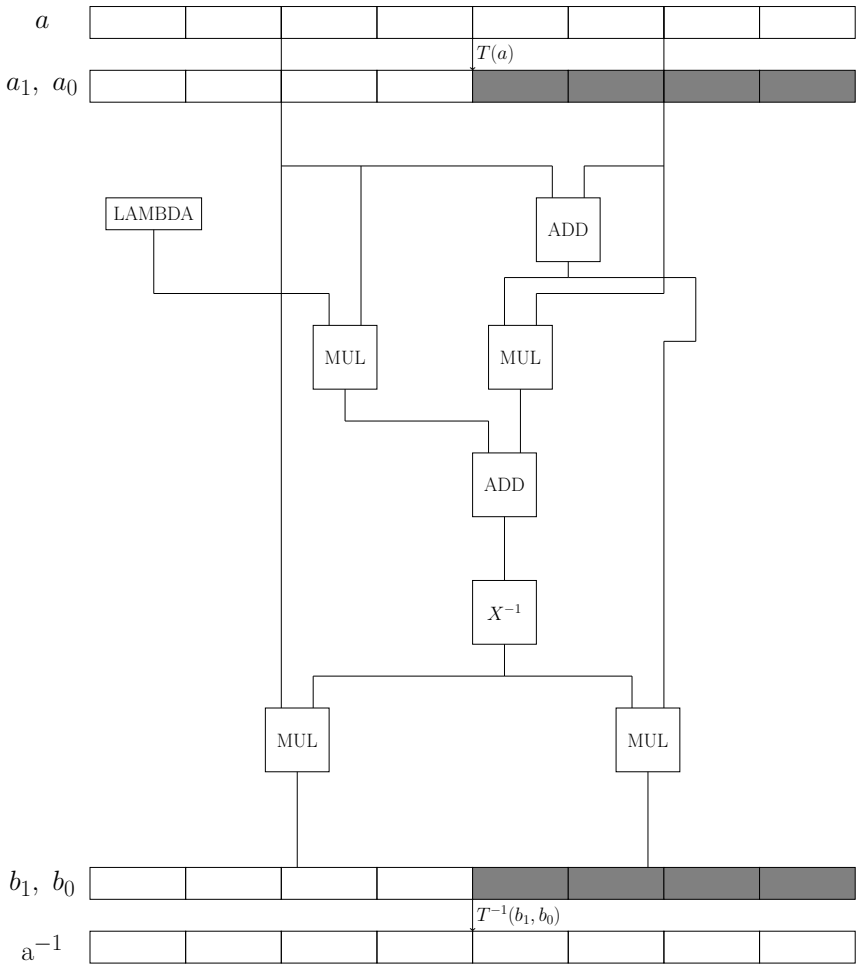


Figure 4. Block diagram of the matrix inversion: One 8-bit value is split into two 4-bit registers. The inverse operation is carried out through the multiplication of each 4-bit value with its reciprocal in GF(2^4), which is obtained from a lookup table. The resulting two 4-bit values are reorganized using bit arrangements to obtain the 8-bit inverse value.

3.4. FLIX

The flexible length instruction extension (FLIX) combines multiple instructions into one instruction using the tensilica instruction-set extension (TIE) [97,98] programming language. The FLIX acceleration method can save cycles at the expense of an increased number of gates, i.e., an increased chip area [99].

The loading instructions `tie_load_reg()` to load the source and the encoding and decoding registers, the multiplication instructions `tie_16x_b8_mult()`, and the storing instructions `tie_store_reg()` to store the resulting encoding and decoding values in Figure 3 can be combined into one FLIX TIE instruction. Each of the instructions in Figure 3 is divided into slots with independent

operations filling the slots either completely or partially. In this respect, a slot must be understood as a sequence of bits composed of an opcode, which provides the slot identity, and the operation and operand specifiers, such as source and destination registers. Specifically, the LX5 core has a 32-bit FLIX instruction format with two slots for the 8-bit and 16-bit load and store instructions and one slot for the multiplication instructions. This FLIX TIE requires all the operations and operands, which run in parallel, to be available multiple times in hardware. As a consequence, the FLIX optimization requires a large chip area.

As part of the exploitation of the FLIX feature, we placed the input vector and the output vector in two different memories. These two different memories made the input vector and the output vector accessible by the processor through two different memory ports. Thus, it was possible for the load operation and the store operation to be executed simultaneously within the FLIX environment. In a similar fashion, a load or a store operation can run in parallel to a MAC operation if the two operations work on disjoint registers. The user/designer only defines the possibility of composing multiple operations into one parallel FLIX command. The actual composition and insertion into the binary design are conducted by the compiler in one of its optimization phases.

4. Evaluation

This section presents the evaluation results in terms of die size, number of clock cycles, throughput, and energy consumption for RLNC encoding and decoding with the Tensilica LX5 processor using the MAC, SIMD, and FLIX TIE extensions. These different TIE extensions were compared with RLNC encoding and decoding through the execution of the plain C reference code on a Tensilica LX5 processor without any TIE. The plain C reference code was compiled with the `gcc-O2` optimization. We also compared the TIE extensions with a fixed instruction set benchmark processor.

4.1. Evaluation Setup

4.1.1. Evaluation Metrics

- Die size (Number of gates): The die size corresponds to the number of `NAND-2` gates (specifically, 2-input `NAND` gates for 8-bit operands) that are required to map the register-transfer level (RTL) circuit to `NAND-2` gates [100] in order to achieve the prescribed RLNC encoding and decoding functionality. The gate count of the TIE is measured as the number of `NAND-2` equivalents as every logical equation can be expressed with only `NAND` gates [101]. We obtained the die size (in number of equivalent `NAND-2` gates) directly from the Instruction Set Simulator (ISS) in the Xtensa Development Environment, Version 6.0.2 with the default profiler settings.
- Clock cycles: The number of clock cycles represents the number of computing cycles required to execute RLNC encoding or decoding of a generation of g symbols, each of size m bytes, with the LX5 processor. One clock cycle represents the time required between two successive pulses of the internal oscillator of the processor [102] which, in the current study, operated at a clock frequency of 300 MHz. Similar to the die size, we obtained the clock cycles with the ISS in the Xtensa Development Environment, Version 6.0.2, with the default profiler settings.
- Speedup: We defined the processor speedup as the ratio of the number of computing cycles required by the reference implementation to complete a prescribed task to the number of computing cycles required by the optimized (accelerated) implementation to complete the same prescribed task [103]. We considered the plain-C code executed on the LX5 processor, i.e., the basic LX5 processor without any application specific Tensilica Instruction-set Extension (TIE), to be the reference implementation.
- Throughput: We evaluated the GF(2^8) and GF(2^{16}) RLNC encoding and decoding throughputs as the size ($g \cdot m$) of the generation in bytes divided by the required computing time, obtained as the number of computing cycles times the inverse of the clock frequency. We report the encoding and decoding throughputs in units of MiB/s, whereby 1 MiB = 2^{20} bytes.

- Energy consumption: We obtained the consumed power levels for each TIE acceleration strategy as follows. We conducted RTL synthesis of the LX5 with TIE processor with the Synopsys Design Compiler to obtain a so-called netlist. Then, we employed Mentor Questasim to conduct a netlist simulation of the processor running the algorithm to obtain the toggle activity data for every flip flop and wire. Finally, we employed Synopsys PrimeTime to conduct a power consumption analysis for the netlist under the obtained toggle activity. The obtained power levels represent the power consumed by all elements of an equivalent digital circuit that executes the same functionality as our RLNC encoding and decoding algorithm. Specifically, for the evaluation of the consumed energy, we considered the GF(2^8) RLNC encoding of a generation followed immediately by the corresponding RLNC decoding to evaluate a complete encoding and decoding sequence. We obtained the average consumed power in Watts = Joules/s for the sequence of RLNC encoding and decoding. We evaluate the energy consumption as the ratio of consumed power (in Joules/s) to the throughput (in MiB/s), i.e., the energy consumption is in units of Joules/MiB of GF(2^8) RLNC encoding and decoding.

