cryptography

MDPI

Article

NLU-V: A Family of Instruction Set Extensions for Efficient Symmetric Cryptography on RISC-V

Hakan Uzuner and Elif Bilge Kavun *

Faculty of Computer Science and Mathematics, University of Passau, 94032 Passau, Germany; uzunerh@outlook.de
* Correspondence: elif.kavun@uni-passau.de; Tel.: +49-851-509-4470

Abstract: Cryptographic primitives nowadays are not only implemented in high-performance systems but also in small-scale systems, which are increasingly powered by open-source processors, such as RISC-V. In this work, we leverage RISC-V's modular base instruction set and architecture to propose a generic instruction set extension (ISE) for symmetric cryptography. We adapt the work from Engels et al. in ARITH'13, the non-linear/linear instruction set extension (NLU), which presents a generic hardware/software co-design solution for efficient symmetric crypto implementations through a hardware unit extending the 8-bit AVR instruction set. These new instructions realize non-linear and linear layers, which are widely used to implement the block ciphers in symmetric cryptography. Our proposal modifies and extends the NLU instructions to a 32-bit RISC-V architecture; hence, we call the proposed ISE 'NLU-V'. The proposed architecture is integrated into the open-source RISC-V implementation 'Icicle' and synthesized on a Xilinx Kintex-7 XC7K160T FPGA. The area overhead for the proposed NLU-V ISE is 1088 slice registers and 4520 LUTs. As case studies, the PRESENT and AES block ciphers are implemented using the new ISE on RISC-V in assembly. Our evaluation metric to showcase the performance gain, Z 'time-area-product (TAP)' (the execution time in clock cycles times code memory consumption), reflects the impact of the proposed family of instructions on the performance of the cipher implementations. The simulations show that the NLU-V achieves 89% gain for PRESENT and 68% gain for AES. Further, the NLU-V requires 44% less lines of code for the PRESENT and 23% less for the AES implementation.

Keywords: symmetric cryptography; block ciphers; instruction set extension; RISC-V; reconfigurable hardware; FPGA

Citation: Uzuner, H.; Kavun, E.B. NLU-V: A Family of Instruction Set Extensions for Efficient Symmetric Cryptography on RISC-V. *Cryptography* **2024**, *8*, 9. https://doi.org/10.3390/cryptography8010009

Academic Editors: Josef Pieprzyk and Jim Plusquellic

Received: 5 December 2023
Revised: 19 February 2024
Accepted: 26 February 2024
Published: 29 February 2024

1. Introduction

The ever-growing development in novel technologies makes our daily lives easier while bringing security and privacy problems. Researchers have made continuous efforts to develop security measures against new and known threats. However, the existing security measures are generally not directly applicable to all systems as different requirements may exist for each. For instance, implementing traditional cryptography schemes for security mechanisms in embedded systems is not a trivial task. Most of the software-only approaches easily become very costly for such systems as the majority of symmetric cryptography schemes use specific substitution and permutation layers, which are run in many rounds [1,2]. Also, having inefficient or too generic hardware solutions may easily increase the area and power consumption. Hence, a hardware–software co-design approach promises an efficient implementation option for such applications.

The non-linear/linear instruction (NLU) set extension (ISE), proposed by Engels et al. at the 21st ARITH Symposium in 2013 (ARITH21) as an ISE for 8-bit AVR microprocessors [3,4], serves as a flexible hardware–software co-design for the implementation of symmetric block ciphers on embedded processors. The NLU tries to find the perfect balance between hardware and software utilization for the best execution performance with

minimal hardware investment. It can speed up cipher implementations in software by pushing the computation-heavy operations to the hardware level as a novel ISE.

RISC-V is an open-source and modular instruction set architecture (ISA), which has attracted huge attention in recent years. Thanks to its modular and open-source characteristics, one can extend its ISA with novel cryptographic ISEs. However, despite a few existing proposals in the literature [5–7], there is still room for novel cryptographic solutions for the RISC-V platform.

Because the NLU was not initially developed to run on a 32-bit architecture like RISC-V, the goal of our study is to extend and adapt the NLU to work on a 32-bit RISC-V environment. The 32-bit version of the NLU, namely, NLU-V (should be read similar to RISC-V—'NLU-five'), aims to provide an improved performance for cryptographic operations implemented on widely-used 32-bit RISC-V open-source processors. Our contributions in this work can be listed as the following.

- The extension and adaptation of the 8-bit NLU ISE approach to 32 bits.
- A novel cryptographic ISE for the open-source RISC-V processor.
- Improved performance for block ciphers on 32-bit RISC-V using the novel ISE, which is showcased through case studies.
- Additional AES sBox (substitution layer) support for the non-linear instruction of the NLU.
- RISC-V GNU compiler toolchain adjustments to support the proposed RISC-V ISE (which was not provided in the previous NLU work).
- Publicly available NLU-V implementation, including the RISC-V core and the modified RISC-V GNU compiler toolchain.

In the following sections, we present our approach on adapting the 8-bit structure to 32 bits and give the details of the hardware units realizing the block cipher operations. Furthermore, we test our work by implementing the ISO-standard PRESENT cipher [1] and NIST-standard AES cipher [2] in the RISC-V assembly language using the RISC-V core instructions as well as new NLU-V instructions. We then compare the results in order to evaluate the performance of the NLU-V unit and reflect the impact of our proposed architecture through these case studies.

2. Related Work and Background

In this paper, we extend the NLU ISE developed by Engels et al., which was presented at ARITH21 in 2013 [3]. In their work, the authors introduce the NLU hardware unit, which is capable of computing non-linear and linear operations to introduce confusion and diffusion to block and stream ciphers in cryptography. Especially, the block ciphers in symmetric cryptography are widely used in many security applications, which is also reflected by the proposal of many different designs targeting different metrics, such as Clefia [8], Square [9], HIGHT [10], KATAN and KTANTAN [11], KLEIN [12], RC6TM [13], mCrypton [14], LED [15], Piccolo [16], SERPENT [17], PRESENT [1], and AES [2]. This wide utilization makes their efficient implementation an even more important task. The original NLU, which inspired this paper, aims to balance hardware and software utilization for implementation efficiency.

The non-linear and linear operations in block ciphers are encapsulated in the NLU together with inputs and modules to control the computations. The authors have presented the advantages of the NLU in their work based on Atmel's 8-bit AVR micro-controller [4] environment. To reliably interpret their results in terms of hardware and software efficiency, they compute the measure of the time-area-product (TAP) of several cipher implementations and compare those with and without the NLU ISE. In this context, TAP is interpreted as the execution time in clock cycles times code memory consumption. The authors of NLU show that they achieve a gain between 20 and 68% in terms of this metric. Also, they achieve a significant reduction in terms of lines of code for the respective implementations of the ciphers.

RISC-V is a 32- or 64-bit ISA following a modular approach, where several standard and custom instructions can be chosen as seen fit for the application at hand [18–20]. The base integer IS (RVI) introduces the most basic arithmetic instructions, 'add' and 'subtract', and some logical instructions, for example, 'AND' and 'OR', and some shift instructions. Further, RISC-V was developed with several standard extensions: the RISC-V multiply extension (RVM) for standard integer multiplication and division and the RISC-V floating-point extension (RVF) for standard floating-point operations, to name a few. This modular approach makes it easier for developers to pick the desired standard extensions that are needed for their proprietary project while keeping the architecture as slim as possible by leaving out unnecessary extensions. Similarly, a developer can introduce a proprietary ISE.

