future internet

MDPI

Article

Design of an SoC Based on 32-Bit RISC-V Processor with Low-Latency Lightweight Cryptographic Cores in FPGA

Khai-Minh Ma [1], Duc-Hung Le [1,*], Cong-Kha Pham [2] and Trong-Thuc Hoang [2]

[1] Faculty of Electronics and Telecommunications, The University of Science, Vietnam National University Ho Chi Minh City, Ho Chi Minh City 700000, Vietnam
[2] Department of Computer and Network Engineering, The University of Electro-Communications (UEC), Tokyo 182-8585, Japan
* Correspondence: ldhung@hcmus.edu.vn

Abstract: The security of Internet of Things (IoTs) devices in recent years has created interest in developing implementations of lightweight cryptographic algorithms for such systems. Additionally, open-source hardware and field-programable gate arrays (FPGAs) are gaining traction via newly developed tools, frameworks, and HDLs. This enables new methods of creating hardware and systems faster, more simply, and more efficiently. In this paper, the implementation of a system-on-chip (SoC) based on a 32-bit RISC-V processor with lightweight cryptographic accelerator cores in FPGA and an open-source integrating framework is presented. The system consists of a 32-bit VexRiscv processor, written in SpinalHDL, and lightweight cryptographic accelerator cores for the PRINCE block cipher, the PRESENT-80 block cipher, the ChaCha stream cipher, and the SHA3-512 hash function, written in Verilog HDL and optimized for low latency with fewer clock cycles. The primary aim of this work was to develop a customized SoC platform with a register-controlled bus suitable for integrating lightweight cryptographic cores to become compact embedded systems that require encryption functionalities. Additionally, custom firmware was developed to verify the functionality of the SoC with all integrated accelerator cores, and to evaluate the speed of cryptographic processing. The proposed system was successfully implemented in a Xilinx Nexys4 DDR FPGA development board. The resources of the system in the FPGA were low with 11,830 LUTs and 9552 FFs. The proposed system can be applicable to enhancing the security of Internet of Things systems.

Keywords: system-on-chip; FPGA; RISC-V; VexRiscv; lightweight cryptography

Citation: Ma, K.-M.; Le, D.-H.; Pham, C.-K.; Hoang, T.-T. Design of an SoC Based on 32-Bit RISC-V Processor with Low-Latency Lightweight Cryptographic Cores in FPGA. *Future Internet* **2023**, *15*, 186. https://doi.org/10.3390/fi15050186

Academic Editor: Wei Yu

Received: 1 May 2023
Revised: 16 May 2023
Accepted: 18 May 2023
Published: 19 May 2023

1. Introduction

In recent years, RISC-V [1], a free and open-source instruction set architecture (ISA) for microprocessors, has received considerable attention. Based on the reduced instruction set computer (RISC) design principle, RISC-V is compact, scalable, and highly configurable. These distinguishing features make it appealing to open-source communities in the academic and commercial sectors. The development of a standalone processor, however, is insufficient. The computational tasks handled by the processor have become increasingly complex, surpassing the general-purpose computing capabilities that they were able to perform whilst still being required to be highly efficient. Using accelerator cores, which are capable of managing such complex and intensive tasks, it reduces the execution time and saves energy compared to microprocessor-based tasks. A number of studies in RISC-V system development with accelerator cores have been performed, with applications including digital signal processing [2], artificial intelligence [3], and the implementation of mathematical algorithms.

On the other hand, extensive real-world and real-time data transferred between peripheral devices and nodes are potential cyber-attack targets due to the rapid expansion of the Internet of Things (IoTs). The obvious and effective countermeasure is to effectively

encrypt the data. As a consequence, lightweight cryptographic algorithms are approaching the horizon of cryptographic solutions, as a result of numerous advancements over the past few years. These lightweight cryptographic algorithms have a small memory footprint and low computational complexity, allowing them to be implemented in devices with limited resources. In addition to RISC-V, the need for encryption in these peripheral devices has spurred the development of new hardware and platforms with improved energy efficiency, performance, connectivity, and security. In recent years, many new tools, toolchains, frameworks, and HDLs have been developed due to the present state of open-source hardware and the popularity of field-programable gate arrays (FPGAs). This enables the development of new methods for creating systems that are quicker, simpler, and more efficient.

In this paper, we present the implementation and results of a system-on-chip (SoC) based on a 32-bit RISC-V processor with lightweight cryptographic accelerator cores in an FPGA. The proposed system was configured with a 32-bit VexRiscv processor, implementing RV32IM instruction sets, and lightweight cryptographic accelerator cores for the PRINCE block cipher, the PRESENT-80 block cipher, the ChaCha stream cipher, and the SHA3-512 hash function. The selection of these three algorithms was driven by a desire to experiment with recently discovered, promising lightweight cryptographic algorithms. These lightweight cryptographic cores were also developed for low-implementation cycles, resulting in low latency. The focus of this work was to integrate and provide a complete system-on-chip implementation of the best RISC-V processors and cryptographic accelerators, with minimal compromise. This was achieved using recently developed and novel toolchains, frameworks, and HDLs to build the system. Using the Configuration/Status Register (CSR) bus to connect customized and optimized cores such as PRINCE, PRESENT-80, ChaCha, and SHA-3 with VexRiscv to form a high-performance SoC with low-implementation clock cycles and low-logic resources was also a notable contribution of this work. Compared to the software implementation of the cryptography algorithms, the implementation of the system using 11,830 look-up tables (LUTs) and 9552 flip-flops (FFs) reduces the execution time by 100 to over 4400 times. The design was deployed using a Xilinx Nexys4 DDR FPGA board and the Vivado toolchain, and firmware was also developed to effectively utilize all lightweight cryptographic accelerator cores. The objective of this paper was to provide a simple and efficient SoC design using CSR configuration and open-source resources.