4.1.2. Fixed Instruction-Set Processor Benchmark

We consider the state-of-the-art ODROID XU3 (Hardkernel Co., Ltd., AnYang, GyeongGi, South Korea) system-on-a-chip as a fixed instruction set processor benchmark. The ODROID XU3 integrates a "big" Samsung Cortex A15 quad-core CPU and a "LITTLE" Samsung Cortex A7 quad-core CPU (Samsung Group, Seoul, South Korea) and has 2 GB LPDDR3 RAM, eMMC 5.0 flash storage, and several peripheral interfaces, such as USB, UART, and I2C. For a fair comparison with the Tensilica Xtensa LX5 single-core processor, we considered only one processor core (thread) of the big CPU. Conducting comparisons between a multicore ASIP, such as the Tomahawk processor [104], and multicore fixed instruction set multicore processors is an interesting direction for future research. The ODROID XU3 employed the default ARM SIMD instructions, also known as NEON that were developed to increase the processing performance for the Cortex A-x core families and were operated and evaluated with the state-of-the-art RLNC encoding and decoding computing methodology for fixed instruction set processors in [105].

4.2. Die Size and Gate Count

Table 1 gives the gate counts of the TIEs for the different hardware acceleration strategies using 8-bit and 16-bit register configurations, i.e., 8-bit and 16-bit word lengths (operands). First, we compared the MAC TIE acceleration with a traditional multiplier. In particular, we considered the Mastrovito Standard Base Multiplier, as it has one of the lowest gate counts among traditional multipliers [88,106]. The Mastrovito Standard Base Multiplier requires 64 `AND` gates and 84 `XOR` gates in the Galois field (GF(2^8)). These gate counts are equivalent to a total of 464 `NAND-2` gates. Our 8-bit GF(2^8) multiplier, described in Section 3.1, has a size of 968 `NAND-2` gates, i.e., about twice the number of gates of the Mastrovito Standard Base Multiplier [88]. This shows that the TIE acceleration comes at the expense of area deployment to perform arithmetics, specifically, the multiplexing (see Section 3.3) in parallel. In particular, our primitive MAC unit requires more than twice the number of gates required for the Mastrovito multiplier.

Table 1. Die size evaluation: Number (count) of NAND-2 equivalent gates for 8-bit operands in the Tensilica Instruction-Set Extensions (TIEs) for the different hardware acceleration strategies. The right-most column reports the gate counts of the SIMD and FLIX strategies as multiplicative factors with respect to the gate counts of the MAC strategy (the 8/16-bit SIMD and FLIX strategies consider the 16-bit MAC as corresponding basis.)

Acceleration Strategy	Number of TIE Gates (In Number NAND-2 Gates)	Increase as a Factor w.r.t. MAC
Original (basic Xtensa LX5 w/o TIE, used for benchmark C code)	0	–
Multiply-Add (MAC) 8 bit	968	0
Multiply-Add (MAC) 16 bit	1183	0
SIMD MAC 8 bit	10,542	10.9
SIMD MAC 16 bit	11,245	9.5
SIMD MAC 8/16 bit	33,768	28.5
Flexible length instruction extension (FLIX) 8/16 bit	45,412	38.4
Inverse 8/16 bit	12,692	–

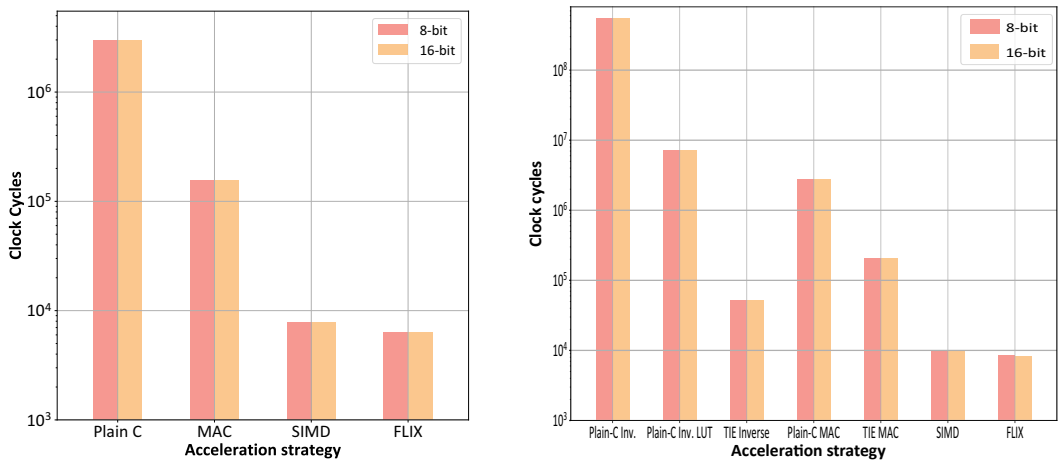
The SIMD parallel processing of multiple data values requires multiple parallel processing units. In our case, we employed multiple parallel multipliers. It can be observed from Table 1 that the SIMD MAC TIE for 8-bit operands requires 10,542 gates. To put this gate count in perspective, note that the 8-bit MAC utilizes 968 NAND-2 gates. Accordingly, the 8-bit SIMD Multiply-Add approach should require $16 \times 968 = 15,488$ NAND-2 gates, since we effectively employed 16 8-bit MAC units. However, in our implementation, the coefficients of $\mathbf{C}$ were stored in an internal 256-bit register that was preloaded in each MAC unit. This optimization reduced the total gate count by almost one-third for the SIMD with 8-bit MAC. On the other hand, for the SIMD with 16-bit MAC for GF(2^{16}), this pre-load register optimization was counterbalanced by additional circuitry to manage the 16-bit processing with elementary NAND-2 gates (processing 8-bits). Thus, the 11,245 NAND-2 gate count for SIMD with 16-bit MAC was slightly more than eight times the gate count for the 16-bit MAC.

The flexibility to compute data with two different word lengths (operand sizes) comes at the expense of a gate count increase to 33,768 NAND-2 gates. The flexible SIMD for 8-bit and 16-bit operands essentially require the gate counts for the individual 8-bit SIMD and the 16-bit SIMD as well as additional circuitry to coordinate the processing as well as the load and store operations for transferring the data from and to the 128-bit wide registers of the LX5.

It can be observed from Table 1 that the FLIX acceleration strategy requires the highest number of gates, 45,412 in total. This high gate count and corresponding large die size are required since the FLIX strategy encapsulates multiple operations, such as the SIMD Multiply-Add and the inverse instructions, into a single instruction that is executed in a single clock cycle. Consequently, more gates are required to execute these instructions in parallel.