There are many such extensions available providing different functionalities. For example, a security extension for the AES cipher, proposed by Marshall et al. [21], who explore the landscape of AES ISEs and present some requirements to choose amongst them. After creating five different variants based on their findings, they show their implementations for the RISC-V cores 'Rocket' [22], which is a 32-bit or 64-bit RISC-V core that supports the RV32IMC or RV64IMC standard instructions, and 'SCARV' [23], a core that supports only the RV32IMC instructions. In conclusion, the authors state that their third variant, a variant based on hardware-assisted T-tables, is the best variant for a 32-bit RISC-V architecture, while variant four, a variant based on a 64-bit datapath, is the best one for implementing an AES on a 64-bit RISC-V architecture. While increasing the silicon area utilization by 13% to 3%, their ISE variants achieve performance gains ranging from 20% up to 87%. As another example, Marshall et al. [5] propose an ISE for RISC-V to support the ChaCha 256-bit stream cipher [24]. The authors use the RV64IMC core provided by 'Rocket' to implement their ISE and synthesize it on the Kintex-7 XC7K160T FPGA. Their ISE achieves performance gains of up to 53% while introducing a silicon area utilization increase of around 3%. There are also more general extensions. For instance, Alkim et al. in [25] propose an ISE for finite field arithmetic. The authors implement their ISE on the 'VexRiscv' implementation of RISC-V's RV32IMAF[D]C instruction sets. They evaluate their extension using the key-encapsulation mechanism 'Kyber' [26], which is 'based on hardness assumptions over module lattices', and the key exchanging protocol 'NewHope' [27], which is based on the ring learning with errors problem. Last but not least, the bit manipulation extension 'Bitmanip' is developed by the 'RISC-V Foundations Bitmanip Extension Working Group' [28,29] for efficient bit manipulation operations in RISC-V. In their documentation, the authors describe an ISE for manipulating bits like 'rotate', 'count leading/trailing zeros', etc., and also provide example applications, such as bit-field extraction or parity checks.

As can be seen from the presented existing works, most of the research has so far focused on algorithm-specific ISEs rather than more generic approaches. We therefore in our work, similar to the NLU approach, would like to propose an ISE that is able to support many symmetric cryptography algorithms in order to boost their performances on the RISC-V platform.

3. Architecture

There are several different RISC-V 'base' hardware implementations publicly available that can serve as the processor platform for our work. Most of the prevalent implementations are listed at [30]. The implementation deemed best fit for this work was Icicle [31], which implements only the base integer IS RV32I, which is the RVI IS on the 32-bit RISC-V architecture. Furthermore, for our case studies, the most feasible solution to assemble the RISC-V assembly code for the targeted ciphers into machine code for running it in the simulation was to use the RISC-V GNU compiler toolchain [32]. It contains the GNU compiler collection and GNU development tools that provide the means to compile the assembly or C-Code into RISC-V executable machine code. There is also a possibility to extend the assembler with custom mnemonics for one's proprietary ISE. This way, cipher

algorithms can be implemented in assembly in both the base and custom IS, allowing for them to be as similar as possible to be able to compare them fairly.

3.1. The Non-Linear/Linear Instruction Set Extension—NLU

This section gives an overview of the NLU architecture presented in ARITH21 and is hence supposed to give an understanding of the aspects that need to be adjusted for the RISC-V environment. The NLU works as a unit that can perform different operations based on the chosen mode. Depending on the input mode, the NLU is able to calculate logical functions based on the algebraic normal form (ANF) and binary matrix multiply and add in its non-linear and linear modes, respectively. The NLU also contains a 64-bit configuration register that functions as a memory to store the mask for the ANF in the non-linear mode and the matrix for the matrix multiplication in the linear mode. To feed the correct inputs into those singular units and control the data and calculation flow, there are several inputs present in the NLU. A graphical overview of the general architecture of the NLU can be found in the original work [3] as well as in Figure 1.

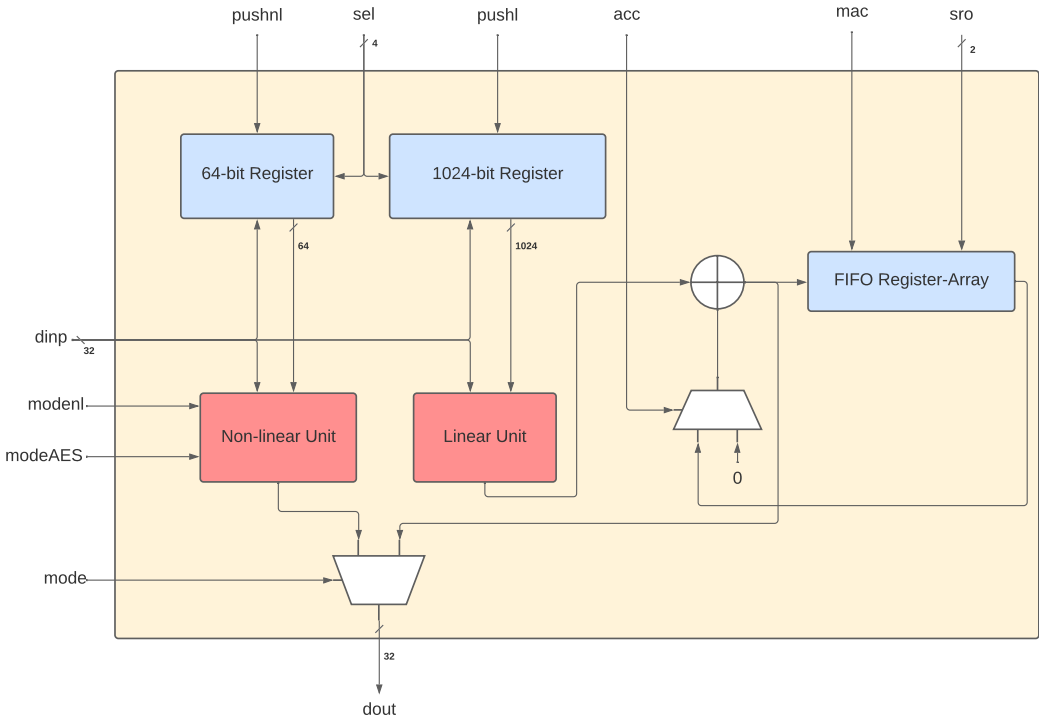

Figure 1. The NLU-V structure (also applicable for the NLU, drawn for 32-bit based on the approach from [3]).

3.1.1. NLU Input/Output

There are seven input signals and one output signal to the NLU unit: 'push', 'sel', 'dinp', 'mac', 'acc', 'mode', 'sro', and 'dout'. To control the loading of data into the configuration register, the calculations, and the outputs, four instructions are defined: 'NLD' for loading the masks into the 64-bit configuration register, 'NNL' for the non-linear operation, and 'NMU' and 'NMA' for the multiply and multiply-and-add operations of the linear unit.

To load data into the 64-bit configuration register, the 'push' signal (1 bit) is set to high by the 'NLD' instruction while the user can determine how many bits of the 'dinp' signal (8 bits) should be pushed by setting the respective number in the 'sel' signal (3 bits). The selected most significant bits of the data input signal is then pushed on the respective least significant bits of the configuration register while the rest of the 64 bits are shifted left. This way, the whole 64-bit configuration register can be loaded in up to eight clock cycles by calling the 'NLD' instruction eight times. This instruction takes two inputs, the data input and the selection input, which correspond to the 'dinp' and 'sel' signals.

If the 'push' signal is not set, the output of the NLU depends on the 'mode' signal (1 bit). This signal determines the output to be either the non-linear or the linear unit's output.

The non-linear unit only requires the configuration register and the data input signal as inputs to calculate its result, which is why the 'NNL' instruction takes only the destination register as the output register and a source register as the data input register as arguments.

The linear unit requires three more signals. The 'mac' signal (1 bit) determines if the result of the linear unit gets pushed on to a four 8-bit register array, which is always the case when using the 'NMU' or 'NMA' instruction. This first-in–first-out (FIFO)-style register array is used to calculate the addition part of the multiply-and-add operation of the linear unit. By pushing the result on to those registers, one can later choose to add one of the four registers to the next result of the multiply operation by setting the 'acc' signal (1 bit) to high and choosing one of the registers with the 'sro' signal (2 bits). Thus, the 'NMU' instruction only takes two inputs, the destination and the source register, as it only has fixed options on the signals: 'mac' is set to high to accumulate the result for further multiply-and-add operations, while the 'acc' signal is set to low, because the caller of the instruction only wants to perform a multiply. The 'NMA' instruction on the other hand has an additional argument corresponding to the 'sro' signal to control which of the four FIFOs should be added to the linear unit result. This way, the NLU's multiply and multiply-and-add operations are designed to store a maximum of four results for the latter operation.

3.1.2. The Non-Linear/Linear Units

The non-linear unit of the NLU is using the ANF approach to calculate the substitution layer (sBox) operations in ciphers. By splitting the 8-bit input into two 4-bit vectors, two 4-bit sBoxes can be calculated at the same time through a function depending on the 64-bit configuration register and a 4-bit input vector. The full circuitry of the ANF unit given in [3] is presented in Figure 2. In general, each of the four output bits can be defined as any logical combination of the four input bits. Block cipher algorithms can use this to implement their own substitution layer (using sBoxes) by configuring the unit via 'NLD' accordingly. The non-linear unit of the NLU simply applies the same configuration values to the two 4-bit vectors resulting from splitting the 8-bit input. By defining this specific ANF circuitry, the NLU is designed to be able to calculate any number of 4-bit sBoxes by simply increasing or decreasing the amount of sBoxes. However, there are additional problems when trying to increase the inputs of the sBox from four to eight bits. This is a topic that needs to be tackled in our adaptation.

The linear unit defines a basic matrix multiplication implemented in the hardware between the input vector and a matrix that is stored in the configuration register. The eight most significant bits of the configuration register are seen as the first row of the matrix, the second eight bits as the second row, and so on. Then, the input data vector is simply a bit-wise matrix multiplied with the 64-bit matrix formed by the configuration register. In combination with the four FIFO registers, data of up to four multiplications (and additions) can be stored and accessed for the multiply-and-add instructions. Especially when defining variations in the identity matrix, one can easily shift around the single bits and put them in any new order, which corresponds to the permutation layer many block ciphers nowadays use, such as in our case studies.

3.2. Adapting the NLU for RISC-V: The NLU-V

As it can be seen in the overall structure of the NLU-V in Figure 1, some changes (such as the datapath width) on the main design of the NLU are necessary. The configuration register size, the datapaths of the non-linear and the linear modules, as well as the input size need to be adjusted.

3.2.1. The Non-Linear Unit

We follow a clear and uncomplicated approach for adapting the 8-bit version of the non-linear unit to 32 bits. Because the intention is to keep the ability of calculating a 4-bit

input sBox, the only change here is to add more sBoxes and split the 32-bit input accordingly. Therefore, instead of using two ANF-based sBoxes like in the NLU, the NLU-V can use eight sBoxes. Similarly, because the circuitry of the ANF stays the same and the input and output of each sBox are still 4 bits wide, no other changes are required with respect to the mask. Therefore, the configuration register can be kept at 64 bits and can be applied to all sBoxes equally. This is the easiest option, which was suggested also in the original work for larger bit widths.

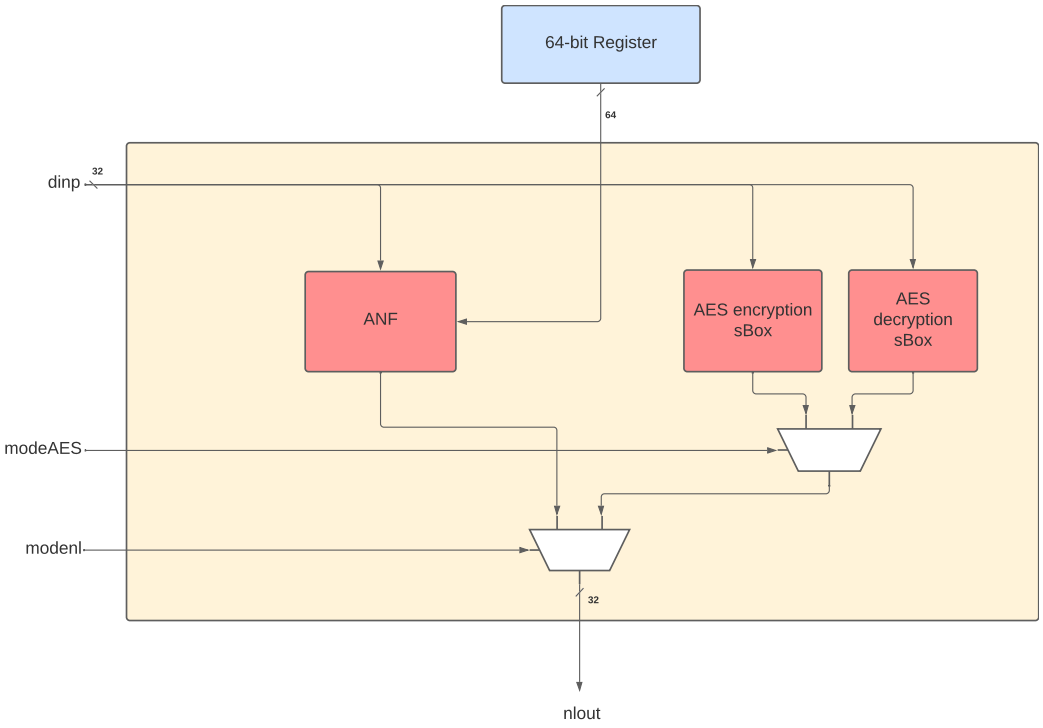

Figure 2. The non-linear unit (also applicable for the NLU, drawn for 32-bit based on the approach from [3]).