4.3. Clock Cycles and Speedup

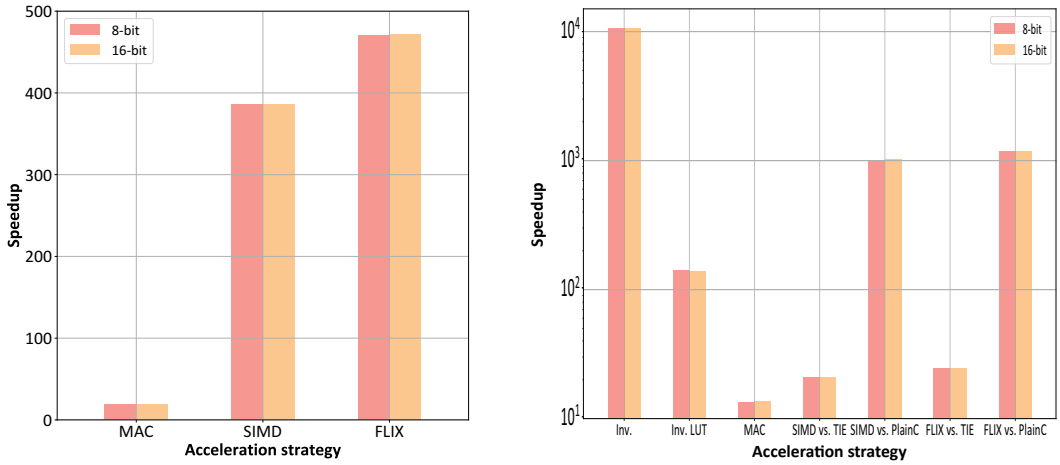
Figure 5 shows the numbers of clock cycles for the different acceleration strategies, and the corresponding speedups are shown in Figure 6. It can be observed from Figure 6a that for the RLNC encoding, the SIMD integration achieves very substantial speedups (approximately 350–380 fold) while requiring a moderately large (around 10.5 thousand NAND gates) TIE. In contrast, the FLIX conversion achieves only slightly higher speedups (approximately 420–460), while requiring a very large TIE (around 45.4 thousand NAND gates). We note that the SIMD instruction is encapsulated within the FLIX instruction, i.e., FLIX essentially represents the combination of SIMD and FLIX.



(**a**) Encoder (**b**) Decoder

Figure 5. Comparison of number of clock cycles for random linear network coding (RLNC) with a generation size of $g = 16$ and packet size of $m = 1536$ bytes. The plain C results correspond to the implementation without any application specific Tensilica Instruction-set Extension (TIE). The MAC, SIMD, and FLIX results correspond to the different examined TIEs with the LX5 core instruction-set. For the decoder, we separately report in part (**b**) the numbers of clock cycles for the matrix inversion, specifically with elementary multiplication operations (Plain C Inv.) with a look-up table (Plain C Inv. LUT) and with a TIE (TIE Inverse), and for the matrix multiplication (MAC without and with TIE), as well as the numbers of clock cycles for the complete decoding with SIMD and FLIX TIEs.

It can be observed from Figures 5b and 6b that the TIE inverse achieves a speedup (ratio of number of clock cycles of plain C inverse to TIE inverse) in the order of 10^4. The plain C inverse computation directly inverts to GF(2^8) (or GF(2^{16})) through elementary Galois field multiplication operations. On the other hand, the TIE inverse computation transforms each given GF(2^8) element into two GF(2^4) elements (or four GF(2^4) elements for GF(2^{16})) (see Section 3.3) and finds the inverse values through table lookups. From examining the inversion speed-up on the order of 10^4 closer, it can be observed from Figure 5b that the plain C inverse with lookup table requires about two orders of magnitude fewer clock cycles than the plain C inverse with elementary multiplication operations. The inversion with TIE, in turn, requires about two orders of magnitude fewer clock cycles than the plain C inverse with the lookup table. Thus, we conclude that the table lookup is responsible for about two orders of magnitude of the speedup, and TIE acceleration is responsible for the remaining two orders of magnitude of the inversion speedup.



(**a**) Encoder (**b**) Decoder

Figure 6. Speedups for different acceleration strategies with respect to the plain C implementation. For the decoding, we considered in part (**b**) the following speedups: "Inv., Mult." is number of clock cycles for "Plain C Inv." divided by number of clock cycles for "TIE Inv."; "Inv., LUT" is number of clock cycles for "Plain C Inv. LUT" divided by number of clock cycles for "TIE Inv."; "SIMD/FLIX vs. TIE" is number of clock cycles for ("TIE Inv" + "TIE MAC") divided by number of clock cycles for "SIMD/FLIX" (complete decoding); "SIMD/FLIX vs. Plain C" is number of clock cycles for ("Plain C Inv. LUT" + "Plain C MAC") divided by number of clock cycles for "SIMD/FLIX" (complete decoding).

The speedup for the matrix multiplication step of the RLNC decoding, i.e., the ratio of the number of clock cycles for the plain C MAC to the number of clock cycles for the TIE MAC in Figure 6b is equivalent to the corresponding MAC encoding speedup in Figure 6a. This is because the matrix multiplication step of the RLNC decoding has essentially the same computational complexity as the RLNC encoding. Notice that the clock cycle numbers for plain C MAC and TIE MAC in Figure 6b match the plain C and MAC clock cycle numbers in Figure 6a.

The SIMD and FLIX results in Figure 6b are for the complete decoding (matrix inversion followed by matrix multiplication) with the SIMD and FLIX TIEs. Complete decoding, conducted by a concatenation of the TIE inverse and the TIE MAC, would require roughly 250,000 clock cycles. The SIMD and FLIX accelerations reduce the computation time down to the order of 10,000 clock cycles. The SIMD and FLIX speedups reported in Figure 6b correspond to the aggregate of the number of clock cycles of the plain C inverse and the plain C MAC divided by the number of clock cycles for complete decoding with SIMD or FLIX. Note that a large portion of the SIMD and FLIX speedup is due to the speedup achieved by the table lookup of the TIE inverse, i.e., the left-most set of bars in Figure 6b.

Importantly, it can be observed from Figures 5 and 6 that 8-bit, i.e., GF(2^8), and 16-bit, i.e., GF(2^{16}), operations give essentially the same numbers of computing cycles and therefore, speedups. This is because the Tensilica LX5 can be flexibly configured to process 8-bit or 16-bit data. More specifically, the clock cycle evaluation considers the computation time (runtime) for one generation of a given size of $g \cdot m$ bytes. For a given acceleration strategy, the computation time is mainly governed by the density (level of parallelism) of the computing and the processor's memory interface. The processor's memory interface is able to handle 128 bits/cycle. For a given acceleration strategy, i.e., a given level of parallelism of the computing, the processor can be flexibly configured to conduct computations with the 128-bit data set with a similar efficiency to sixteen 8-bit values or eight 16-bit values. Thus, the clock cycle count for RLNC encoding or decoding of one generation of a fixed size in bytes is nearly the same with either 8-bit or 16-bit processing for a given acceleration strategy.

Close inspection of Figures 5b and 6b reveals that the 16-bit decoder gives slightly lower cycle counts and correspondingly slightly higher speedups than the 8-bit decoder. This is mainly due to the data transformations involved in the matrix inversion which can be executed slightly faster with larger registers.

Overall, we conclude from this evaluation of the computation cycles and speedups that the Xtensa LX5 with the SIMD and the FLIX TIE instructions achieves very substantial improvements compared to an Xtensa LX5 without any TIE. In order to put this RLNC computing performance of an ASIP in perspective, we compare it with a state-of-the-art fixed instruction processor in the next section.