However, in the main NLU study, the authors have not considered the option to add an 8-bit sBox support, for example, for the AES [2]. Extending the ANF approach to solve this problem bloats up the software implementation vastly and hence renders the whole unit inefficient in terms of NLU-V's silicon area utilization. Therefore, a new approach in the NLU-V for larger sBox sizes is necessary. It is common to use finite field arithmetic to calculate AES-like sBoxes. There are several studies in the literature implementing finite field arithmetic in hardware for different processor platforms ([33–37]) and even for a RISC-V based implementation [25]. However, none of the previous studies offer a holistic approach to efficiently implement the operations needed for the targeted ciphers. Inspecting the previous works, one major problem seems to be providing a hardware design that can implement the polynomial division or calculate the inverse of an 8-bit vector in the finite field $GF(2^8)$ in an efficient manner. None of the studies investigated so far provide an efficient enough approach to solve this problem: either there are many cycles needed per instruction or the silicon area increases significantly.

Despite these hurdles, we provide an alternative solution: because the AES is widely used (not only in AES but also in some other ciphers), the non-linear unit of the NLU-V will contain a specific sub-unit calculating the sBox values of the AES. According to Canright [38,39], calculating inverses of the polynomials in $GF(2^8)$ can also be performed in $GF((2^2)^2)$ through a composite field approach, which is used also in previous works [40]. The composite field calculation of the AES sBox encryption and decryption presented in Canright's work is publicly available as a hardware implementation in Verilog-HDL [39], which we also employ in our work.

Adding the AES sub-unit requires a couple of additional input signals in the NLU-V architecture as described in Section 3.2.4. An overview of the non-linear unit can be seen in Figure 2. The ANF sub-module contains the same ANF circuit as in the original NLU work [3], but the number of single sBoxes increased to eight instead of two, while the AES encryption and decryption sub-modules use the composite field AES 'Canright' sBox. The 64-bit configuration register, as an input from the top-level NLU-V module, is only fed into the ANF sub-module.

3.2.2. The Linear Unit

The linear unit of the NLU-V is also very similar to its counterpart in the original NLU. Because the input is now a 32-bit vector instead of an 8-bit vector, the multiplication requires a 32×32-bit matrix. Hence, for the new linear unit, instead of a 64-bit configuration register, a 1024-bit register is needed to store the matrix. Other approaches, such as using multiple configuration registers, executing the multiplication in multiple cycles, or reducing the configuration register size and adding more circuitry, which is all controlled by additional instructions and signals, resulted in not being beneficial or feasible. While requiring significantly more area in terms of the register, the approach of increasing the configuration register size to 1024 bits saves additional circuitry and lines of code when implementing ciphers, and hence execution time, by skipping the complex management of loading multiple registers, saving intermediary results, or handling multiple cycles. Therefore, using a 1024-bit configuration register for the linear unit, which is the simpler and probably more efficient approach, is chosen. The new design of the linear unit is presented in Figure 3. The FIFO registers are not affected by any changes and therefore are kept as they are, as can be seen in Figure 4, where 'xlo' and 'rl' correspond to the input and output, respectively (see Figure 1).

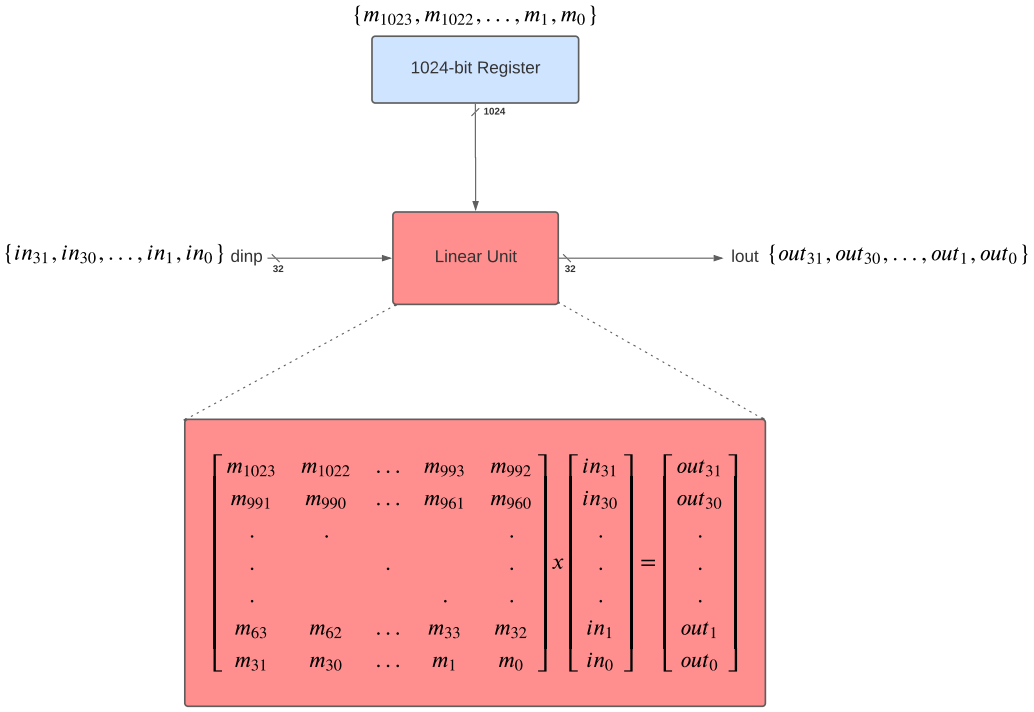

Figure 3. The linear unit.

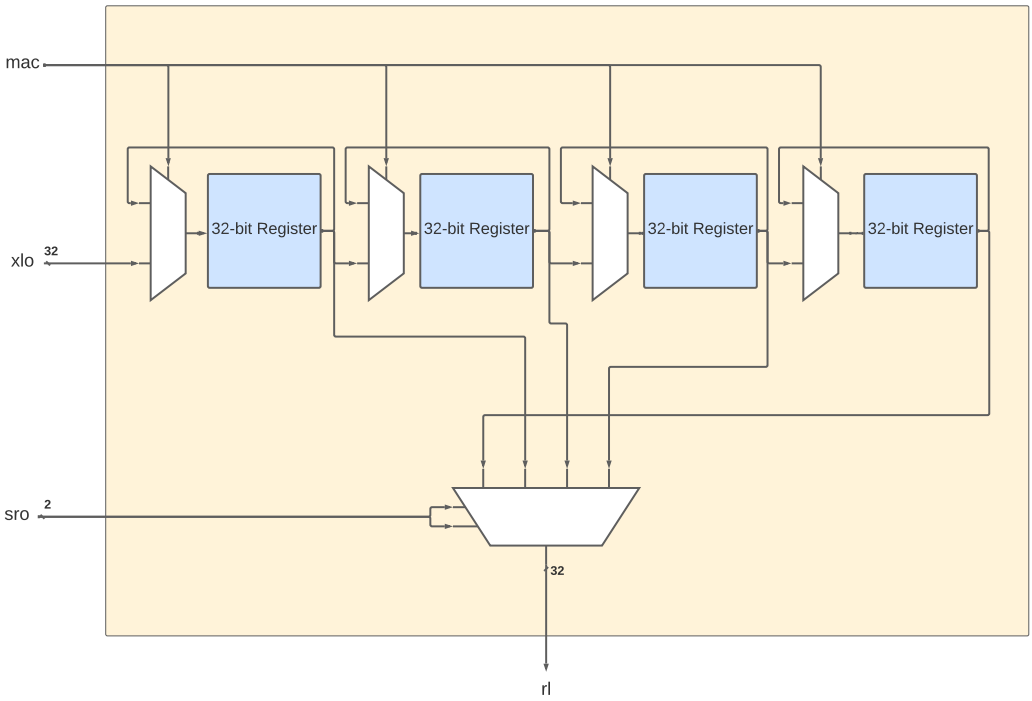

Figure 4. The first-in–first-out (FIFO) register array.

3.2.3. The Configuration Register

Because the NLU-V non-linear unit simply uses several more sBox units without any changes to the use of the configuration register, the NLU-V still needs merely a 64-bit configuration register for the non-linear unit. However, the linear module has to somehow adapt to the fact that a matrix multiplication of a 32-bit input vector will require a 32×32-bit matrix. This causes an additional efficiency problem: in the 8-bit NLU, the shared configuration register is used by the non-linear and linear units simultaneously by loading them every time before the respective 'NNL', 'NMU', and 'NMA' instructions are called. In a 32-bit environment, however, this would take too many cycles with a 1024-bit configuration register and makes the whole process too inefficient. The original NLU is designed with a lightweight approach in mind. Although the silicon area is not unlimited in any case, because the new NLU-V design is not targeting 'Application-specific Integrated Circuits (ASICs)' platforms (at least initially, as an integration with the Icicle RISC-V on FPGA is targeted), there is room for some extra area utilization on the target FPGA. Therefore, for simplifying the loading process of the configuration register, it is split into two registers: one 64-bit register for the non-linear unit and one 1024-bit register for the linear unit. This way, the masks for the non-linear and linear operations can be loaded independent from each other, making the sequential calls of these operations more efficient with some losses in terms of silicon area. This requires new input signals as described in Section 3.2.4. Other alternatives, either having one 1024-bit configuration register for both units or leaving the configuration register at 64 bits, are not implemented. Our investigations show that they are less efficient because they would require significantly more load instructions in assembly.

3.2.4. Input Signals

The changes to the NLU require some new input signals to control the execution and results of the sub-units. Therefore, in addition to the mode signal, a 'modenl' signal is introduced, which determines if the non-linear module uses the ANF or the AES sub-unit. The 'modeAES' signal is further introduced to determine the encryption or decryption mode of the AES sBox module in particular.

The two configuration registers need their own push signals. Therefore, the 'pushnl' and 'pushl' signals are introduced, replacing the 'push' signal, to push the non-linear or linear configuration register, respectively.

The bit size of the following signals are adjusted to fit to the 32-bit design: the 'sel' signal changed from a 3-bit to a 4-bit signal, the 'dinp' signal changed from an 8-bit to a 32-bit signal, and the 'dout' signal changed from an 8-bit to a 32-bit signal.

3.2.5. The NLU-V Instructions

The NLU-V is designed to provide instructions for non-linear and linear operations. The 'NNLANF' instruction can be used to perform the non-linear ANF operation, while the 'NNLAESE' and 'NNLAESD' instructions are for AES encryption and decryption, respectively. To be able to perform the ANF operation, the non-linear unit needs the ANF mask stored in the 64-bit configuration register. This register can be filled by using the 'NLDNL' instruction and providing a 16-bit immediate value. So, the configuration register can be pushed in four instruction calls.

The 'NMU' instruction can be used to execute the linear operation of matrix multiplication and push the result into the FIFO registers. Following that operation, one can use the 'NMA#' (# is the index of the instruction) operation to perform another multiplication, together with an addition of a FIFO register, determined by the number in the name of the four multiply-and-add instructions: 'NMA0' to add register three, 'NMA1' to add register zero, 'NMA2' to add register one, and 'NMA3' to add register two. For the linear unit to function, the 1024-bit configuration register has to be pushed to define the matrix for the matrix multiplication. This can be achieved by using the 'NLDL' instruction. Because it also pushes a 16-bit immediate value, to completely push a new 1024-bit configuration register it takes 64 instructions.

All instructions take two inputs. The first parameter of the two load instructions defines how many bits of the immediate value are pushed into the configuration registers while that value is defined by the second input parameter. The other instructions take the destination and source registers as the first and second inputs. The first input defines where the result should be stored, while the second input defines which value is provided as an input for the NLU-V's non-linear and linear units. The resulting NLU-V IS can be seen in Table 1.

Table 1. NLU-V instructions.

Instruction	Syntax	Description
NLDNL Load non-linear configuration	NLDNL d, u	$CONF_L = CONF_L << u[MSB - d], if\ d > 0$ $CONF_L = CONF_L << u, else$
NLDL Load linear configuration	NLDL d, u	$CONF_{NL} = CONF_{NL} << u[MSB - d], if\ d > 0$ $CONF_{NL} = CONF_{NL} << u, else$
NNLANF NLU non-linear ANF operation	NNLANF d, s	$d(31:28) = ANF[s(31:28)]$ $d(27:24) = ANF[s(27:24)]$ $d(23:20) = ANF[s(23:20)]$ $d(19:16) = ANF[s(19:16)]$ $d(15:12) = ANF[s(15:12)]$ $d(11:8) = ANF[s(11:8)]$ $d(7:4) = ANF[s(7:4)]$ $d(3:0) = ANF[s(3:0)]$

Table 1. *Cont.*

Instruction	Syntax	Description
NNLAESE NLU non-linear AES-sBox encryption	NNLAESE d, s	$d(31:24) = AESEnc[s(31:24)]$ $d(23:16) = AESEnc[s(23:16)]$ $d(15:8) = AESEnc[s(15:8)]$ $d(7:0) = AESEnc[s(7:0)]$
NNLAESD NLU non-linear AES-sBox decryption	NNLAESD d, s	$d(31:24) = AESDec[s(31:24)]$ $d(23:16) = AESDec[s(23:16)]$ $d(15:8) = AESDec[s(15:8)]$ $d(7:0) = AESDec[s(7:0)]$
NMU NLU multiply	NMU d, s	$d = M \times s$ $FIFO = FIFO << M \times s$
NMA0 NLU multiply-and-add FIFO zero	NMA0 d, s	$d = M \times s + FIFO(0)$ $FIFO = FIFO << [M \times s + FIFO(0)]$
NMA1 NLU multiply-and-add FIFO one	NMA1 d, s	$d = M \times s + FIFO(1)$ $FIFO = FIFO << [M \times s + FIFO(1)]$
NMA2 NLU multiply-and-add FIFO two	NMA2 d, s	$d = M \times s + FIFO(2)$ $FIFO = FIFO << [M \times s + FIFO(2)]$
NMA3 NLU multiply-and-add FIFO three	NMA3 d, s	$d = M \times s + FIFO(3)$ $FIFO = FIFO << [M \times s + FIFO(3)]$

4. Results and Discussion

In this section, the post-placement results for the proposed NLU-V design on FPGA as well as a discussion on the performance improvement in the ciphers under study are introduced. The FPGA area utilization for the Icicle, NLU-V, and combined Icicle–NLU-V implementations is presented by showing how much of the total FPGA resources are used in each design. Additionally, the results of the case studies, the PRESENT and AES cipher algorithms, are depicted in this section. To give an overview of the performance, different metrics are chosen for comparison reasons. The lines of code (LoCs) metric shows the difference in the assembly code lines needed in both versions of the algorithms. Furthermore, the TAP reflects the product of the time that each implementation needs to calculate the expected result, given in clock cycles, and the flash memory utilization in terms of how much random access memory (RAM) the program memory and the stack of each implementation requires, given in words. The Icicle implementation has a built-in clock cycle counter which is used for the execution time comparison. Other works in the literature testing ISEs on FPGAs (see [5,41]) are synthesized on a Kintex-7 XC7K160T FPGA (xc7k160tfbg484-3). For a fair comparison with these studies, our implementations are also synthesized on the same FPGA (xc7k160tfbg484-3) using Xilinx Vivado 2021.2. Our implementation is publicly available on GitHub [42].