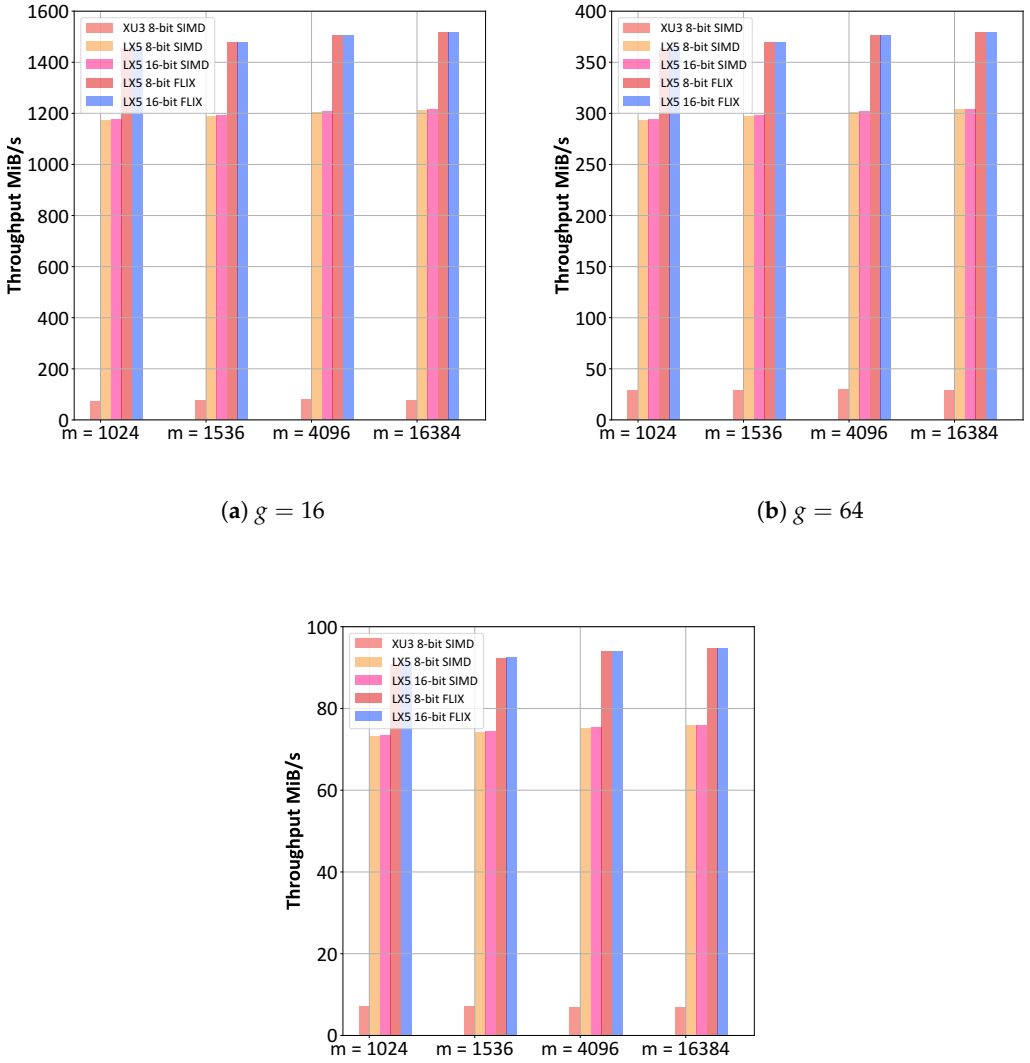
4.4. RLNC Encoding and Decoding Throughput

Figures 7 and 8 present the RLNC encoding and decoding throughput for various generation sizes (g) and packet sizes (m) in bytes. We compared the ODROID XU3 processing 8-bit, i.e., GF(2^8), operands with SIMD (NEON) [105] with the Xtensa LX5 processing 8-bit, i.e., GF(2^8), and 16-bit, i.e., GF(2^{16}), operands with the SIMD and FLIX TIEs. It can be observed from Figures 7 and 8 that the encoding and decoding throughputs are generally inversely proportional to the generation size (g). In particular, it can be observed from Figure 7a that the encoding throughput of around 1200–1500 MiB/s for $g = 16$ is reduced to approximately 300–375 MiB/s for $g = 64$ in Figure 7b. Increasing g by a factor of four to $g = 256$ in Figure 7c further reduces the encoding throughput by one-fourth. This encoding throughput scaling inversely with the generation size g is due to the proportional increase of the computational complexity as g source symbols need to be linearly combined through MAC operations, see [105] (Equation (3)) for details.

For decoding, it can be observed from Figure 8 that, generally, the same throughput reduction as for encoding occurs with an increasing generation size (g). Moreover, it can be observed from Figure 8 that there is a tendency for the decoding throughput to increase with the symbol size (m); this effect of m is most pronounced for small generation sizes (g) and diminishes for large g. In particular, it can be observed that for large symbol sizes (m), the decoding throughput approaches the encoding throughput observed in Figure 8. The decoding throughput is governed by the combination of the computational complexity for the matrix inversion and the matrix multiplication. With an increasing symbol size (m), the matrix multiplication complexity (which is proportional to m) dominates over the inversion complexity (which only depends on g). Thus, for large m, the relative impact of the matrix inversion diminishes and the decoding throughput becomes mainly governed by the matrix multiplication complexity.

It can also be observed from Figures 7 and 8 that similar to the clock cycle and speed-up results in Section 4.3, the 8-bit SIMD and FLIX LX5 give essentially the same throughputs as the 16-bit SIMD and FLIX LX5. Thus, the Xtensa LX5 can flexibly conduct RLNC encoding and decoding with 8-bit operands or 16-bit operands, i.e., in both GF(2^8) and GF(2^{16}), whereas 16-bit operation is very challenging for conventional fixed instruction set processors. Thus, the Xtensa LX5 readily accommodates GF(2^{16}) RLNC, which has very low linear dependencies [50].

Importantly, the results in Figures 7 and 8 demonstrate that RLNC encoding and decoding can be effectively accelerated with an ASIP. It can be observed from Figure 7 that an approximately 25 to 30 times higher RLNC encoding throughput occurs for small generation sizes (g) and a 7 to 10 times higher RLNC encoding throughput occurs for large g with the Xtensa LX5 ASIP compared to the ODROID XU3 fixed instruction set processor. Similarly, it can be observed from Figure 8 that the ASIP increases the RLNC decoding throughput by approximately 10 to 15 times compared to the fixed instruction processor. The ASIP achieves these RLNC encoding and decoding performance improvements through data processing parallelization with SIMD and FLIX TIE instructions translated into custom chip hardware.



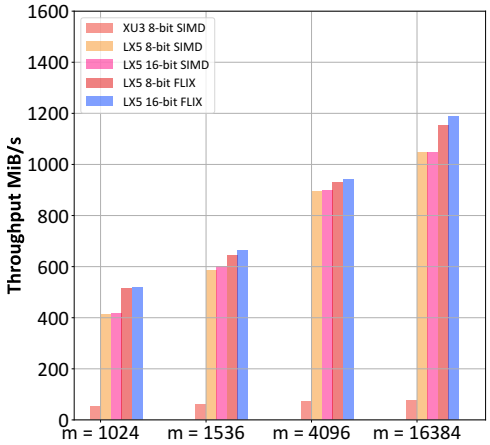
(**a**) $g = 16$ (**b**) $g = 64$

(**c**) $g = 256$

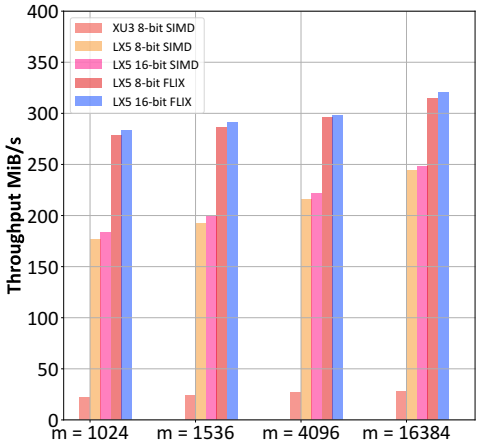
Figure 7. RLNC encoding throughput (in MiB/s = 2^{20} Byte/s) comparison between ODROID XU3 with SIMD (NEON) instructions and Xtensa LX5 core with SIMD and FLIX TIE instructions for various generation sizes (g) and symbol (packet) sizes (m).