Table 2 shows the Icicle implementation, which only implements the RV32I standard instruction set and therefore is very lightweight. Most of the silicon area for the NLU-V results from the linear unit and the need of a 1024-bit configuration register for the matrix multiplication in the linear unit. As an interesting comparison, Table 2 also shows the NLU-V area consumption without the linear unit, which is significantly lower due to the missing 1024-bit configuration register. Therefore, reducing the size of the configuration register or the wiring needed for the multiplication in the unit itself, the linear unit can be part of further optimizations.

We also provide the post-placement layout view of our implementations, taken from the Vivado tool. As can be seen in Figure 5, Figure 5a is the Icicle-only implementation, while Figure 5b,c present the NLU-V utilization, with and without the linear unit, respectively. Figure 6 reflects the linear unit impact in more detail: Figure 6a is the NLU-V core with the linear unit implementing the 1024-bit register and Figure 6b is excluding the linear unit, which makes it much smaller.

Table 2. FPGA area utilization, LU: linear unit (* slice register as flip flop, ** multiplexers). *Module hierarchy*: icicle32 contains rv32, rv32 contains nlunit32.

		FLOP_LATCH *	LUT	MUXFX **	DMEM
NLU-V-only (w/o LU)	nlunit32 (w/o LU)	64	942	-	64
NLU-V-only	nlunit32	1088	4520	-	32
Icicle-only	icicle32	974	2128	3230	4184
	rv32	778	1735	30	88
Icicle + NLU-V (w/o LU)	icicle32	1051	3140	3229	4248
	rv32	855	2746	29	152
	nlunit32	64	252	-	64
Icicle + NLU-V	icicle32	2115	6762	3235	4216
	rv32	1919	6365	35	120
	nlunit32 (w/o LU)	1088	3406	-	32

In previous works [5,41], the proposed ISEs increase the resource utilization by approximately 1–3% in comparison to the base processor cores used. Marshall et al. show multiple variants of their ISE, which increases the area consumption around 3–13% [21]. This seems significantly lower than the area increase that the NLU-V causes. But it should be noted that the other ISEs implement only one specific algorithm, while the NLU-V aims to address many different cryptographic block ciphers. This main difference in the implementation approach makes a direct comparison of the NLU-V to previous works impractical.

To conduct a precise analysis of the impact of the NLU-V on another RISC-V system similar to the RV64G Rocket Core used by Marshall et al. [5], implementing and synthesizing the NLU-V on that system is necessary, which is not in the scope of this paper. We however estimated the costs roughly: An increase of only 30–40% can be expected if the NLU-V is added to the Rocket Core; however, note that the NLU-V would still need to be adjusted for a 64-bit environment, which would mean additional cost.

In order to introduce our novel NLU-V instructions to the compiler, we have also modified the RISC-V GNU compiler toolchain. We have appended the NLU-V instructions presented in Table 1 to the compiler, which can also be seen in our repository [42].

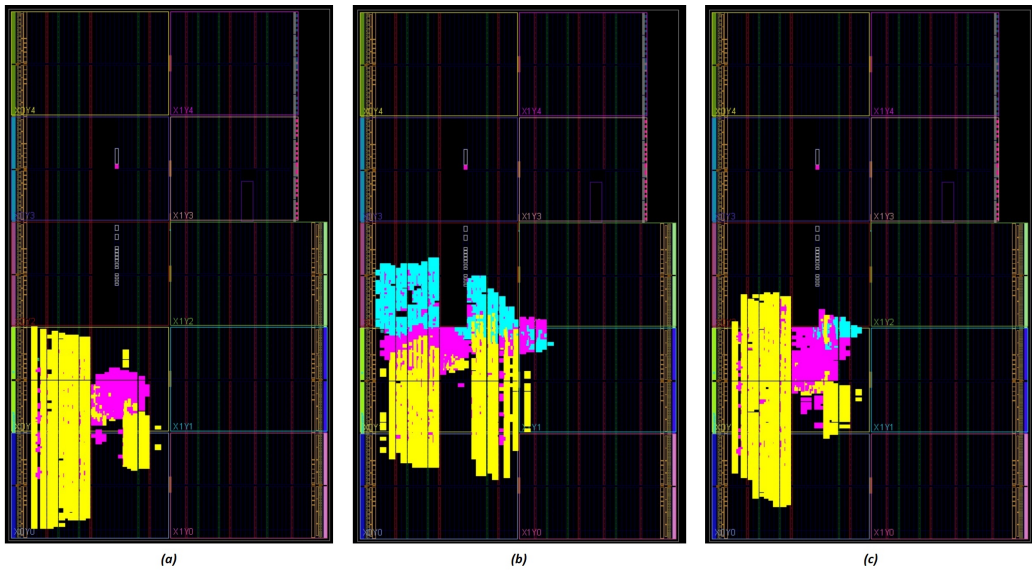

Figure 5. FPGA layout: (**a**) Icicle-only, (**b**) Icicle + NLU-V, and (**c**) Icicle + NLU-V w/out linear unit (*yellow*: icicle32, *magenta*: rv32, and *cyan*: nlunit32).

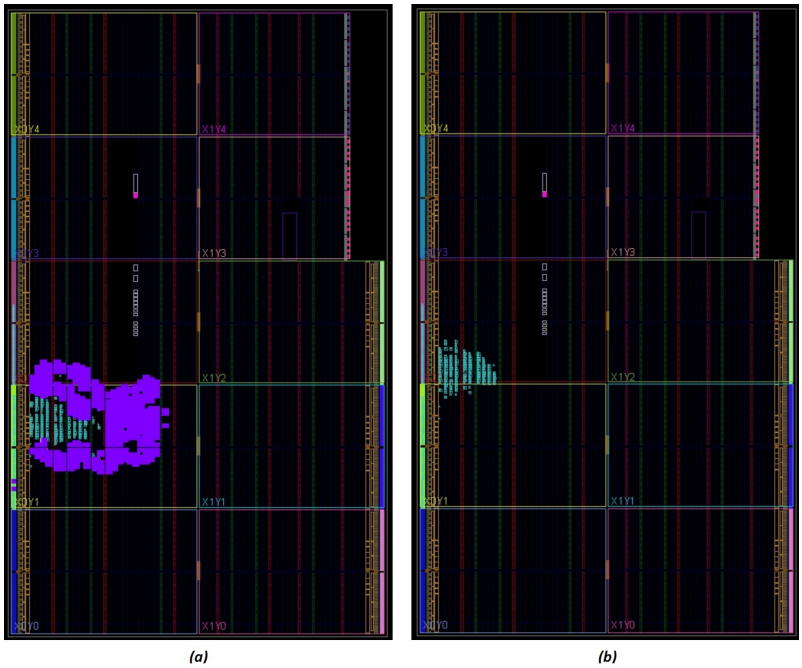

Figure 6. FPGA layout NLU-V (**a**) with linear unit, and (**b**) without linear unit (*purple*: linear unit).

The metrics reflecting the assembly implementation and simulation results for the cipher algorithms are depicted in Table 3. This table shows how many LoCs each algorithm requires as well as the gain (in percentage) that the NLU-V versions have in comparison to their RV32I counterpart. Additionally, the table depicts the number of clock cycles as a metric of time and the flash memory consumption as a metric of the memory utilization (code size) that each algorithm needs for its execution. Multiplying those metrics, Table 4 shows the TAP, which in its turn allows for showing the TAP gain of the NLU-V in comparison to the RV32I versions of the algorithms.

Table 3. Performance comparison.

	LoCs	LoC Gain in %	Time (Clock Cycles)	Flash Memory (Words)
PRESENT RV32I	349	0	42,862	593
PRESENT NLU-V	197	44	13,898	201
AES RV32I	298	0	25,824	542
AES NLU-V	228	23	19,056	234

Table 4. Time-area-product (TAP) gain.

	Time-Area-Product (TAP)	TAP Gain in %
PRESENT RV32I	25.4×10^6	0
PRESENT NLU-V	2.8×10^6	89
AES RV32I	14×10^6	0
AES NLU-V	4.5×10^6	68

5. Conclusions and Future Work

In this study, the 8-bit NLU ISE architecture presented by Engels et al. [3] is re-designed and extended to a 32-bit NLU-V architecture with the RISC-V architecture in

mind. The RISC-V implementation Icicle that implements the RV32I standard instruction set of the RISC-V architecture is selected as the target implementation platform. Hence, the RV32I standard instruction set is extended with the NLU-V instructions, which provide accelerating instructions for non-linear and linear operations of block cipher algorithms. Two cipher algorithms, the PRESENT and AES ciphers, are implemented in both the RISC-V-only and the NLU-V versions. For the RISC-V-only versions, only the RV32I standard instruction set is used, which is implemented by using Icicle. For the NLU-V versions, the NLU-V instructions are used in addition to the standard instructions. For both ciphers, the LoCs and TAP metrics show that one can achieve significant gain in terms of execution time, flash memory usage, and LoCs.

Our future steps include improving the proposed hardware units to decrease the hardware utilization and extend the case studies with further ciphers in order to include novel standard algorithms. As a result of implementing such additional ciphers in the NLU-V ISE, we may come across potential hardware modifications and optimizations to address these ciphers better with some new NLU-V instructions, hence covering a larger set of block ciphers.

In this work, a detailed analysis of the security implications of the NLU-V extension and its potential vulnerabilities are beyond our scope. In order to protect the ISE against side-channel-based physical attacks, masked architectures can be considered; however, we omit this for this paper due to our lightweight focus. In future studies, we plan to provide an evaluation of masked implementations of the NLU-V and report the corresponding costs. Moreover, we plan to look into fault attacks on the NLU-V.

Author Contributions: Conceptualization and methodology, H.U. and E.B.K.; software and hardware code, H.U.; validation and formal analysis, H.U. and E.B.K.; investigation, H.U.; resources, E.B.K.; data curation and base work, H.U. and E.B.K.; writing—original draft preparation, H.U. and E.B.K.; writing—review and editing, E.B.K.; visualization, H.U. and E.B.K.; supervision, E.B.K.; project administration and funding acquisition, E.B.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by AMD under the Xilinx University Program.

Data Availability Statement: The source code of our work is accessible in the following link: https://github.com/UzunerH/MasterThesisCode (accessed on 15 February 2022).

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.B.; Seurin, Y.; Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2007, Vienna, Austria, 10–13 September 2007*; Paillier, P., Verbauwhede, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466. [CrossRef]
2. FIPS 197. Advanced Encryption Standard (AES). 2001. Available online: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (accessed on 15 February 2022).
3. Engels, S.; Kavun, E.B.; Paar, C.; Yalçın, T.; Mihajloska, H. A Non-Linear/Linear Instruction Set Extension for Lightweight Ciphers. In Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic, Austin, TX, USA, 7–10 April 2013; pp. 67–75. [CrossRef]
4. Atmel. ATmega8 Datasheet. 2013. Available online: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2486-8-bit-AVR-microcontroller-ATmega8_L_datasheet.pdf (accessed on 15 February 2022).
5. Marshall, B.; Page, D.; Pham, T.H. A lightweight ISE for ChaCha on RISC-V. Cryptology ePrint Archive, Paper 2021/1030. 2021. Available online: https://eprint.iacr.org/2021/1030 (accessed on 15 February 2022).
6. Marshall, B.; Page, D.; Hung Pham, T. A Lightweight ISE for ChaCha on RISC-V. In Proceedings of the 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP), Virtual Conference, 7–9 July 2021; pp. 25–32. [CrossRef]
7. Marshall, B.; Page, D.; Pham, T. Implementing the Draft RISC-V Scalar Cryptography Extensions. In Proceedings of the Hardware and Architectural Support for Security and Privacy (HASP '20), Virtual, Greece, 17 October 2020. [CrossRef]