4.5. Energy Consumption

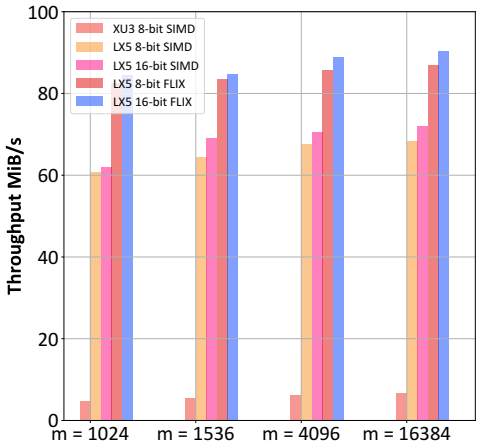
It can be observed from Table 2 that the Tensilica LX5 processor without TIE consumes over an order of magnitude less energy than the ODROID XU3. The FLIX TIE further reduces the energy consumption by over two orders of magnitude compared to the Tensilica LX5 without TIE. Thus, overall, the ASIP approach in the form of the Tensilica LX5 with FLIX TIE reduces the energy consumption by over three orders of magnitude compared to the fixed instruction-set processor in the form of the ODROID XU3.



(**a**) $g = 16$



(**b**) $g = 64$



(**c**) $g = 256$

Figure 8. RLNC decoding throughput (in MiB/s = 2^{20} Byte/s) comparison between ODROID XU3 with SIMD (NEON) instructions and Xtensa LX5 core with SIMD and FLIX TIE instructions for various generation sizes g and symbol (packet) sizes m in bytes.

Table 2. Energy consumption for RLNC encoding followed by RLNC decoding.

Operation	Energy Consumption (J/MiB)
ODROID XU3	4.3
Xtensa LX5 Plain C code w/o TIE	0.17
Xtensa LX5 FLIX TIE	0.0014

We note that the ODROID XU3 platform has four main power-consuming components: the Cortex-A7 CPU, the Cortex-A15 CPU, the DDR3 RAM, and the GPU. We operated the ODROID XU3 in single thread mode, i.e., we utilized only a single thread (core) of the A-15 quad-core processor. Thus, the power consumption contributions of the other components were essentially negligible [107]. The energy results for the Xtensa platform consider all Xtensa subsystems, including the memory subsystem. In particular, the Xtensa LX5 completed the RLNC encoding and decoding with an SRAM scratchpad module that is positioned next to the processor on the same chip.

The energy consumption reductions with the Tensilica LX5 with FLIX TIE were due to the high energy-efficiency of computations with application-specific configurable hardware. Pre-configured fixed instruction set processors generally consume much more energy than specialized hardware [31–33]. This case study quantified this energy consumption reduction to be over three orders of magnitude for the RLNC encoding and decoding.

5. Conclusions

This case study investigated the use of an application-specific instruction-set processor (ASIP) for random linear network coding (RLNC). RLNC is highly useful for protecting wireless communication in a wide range of settings, including Internet of Things (IoT) settings where small devices need to communicate reliably. RLNC requires computationally demanding matrix multiplications and inversions, limiting the use of RLNC in small battery powered IoT nodes.

This case study focused on the Tensilica Xtensa LX5 processor which can accelerate computations in hardware though configurable tensilica instruction-set extensions (TIEs). We outlined the development of TIEs for the multiply-and-accumulate (MAC) function, which is at the center of the matrix multiplication in Galois fields required for RLNC. Furthermore, we outlined acceleration through TIEs that implement single instruction multiple data (SIMD) and flexible-length instruction extension (FLIX) strategies. We comprehensively evaluated the developed TIE hardware acceleration strategies for RLNC by measuring the die size (gate count), clock cycles required for RLNC encoding and decoding, as well as RLNC encoding and decoding throughput and the consumed energy. We found that the ASIP with TIEs increases the RLNC encoding and decoding throughput by roughly an order of magnitude while reducing the energy consumption by over three orders of magnitude compared to a state-of-the-art preconfigured fixed instruction set processor. These performance comparisons provide insightful benchmark data for designers of minuscule IoT devices that need to communicate reliably over wireless links. Due to the pronounced reduction in consumed energy, the use of an ASIP can dramatically extend the battery life while enhancing the communication performance compared to general purpose processors.

There are interesting directions for future research on boosting computation performance for RLNC encoding and decoding. The present study considered a single core ASIP. Future research should examine multicore ASIPs, such as the Tomahawk processor [104]. Extending the SIMD and FLIX TIE instructions to such a multi-core reconfigurable platform may achieve additional performance increases while still keeping the energy consumption low. However, a multi-core platform will likely increase the programming complexity as the parallel computation model should efficiently exploit the hardware resources of the multiple cores. Another research direction is to evaluate the computing performance for novel flexible forms of RLNC, such as Fulcrum RLNC [108] which employs two layers of RLNC for a given generation of source symbols to flexibly reach devices with heterogeneous capabilities.

Author Contributions: Conceptualization, R.S., S.W., M.H., S.P., F.H.P.F., G.F. and M.R.; Data curation, J.A., S.W. and M.H.; Funding acquisition, F.H.P.F. and G.F.; Methodology, S.P. and M.R.; Project administration, F.H.P.F.; Resources, J.C.; Supervision, F.H.P.F. and G.F.; Validation, S.P.; Writing-original draft, J.A., R.S. and M.R.; Writing-review & editing, J.A., S.W., M.H., J.C., F.H.P.F. and M.R.

Acknowledgments: We are grateful to Sebastian Haas of the TU Dresden Vodafone Chair Mobile Communication Systems for assistance with the configuration of the Xtensa Xplorer.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bassoli, R.; Marques, H.; Rodriguez, J.; Shum, K.W.; Tafazolli, R. Network coding theory: A survey. *IEEE Commun. Surv. Tutor.* **2013**, *15*, 1950–1978. [CrossRef]
2. Farooqi, M.Z.; Tabassum, S.M.; Rehmani, M.H.; Saleem, Y. A survey on network coding: From traditional wireless networks to emerging cognitive radio networks. *J. Netw. Comput. Appl.* **2014**, *46*, 166–181. [CrossRef]

3. Ho, T.; Médard, M.; Koetter, R.; Karger, D.R.; Effros, M.; Shi, J.; Leong, B. A random linear network coding approach to multicast. *IEEE Trans. Inf. Theory* **2006**, *52*, 4413–4430. [CrossRef]
4. Amanowicz, M.; Krygier, J. On applicability of network coding technique for 6LoWPAN-based sensor networks. *Sensors* **2018**, *18*, 1–20. [CrossRef] [PubMed]
5. Chau, P.; Bui, T.D.; Lee, Y.; Shin, J. Efficient data uploading based on network coding in LTE-Advanced heterogeneous networks. In Proceedings of the 2017 19th International Conference on Advanced Communication Technology (ICACT), Bongpyeong, Korea, 19–22 February 2017; pp. 252–257.
6. Ji, X.; Wang, A.; Li, C.; Ma, C.; Peng, Y.; Wang, D.; Hua, Q.; Chen, F.; Fang, D. ANCR–An adaptive network coding routing scheme for WSNs with different-success-rate links. *Appl. Sci.* **2017**, *7*, 809. [CrossRef]
7. Kartsakli, E.; Antonopoulos, A.; Alonso, L.; Verikoukis, C. A cloud-assisted random linear network coding medium access control protocol for healthcare applications. *Sensors* **2014**, *14*, 4806–4830. [CrossRef] [PubMed]
8. Kang, H.; Yoo, H.; Kim, D.; Chung, Y.S. CANCORE: Context-Aware Network COded REpetition for VANETs. *IEEE Access* **2017**, *5*, 3504–3512. [CrossRef]
9. Li, X.; Chang, Q.; Xu, Y. Queueing characteristics of the best effort network coding strategy. *IEEE Access* **2016**, *4*, 5990–5997. [CrossRef]
10. Liu, J.S.; Lin, C.H.R.; Tsai, J. Delay and energy tradeoff in energy harvesting multi-hop wireless networks with inter-session network coding and successive interference cancellation. *IEEE Access* **2017**, *5*, 544–564. [CrossRef]
11. Khamfroush, H.; Lucani, D.E.; Pahlevani, P.; Barros, J. On optimal policies for network-coded cooperation: theory and implementation. *IEEE J. Sel. Areas Commun.* **2015**, *33*, 199–212. [CrossRef]
12. Nessa, A.; Kadoch, M.; Rong, B. Fountain coded cooperative communications for LTE-A connected heterogeneous M2M network. *IEEE Access* **2016**, *4*, 5280–5292. [CrossRef]
13. Pahlevani, P.; Hundebøll, M.; Pedersen, M.V.; Lucani, D.E.; Charaf, H.; Fitzek, F.H.; Bagheri, H.; Katz, M. Novel concepts for device-to-device communication using network coding. *IEEE Commun. Mag.* **2014**, *52*, 32–39. [CrossRef]
14. Peng, Y.; Wang, X.; Guo, L.; Wang, Y.; Deng, Q. An efficient network coding-based fault-tolerant mechanism in WBAN for smart healthcare monitoring systems. *Appl. Sci.* **2017**, *7*, 817. [CrossRef]
15. Vukobratovic, D.; Tassi, A.; Delic, S.; Khirallah, C. Random linear network coding for 5G mobile video delivery. *Information* **2018**, *9*, 72. [CrossRef]
16. Wang, H.; Wang, S.; Bu, R.; Zhang, E. A novel cross-layer routing protocol based on network coding for underwater sensor networks. *Sensors* **2017**, *17*, 1821. [CrossRef] [PubMed]
17. Yang, Y.; Chen, W.; Li, O.; Hanzo, L. Joint rate and power adaptation for amplify-and-forward two-way relaying relying on analog network coding. *IEEE Access* **2016**, *4*, 2465–2478. [CrossRef]
18. Yang, Y.; Chen, W.; Li, O.; Liu, Q.; Hanzo, L. Truncated-ARQ aided adaptive network coding for cooperative two-way relaying networks: Cross-layer design and analysis. *IEEE Access* **2016**, *4*, 9361–9376. [CrossRef]
19. Zhang, G.; Cai, S.; Xiong, N. The application of social characteristic and L1 optimization in the error correction for network coding in wireless sensor networks. *Sensors* **2018**, *18*, 450. [CrossRef] [PubMed]
20. Joshi, G.; Liu, Y.; Soljanin, E. On the delay-storage trade-off in content download from coded distributed storage systems. *IEEE J. Sel. Areas Commun.* **2014**, *32*, 989–997. [CrossRef]
21. Kumar, N.; Zeadally, S.; Rodrigues, J.J.P.C. QoS-aware hierarchical web caching scheme for online video streaming applications in Internet-based vehicular ad hoc networks. *IEEE Trans. Ind. Electron.* **2015**, *62*, 7892–7900. [CrossRef]
22. Sipos, M.; Gahm, J.; Venkat, N.; Oran, D. Erasure coded storage on a changing network: The untold story. In Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM), Washington, DC, USA, 4–8 December 2016; pp. 1–6.
23. Wang, L.; Wu, H.; Han, Z. Wireless distributed storage in socially enabled D2D communications. *IEEE Access* **2016**, *4*, 1971–1984. [CrossRef]
24. Chau, P.; Shin, J.; Jeong, J. Distributed systematic network coding for reliable content uploading in wireless multimedia sensor networks. *Sensors* **2018**, *18*, 1824, doi:10.3390/s18061824. [CrossRef] [PubMed]
25. Chen, B.W.; Ji, W.; Jiang, F.; Rho, S. QoE-enabled big video streaming for large-scale heterogeneous clients and networks in smart cities. *IEEE Access* **2016**, *4*, 97–107. [CrossRef]

26. Gkantsidis, C.; Rodriguez, P.R. Network coding for large scale content distribution. In Proceedings of the 2005 24th Annual Joint Conference of the IEEE Computer and Communications Societies, Miami, FL, USA, 13–17 March 2005; pp. 2235–2245.
27. Li, B.; Niu, D. Random network coding in peer-to-peer networks: from theory to practice. *Proc. IEEE* **2011**, *99*, 513–523.
28. Maboudi, B.; Sehat, H.; Pahlevani, P.; Lucani, D.E. On the performance of the cache coding protocol. *Information* **2018**, *9*, 62. [CrossRef]
29. Rekab-Eslami, M.; Esmaeili, M.; Gulliver, T.A. Multicast convolutional network codes via local encoding kernels. *IEEE Access* **2017**, *5*, 6464–6470. [CrossRef]
30. Skevakis, E.; Lambadaris, I. Optimal control for network coding broadcast. In Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM), Washington, DC, USA, 4–8 December 2016; pp. 1–6.
31. Haas, S.; Karnagel, T.; Arnold, O.; Laux, E.; Schlegel, B.; Fettweis, G.; Lehner, W. HW/SW-database-codesign for compressed bitmap index processing. In Proceedings of the 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), London, UK, 6–8 July 2016; pp. 50–57.
32. Arnold, O.; Haas, S.; Fettweis, G.; Schlegel, B.; Kissinger, T.; Karnagel, T.; Lehner, W. HASHI: An application specific instruction set extension for hashing. In Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures at International Conference on Very Large Data Bases (VLDB), Hangzhou, China, 1–5 September 2014; pp. 25–33.
33. Arnold, O.; Neumaerker, F.; Fettweis, G. L2_ISA++: Instruction set architecture extensions for 4G and LTE-advanced MPSoCs. In Proceedings of the 2014 International Symposium on System-on-Chip (SoC), Tampere, Finland, 28–29 October 2014; pp. 1–8.
34. Gonzalez, R.E. Xtensa: A configurable and extensible processor. *IEEE Micro* **2000**, *20*, 60–70. [CrossRef]
35. Maymounkov, P.; Harvey, N.J.; Lun, D.S. Methods for efficient network coding. In Proceedings of the 44th Annual Allerton Conference on Communication, Control, and Computing, Monticello, Illinois, 27–29 September 2006; pp. 482–491.
36. Aboutorab, N.; Sadeghi, P.; Sorour, S. Enabling a tradeoff between completion time and decoding delay in instantly decodable network coded systems. *IEEE Trans. Commun.* **2014**, *62*, 1296–1309. [CrossRef]
37. Douik, A.; Karim, M.S.; Sadeghi, P.; Sorour, S. Delivery time reduction for order-constrained applications using binary network codes. In Proceedings of the 2016 IEEE Wireless Communications and Networking Conference, Doha, Qatar, 3–6 April 2016; pp.1–6.
38. Douik, A.; Sorour, S.; Al-Naffouri, T.; Alouini, S. Instantly decodable network coding: From centralized to device-to-device communications. *IEEE Commun. Surv. Tutor.* **2017**, *19*, 1201–1224. [CrossRef]
39. Li, X.; Wang, C.C.; Lin, X. Optimal immediately-decodable inter-session network coding (IDNC) schemes for two unicast sessions with hard deadline constraints. In Proceedings of the 2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton), Monticello, IL, USA, 28–30 September 2011; pp. 784–791.
40. Qureshi, J.; Foh, C.H.; Cai, J. Online XOR packet coding: Efficient single-hop wireless multicasting with low decoding delay. *Comput. Commun.* **2014**, *39*, 65–77. [CrossRef]
41. Sorour, S.; Valaee, S. Completion delay minimization for instantly decodable network codes. *IEEE/ACM Trans. Netw.* **2015**, *23*, 1553–1567. [CrossRef]
42. Yu, M.; Aboutorab, N.; Sadeghi, P. From instantly decodable to random linear network coded broadcast. *IEEE Trans. Commun.* **2014**, *62*, 3943–3955. [CrossRef]
43. Barros, J.; Costa, R.A.; Munaretto, D.; Widmer, J. Effective delay control in online network coding. In Proceedings of the IEEE INFOCOM, Rio de Janeiro, Brazil, 19–25 April 2009; pp. 208–216.
44. Costa, R.A.; Munaretto, D.; Widmer, J.; Barros, J. Informed network coding for minimum decoding delay. In Proceedings of the 2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, Atlanta, GA, USA, 29 September–2 October 2008; pp. 80–91.
45. Fu, A.; Sadeghi, P.; Médard, M. Dynamic rate adaptation for improved throughput and delay in wireless network coded broadcast. *IEEE/ACM Trans. Netw.* **2014**, *22*, 1715–1728. [CrossRef]
46. Sadeghi, P.; Shams, R.; Traskov, D. An optimal adaptive network coding scheme for minimizing decoding delay in broadcast erasure channels. *EURASIP J. Wirel. Commun. Netw.* **2010**, *2010*, 1–14. [CrossRef]
47. Sorour, S.; Valaee, S. An adaptive network coded retransmission scheme for single-hop wireless multicast broadcast services. *IEEE/ACM Trans. Netw.* **2011**, *19*, 869–878. [CrossRef]