8. Shirai, T.; Shibutani, K.; Akishita, T.; Moriai, S.; Iwata, T. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In *Fast Software Encryption*; Biryukov, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 181–195. [CrossRef]
9. Daemen, J.; Knudsen, L.; Rijmen, V. The Block Cipher Square. In *Fast Software Encryption*; Biham, E., Ed.; Springer: Berlin/Heidelberg, Germany, 1997; pp. 149–165. [CrossRef]
10. Hong, D.; Sung, J.; Hong, S.; Lim, J.; Lee, S.; Koo, B.S.; Lee, C.; Chang, D.; Lee, J.; Jeong, K.; et al. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2006, Yokohama, Japan, 10–13 October 2006*; Goubin, L., Matsui, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 46–59. [CrossRef]
11. De Cannière, C.; Dunkelman, O.; Knežević, M. KATAN and KTANTAN—A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2009, Lausanne, Switzerland, 6–9 September 2009*; Clavier, C., Gaj, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 272–288. [CrossRef]
12. Gong, Z.; Nikova, S.; Law, Y.W. KLEIN: A New Family of Lightweight Block Ciphers. In *RFID. Security and Privacy*; Juels, A., Paar, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 1–18. [CrossRef]
13. Rivest, R.L.; Robshaw, M.J.; Sidney, R.; Yin, Y.L. The RC6TM Block Cipher. In Proceedings of the First Advanced Encryption Standard (AES) Conference, Ventura, CA, USA, 20–22 August 1998; p. 16. Available online: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=61b9b24c25c2e1e4cf4acbf93a7578121429d758 (accessed on 15 February 2022).
14. Lim, C.H.; Korkishko, T. mCrypton—A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors. In *Information Security Applications*; Song, J.S., Kwon, T., Yung, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 243–258. [CrossRef]
15. Guo, J.; Peyrin, T.; Poschmann, A.; Robshaw, M. The LED Block Cipher. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2011, Nara, Japan, 28 September–1 October 2011*; Preneel, B., Takagi, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 326–341. [CrossRef]
16. Shibutani, K.; Isobe, T.; Hiwatari, H.; Mitsuda, A.; Akishita, T.; Shirai, T. Piccolo: An Ultra-Lightweight Blockcipher. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2011, Nara, Japan, 28 September–1 October 2011*; Preneel, B., Takagi, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 342–357. [CrossRef]
17. Anderson, R.; Biham, E.; Knudsen, L. Serpent: A Proposal for the Advanced Encryption Standard. *NIST AES Propos.* **1998**, *174*, 1–23. Available online: https://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf (accessed on 15 February 2022).
18. RISC-V. RISC-V Specifications. 2021. Available online: https://riscv.org/technical/specifications/ (accessed on 15 February 2022).
19. RISC-V. RISC-V Specifications Volume 1, Unprivileged Spec v. 20191213. 2019. Available online: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (accessed on 15 February 2022).
20. RISC-V. RISC-V Specifications Volume 2, Privileged Spec v. 20211203. 2021. Available online: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf (accessed on 15 February 2022).
21. Marshall, B.; Newell, G.R.; Page, D.; Saarinen, M.J.O.; Wolf, C. The Design of Scalar AES Instruction Set Extensions for RISC-V. Cryptology ePrint Archive, Report 2020/930. 2020. Available online: https://ia.cr/2020/930 (accessed on 15 February 2022).
22. Asanović, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. *The Rocket Chip Generator*; Technical Report UCB/EECS-2016-17; EECS Department, University of California: Berkeley, CA, USA, 2016. Available online: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html (accessed on 15 February 2022).
23. Marshall, B. SCARV: A Side-Channel Hardened RISC-V Platform. 2021. Available online: https://github.com/scarv/scarv (accessed on 30 March 2022).
24. Bernstein, D.J. ChaCha, A Variant of Salsa20. In Proceedings of the Workshop Record of SASC, Lausanne, Switzerland, 13–14 February 2008; pp. 3–5. Available online: https://cr.yp.to/chacha/chacha-20080120.pdf (accessed on 30 March 2022).
25. Alkim, E.; Evkan, H.; Lahr, N.; Niederhagen, R.; Petri, R. ISA Extensions for Finite Field Arithmetic—Accelerating Kyber and NewHope on RISC-V. Cryptology ePrint Archive, Report 2020/049. 2020. Available online: https://ia.cr/2020/049 (accessed on 15 February 2022).
26. Bos, J.; Ducas, L.; Kiltz, E.; Lepoint, T.; Lyubashevsky, V.; Schanck, J.M.; Schwabe, P.; Seiler, G.; Stehlé, D. CRYSTALS—Kyber: A CCA-Secure Module-Lattice-Based KEM. Cryptology ePrint Archive, Report 2017/634. 2017. Available online: https://ia.cr/2017/634 (accessed on 15 February 2022).
27. Alkim, E.; Avanzi, R.; Bos, J.; Ducas, L.; de la Piedra, A.; Pöppelmann, T.; Schwabe, P.; Stebila, D. NewHope. 2020. Available online: https://newhopecrypto.org/index.shtml (accessed on 30 March 2022).
28. Claire Wolf Symbiotic GmbH. RISC-V Bitmanip Extension Document Version 0.94-Draft. 2021. Available online: https://raw.githubusercontent.com/riscv/riscv-bitmanip/master/bitmanip-draft.pdf (accessed on 29 March 2022).
29. RISC-V Foundations Bitmanip Extension Working Group. RISC-V Bitmanip (Bit Manipulation) Extension. 2021. Available online: https://github.com/riscv/riscv-bitmanip/tree/main-history (accessed on 30 March 2022).
30. RISC-V. RISC-V Exchange: Cores & SoCs. 2021. Available online: https://riscv.org/exchange/cores-socs/ (accessed on 21 February 2022).
31. Edgecombe, G. Icicle—32-bit RISC-V Implementation. 2019. Available online: https://github.com/grahamedgecombe/icicle (accessed on 15 February 2022).
32. The Regents of the University of California. RISC-V GNU Compiler Tool Chain. 2016. Available online: https://github.com/riscv-collab/riscv-gnu-toolchain (accessed on 15 February 2022).

33. Fiaz, F.; Masud, S. Design and Implementation of A Hardware Divider in Finite Field. In Proceedings of the National Conference on Emerging Technologies, Karachi, Pakistan, 18 December 2004. Available online: https://www.researchgate.net/publication/237228696_Design_and_Implementation_of_a_Hardware_Divider_in_Finite_Field (accessed on 15 February 2022).
34. Ward, R.W.; Molteno, D.T.C.A. Efficient Hardware Calculation of Inverses in GF(2^8). 2015. Available online: https://api.semanticscholar.org/CorpusID:27223451 (accessed on 15 February 2022).
35. Md Naziri, S.Z.; Mei, Y.; Idris, N. The Verilog HDL-based Design of Multiplicative Inverse Value of $GF(2^8)$ Auto-Generator Using Extended Euclid Algorithm Method for Advanced Encryption Standard Algorithm. In *Integrated Electronics: Designs and Systems*; Penerbit Universiti Malaysia Perlis: Kangar, Malaysia, 2013; Volume 1, pp. 77–90. Available online: https://www.researchgate.net/publication/265729275_The_Verilog_HDL-based_Design_of_Multiplicative_Inverse_Value_of_GF28_Auto-generator_using_Extended_Euclid_Algorithm_Method_for_Advanced_Encryption_Standard_Algorithm (accessed on 15 February 2022).
36. Chen, T.C.; Wei, S.W.; Tsai, H.J. Arithmetic Unit for Finite Field $GF(2^m)$. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2008**, *55*, 828–837. [CrossRef]
37. Sarkar, P.; Roy, B.; Choudhury, P.; Barua, R. Polynomial Division Using Left Shift Register. *Comput. Math. Appl.* **1998**, *35*, 27–31. [CrossRef]
38. Canright, D. A Very Compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems, Proceedings of the CHES 2005, Edinburgh, UK, 29 August–1 September 2005*; Rao, J.R., Sunar, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 441–455. [CrossRef]
39. Canright, D. A Very Compact Rijndael S-Box. 2005. Available online: https://core.ac.uk/download/pdf/36694529.pdf (accessed on 15 February 2022).
40. Moradi, A.; Poschmann, A.; Ling, S.; Paar, C.; Wang, H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Advances in Cryptology, Proceedings of the EUROCRYPT 2011, Tallinn Estonia, 15–19 May* 2011; Springer: Berlin/Heidelberg, Germany; Volume 6632, pp. 69–88. [CrossRef]
41. Gao, S.; Marshall, B.; Page, D.; Pham, T. FENL: An ISE to Mitigate Analogue Micro-architectural Leakage. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2020**, *2020*, 73–98. [CrossRef]
42. Uzuner, H. NLU-V. 2022. Available online: https://github.com/UzunerH/MasterThesisCode (accessed on 30 March 2022).