48. Yeow, W.L.; Hoang, A.T.; Tham, C.K. Minimizing delay for multicast-streaming in wireless networks with network coding. In Proceedings of the IEEE INFOCOM, Rio de Janeiro, Brazil, 19–25 April 2009; pp. 190–198.
49. Heide, J.; Pedersen, M.V.; Fitzek, F.H.; Larsen, T. Cautious view on network coding-From theory to practice. *J. Commun. Netw.* **2008**, *10*, 403–411. [CrossRef]
50. Heide, J.; Pedersen, M.V.; Fitzek, F.H.; Médard, M. On code parameters and coding vector representation for practical RLNC. In Proceedings of the 2011 IEEE International Conference on Communications (ICC), Kyoto, Japan, 5–9 June 2011; pp. 1–5.
51. Shah-Mansouri, V.; Srinivasan, S. Retransmission scheme for intra-session linear network coding in wireless networks. *Int. J. Adv. Intell. Paradigms* **2017**, *9*, 326–346. [CrossRef]
52. Sundararajan, J.K.; Shah, D.; Medard, M.; Sadeghi, P. Feedback-Based Online Network Coding. *IEEE Trans. Inf. Theory* **2017**, *63*, 6628–6649. [CrossRef]
53. Zeng, D.; Guo, S.; Jin, H.; Leung, V. Segmented network coding for stream-like applications in delay tolerant networks. In Proceedings of the 2011 IEEE Global Telecommunications Conference—GLOBECOM, Kathmandu, Nepal, 5–9 December 2011; pp. 1–5.
54. Krigslund, J.; Fitzek, F.; Pedersen, M.V. On the combination of multi-layer source coding and network coding for wireless networks. In Proceedings of the 2013 IEEE 18th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Berlin, Germany, 25–27 September 2013; pp. 1–6.
55. Matsuzono, K.; Asaeda, H.; Turletti, T. Low latency low loss streaming using in-network coding and caching. In Proceedings of the IEEE INFOCOM 2017—IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
56. Swamy, V.N.; Rigge, P.; Ranade, G.; Sahai, A.; Nikolić, B. Network coding for high-reliability low-latency wireless control. In Proceedings of the 2016 IEEE Wireless Communications and Networking Conference, Doha, Qatar, 3–6 April 2016; pp. 1–7.
57. Yu, K.; Yue, J.; Lin, Z.; Åkerberg, J.; Björkman, M. Achieving reliable and efficient transmission by using network coding solution in industrial wireless sensor networks. In Proceedings of the 2016 IEEE 25th International Symposium on Industrial Electronics (ISIE), Santa Clara, CA, USA, 8–10 June 2016; pp. 1162–1167.
58. Júnior, N.d.S.R.; Tavares, R.C.; Vieira, M.A.; Vieira, L.F.; Gnawali, O. CodeDrip: Improving data dissemination for wireless sensor networks with network coding. *Ad Hoc Netw.* **2017**, *54*, 42–52. [CrossRef]
59. Cloud, J.; Medard, M. Multi-Path Low Delay Network Codes. In Proceedings of the 2016 IEEE Global Communications Conference (GLOBECOM), Washington, DC, USA, 4–8 December 2016; pp. 1–7.
60. Gabriel, F.; Wunderlich, S.; Pandi, S.; Fitzek, F.H.; Reisslein, M. Caterpillar RLNC With Feedback (CRLNC-FB): Reducing Delay in Selective Repeat ARQ Through Coding. *IEEE Access* **2018**, *6*, 44787–44802. [CrossRef]
61. Garcia-Saavedra, A.; Karzand, M.; Leith, D.J. Low delay random linear coding and scheduling over multiple interfaces. *IEEE Trans. Mob. Comput.* **2017**, *16*, 3100–3114. [CrossRef]
62. Karzand, M.; Leith, D.J.; Cloud, J.; Medard, M. Design of FEC for Low Delay in 5G. *IEEE J. Sel. Areas Commun.* **2017**, *35*, 1783–1793. [CrossRef]
63. Pandi, S.; Gabriel, F.; Cabrera, J.; Wunderlich, S.; Reisslein, M.; Fitzek, F.H.P. PACE: Redundancy Engineering in RLNC for Low-Latency Communication. *IEEE Access* **2017**, *5*, 20477–20493. [CrossRef]
64. Wunderlich, S.; Gabriel, F.; Pandi, S.; Fitzek, F.H.; Reisslein, M. Caterpillar RLNC (CRLNC): A practical finite sliding window RLNC approach. *IEEE Access* **2017**, *5*, 20183–20197. [CrossRef]
65. Choi, S.M.; Lee, K.; Park, J. Massive parallelization for random linear network coding. *Appl. Math. Inf. Sci.* **2015**, *9*, 571–578.
66. Gan, X.B.; Shen, L.; Wang, Z.Y.; Lai, X.; Zhu, Q. Parallelizing network coding using CUDA. *Adv. Mater. Res.* **2011**, *186*, 484–488. [CrossRef]
67. Kim, M.; Park, K.; Ro, W.W. Benefits of using parallelized non-progressive network coding. *J. Netw. Comput. Appl.* **2013**, *36*, 293–305. [CrossRef]
68. Lee, S.; Ro, W.W. Accelerated network coding with dynamic stream decomposition on graphics processing unit. *Comput. J.* **2012**, *55*, 21–34. [CrossRef]
69. Park, J.S.; Baek, S.J.; Lee, K. A highly parallelized decoder for random network coding leveraging GPGPU. *Comput. J.* **2014**, *57*, 233–240. [CrossRef]

70. Shojania, H.; Li, B. Pushing the envelope: Extreme network coding on the GPU. In Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), Montreal, QC, Canada, 22–26 June 2009; pp. 490–499.
71. Shojania, H.; Li, B.; Wang, X. Nuclei: GPU-accelerated many-core network coding. In Proceedings of the IEEE Infocom, Rio de Janeiro, Brazil, 19–25 April 2009; pp. 459–467.
72. Shojania, H.; Li, B. Tenor: making coding practical from servers to smartphones. In Proceedings of the 18th ACM international conference on Multimedia , Firenze, Italy, 25–29 October 2010; pp. 45–54.
73. Rizvi, S.T.H.; Cabodi, G.; Patti, D.; Francini, G. GPGPU accelerated deep object classification on a heterogeneous mobile platform. *Electronics* **2016**, *5*, 88. [CrossRef]
74. Chen, D.; Cong, J.; Gurumani, S.; Hwu, W.m.; Rupnow, K.; Zhang, Z. Platform choices and design demands for IoT platforms: Cost, power, and performance tradeoffs. *IET Cyber Phys.Syst. Theory Appl.* **2016**, *1*, 70–77. [CrossRef]
75. Rizk, M.; Baghdadi, A.; Jézéquel, M.; Mohanna, Y.; Atat, Y. Design and prototyping flow of flexible and efficient NISC-Based architectures for MIMO Turbo equalization and demapping. *Electronics* **2016**, *5*, 1–27. [CrossRef]
76. Shojania, H.; Li, B. Parallelized progressive network coding with hardware acceleration. In Proceedings of the Fifteenth IEEE International Workshop on Quality of Service, Evanston, IL, USA, 21–22 June 2007; pp. 47–55.
77. Hernández Marcano, N.J.; Sørensen, C.W.; Cabrera G, J.A.; Wunderlich, S.; Lucani, D.E.; Fitzek, F.H. On goodput and energy measurements of network coding schemes in the Raspberry Pi. *Electronics* **2016**, *5*, 1–27, doi:10.3390/electronics5040066. [CrossRef]
78. Sørensen, C.W.; Hernández Marcano, N.J.; Cabrera Guerrero, J.A.; Wunderlich, S.; Lucani, D.E.; Fitzek, F.H. Easy as Pi: A network coding Raspberry Pi testbed. *Electronics* **2016**, *5*, 1–25, doi:10.3390/electronics5040067. [CrossRef]
79. Arrobo, G.E.; Gitlin, R.D. Minimizing energy consumption for cooperative network and diversity coded sensor networks. In Proceedings of the 2014 Wireless Telecommunications Symposium, Washington, DC, USA, 9–11 April 2014; pp. 1–7.
80. Fiandrotti, A.; Bioglio, V.; Grangetto, M.; Gaeta, R.; Magli, E. Band codes for energy-efficient network coding with application to P2P mobile streaming. *IEEE Trans. Multimedia* **2014**, *16*, 521–532. [CrossRef]
81. Angelopoulos, G.; Médard, M.; Chandrakasan, A. Energy-aware hardware implementation of network coding. In Proceedings of the International Conference on Research in Networking, Valencia, Spain, 13 May 2011; pp. 137–144.
82. *Tensilica Xtensa LX5 Customizable DPU*; Cadence Design Systems: San Jose, CF, USA, 2013.
83. Wolkerstorfer, J.; Oswald, E.; Lamberger, M. An ASIC implementation of the AES SBoxes. In Proceedings of the CT-RSA 2002: The Cryptographers' Track at the RSA Conference 2002, San Jose, CA, USA, 18–22 February 2002; pp. 67–78.
84. Satoh, A.; Morioka, S.; Takano, K.; Munetoh, S. A compact Rijndael hardware architecture with S-box optimization. In Proceedings of the ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, 9–13 December 2001; pp. 239–254.
85. Standaert, F.X.; Rouvroy, G.; Quisquater, J.J.; Legat, J.D. Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs. In Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems, Cologne, Germany, 8–10 September 2003; pp. 334–350.
86. Wang, C.C.; Truong, T.K.; Shao, H.M.; Deutsch, L.J.; Omura, J.K.; Reed, I.S. VLSI architectures for computing multiplications and inverses in GF$(2m)$. *IEEE Trans. Comput.* **1985**, *34*, 709–717. [CrossRef] [PubMed]
87. Guajardo, J.; Paar, C. Efficient algorithms for elliptic curve cryptosystems. In Proceedings of the 17th Annual International Cryptology Conference, Barbara, California, USA, 17–21 August 1997; pp. 342–356.
88. Paar, C. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*; VDI-Verlag: Düsseldorf, Germany, 1994.
89. Sghaier, A.; Zeghid, M.; Massoud, C.; Mahchout, M. Design and Implementation of Low Area/Power Elliptic Curve Digital Signature Hardware Core. *Electronics* **2017**, *6*, 46. [CrossRef]
90. Mangard, S.; Aigner, M.; Dominikus, S. A highly regular and scalable AES hardware architecture. *IEEE Trans. Comput.* **2003**, *52*, 483–491. [CrossRef]
91. Fraleigh, J.B.; Beauregard, R. *Linear Algebra*, 3rd ed.; Pearson: London, UK, 2013.

92. Quinnell, E.; Swartzlander, E.; Lemonds, C. Floating-point fused multiply-add architectures. In Proceedings of the 2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 4–7 November 2007; pp. 331–337.
93. *MCF5307 ColdFire Integrated Microprocessor User Manual*; Technical report for Freescale Semiconductor, Inc.: Austin, TX, USA, 2016.
94. Gimmestad, B.J. The Russian Peasant Multiplication Algorithm: A Generalization. *Math. Gazette* **1991**, *75*, 169–171. [CrossRef]
95. Chen, H.; Liu, Z.; Liu, S.; Ma, S. An efficient vector memory unit for SIMD DSP. In Proceedings of the CCF National Conf. on Computer Engineering and Technology (NCCET), Guiyang, China, 29 July–1 August 2014; pp. 1–11.
96. Jutla, C.S.; Kumar, V.; Rudra, A. *On the Circuit Complexity of Isomorphic Galois Field Transformations*; No. 93; Electronic Colloquium on Computational Complexity: Potsdam, Germany, 2012.
97. Tensilica Instruction Extension. Available online: https://en.wikipedia.org/wiki/Tensilica_Instruction_Extension (accessed on 13 February 2017).
98. Increasing Processor Computational Performance with More Instruction Parallelism. Available online: https://ip.cadence.com/uploads/927/TIP$_$WP$_$FLIX$_$FINAL-pdf (accessed on 25 July 2018).
99. Nurmi, J. *Processor Design: System-On-Chip Computing for ASICs and FPGAs*; Springer: Dordrecht, The Netherlands, 2007.
100. Oklobdzija, V. *The Computer Architecture: Handbook*; CRC Press: Boca Raton, FL, USA, 2002.
101. Gregg, J. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets*; Wiley-IEEE Press: Hoboken, NJ, USA, 1998.
102. Hennessy, J.; Patterson, D. *Computer Architecture: A Quantitative Approach*, 4th ed; Morgan Kaufmann: Burlington, MA, USA, 2006.
103. Gropp, W. Lecture 14: Discussing Speedup. Parallel @ Illinois. Available online: http://wgropp.cs.illinois.edu/courses/cs598s16/lectures/lecture14.pdf (accessed on 4 July 2018).
104. Arnold, O.; Matus, E.; Noethen, B.; Winter, M.; Limberg, T.; Fettweis, G. Tomahawk: Parallelism and heterogeneity in communications signal processing MPSoCs. *ACM Trans. Embedded Comput. Syst.* **2014**, *13*, 107. [CrossRef]
105. Wunderlich, S.; Cabrera, J.; Fitzek, F.H.; Reisslein, M. Network coding in heterogeneous multicore IoT nodes with DAG scheduling of parallel matrix block operations. *IEEE Internet Things J.* **2017**, *4*, 917–933. [CrossRef]
106. Halbutogullari, A.; Koc, C.K. Mastrovito multiplier for general irreducible polynomials. *IEEE Trans. Comput.* **2000**, *49*, 503–518. [CrossRef]
107. Gsching, P. Energy Efficient Processors–ARM big.LITTLE Technology. Available online: http://www.ziti.uniheidelberg.de/ziti/uploads/ce_group/seminar/2015-Philipp_Gsching.pdf (accessed on January 2016).
108. Nguyen, V.; Nguyen, G.T.; Gabriel, F.; Lucani, D.E.; Fitzek, F.H. Integrating sparsity into Fulcrum codes: Investigating throughput, complexity and overhead. In Proceedings of the 2018 IEEE International Conference on Communications Workshops (ICC Workshops), Kansas City, MO, USA, 20–24 May 2018.