The remainder of this paper is organized as follows. Section 2 provides some background of relevant works around the related topics of RISC-V and cryptography. Sections 3 and 4 cover the relevant fundamental subjects, including the VexRiscv core, lightweight cryptographic algorithms, and the hash function. The implementation process is depicted in Section 5. The experiment and validation results, along with discussions and comparisons with other works, are analyzed in Section 6. Finally, Section 7 concludes the paper.

2. Related Works

In this section, we have examined a wide variety of interesting studies and topics that have recently experienced a remarkable expansion and increase in research. The first is the emergence of RISC-V, which has resulted in many open-source projects and academic research [4] focusing in RISC-V-based processor implementations. The growth and expansion of the ISA have been commented on as being "inevitable" [5], as adoption was already happening. Some of the work even progressed to the stage where it was market-ready [6]. Works such as [7] provide freely available courses along with comprehensive instructions and labs for educational purposes, promoting the RISC-V ecosystem. Applications for these RISC-V processors vary from the Internet of Things, such as security monitoring systems [8] with real-time detection and tracking and edge computing platforms based on callability [9], to neural networks, artificial intelligence, and more.

An effective DNN (deep neural network)-application-focused RISC-V processor was proposed by Zhang H. et al. [10]. The work demonstrated promising capabilities in effec-

tively executing DNN tasks while minimizing power consumption. This design approach ensured that the processor was well suited for edge devices, where power efficiency is crucial for extending battery life and enabling real-time processing. Lim S.-H. et al. also proposed work [11] on implementing a DNN operation accelerator based on a virtual platform of RISC-V and successfully processed the darknet CNN model. This integration enabled the efficient execution of complex convolutional operations, resulting in enhanced performance and reduced computational overhead. Another work by Gamino del Río I. proposed modifying the architecture of an RISC pipelined processor to eliminate the execution time overhead introduced by code instrumentation [12]. An RISC-V processor was implemented in VHDL and synthesized in an FPGA, which enabled non-intrusive tracing while executing instrumented code without introducing additional delays. Robotics applications were also researched by Lee J. by implementing the robotics operating system on top of an RISC-V processor in an FPGA [13].

The categories even extended to some special applications, such as in space environments, where D. A. Santos et al. presented a low-cost fault-tolerant implementation of the RISC-V architecture that reduced error propagation and was aimed at space applications [14]. A similar implementation was NOEL-V which was compatible with the AMBA AHB 2.0 bus and could efficiently be deployed in an FPGA and ASIC [15].

The second topic that is related to this work and received the same amount of attention is cryptographic algorithms, especially lightweight cryptographic algorithms in RISC-V. They are algorithms that were designed to be implemented in resource-constrained devices. Some analysis was made by El-hajj, M. et al. to further investigate the adequate cryptographic algorithms for these systems by evaluating and benchmarking more than 39 symmetric block ciphers [16]. The work by Hao Cheng et al. [17] provided the completed fundamental analysis, design, implementation results, hardware, and software of an ISE (instruction set extension) for 10 lightweight cryptography algorithms. In addition to utilizing the C programing language for pure software implementations [18], it seemed that the preferred implementation approach also involved using an ISE. The work [19] introduced a lightweight ISE designed to support the ChaCha stream cipher in RISC-V architectures, which achieved a speedup gain of at least 5.4 times compared to the OpenSSL baseline and 3.4 times compared to an ISA optimized implementation. The study [20] focused on achieving secure and efficient execution of AES by separating different ISEs for 32-bit and 64-bit, which demonstrated significant performance improvements for AES-128 block encryption. Furthermore, the authors explored how the proposed standard bit manipulation extension in RISC-V can be effectively utilized for the efficient implementation of AES-GCM (Galois/Counter Mode). The GIFT family of block ciphers was utilized in various NIST candidates but required optimization techniques such as bit-slicing and fix-slicing for optimal performance. The researchers of [21] developed assembly implementations for GIFT-64 and GIFT-128 using the RV32I ISA, evaluated their performance in the HiFive1 development board, and achieved clock cycle reductions of 88.69% (GIFT-64) and 95.05% (GIFT-128) using fix-slicing with the key pre-computation technique. ASCON, a recently standardized lightweight cryptographic algorithm by NIST, has been implemented by Altınay Ö. in the base RV32I processor. The proposed work [22] implemented non-standard RISC-V instructions, and an end-to-end test environment was formed by extending the GNU Compiler Collection and Spike RISC-V ISA Simulator.

Another work, [23], proposed a new and efficient validation platform by deploying a cryptographic SoC as a virtual prototype using a hybrid hardware and software design strategy. Compared to RTL simulation, the custom virtual prototype demonstrated significant performance advantages, performing approximately 10–450 times faster while maintaining a simulation error of only about 4%.

Among the numerous related research projects and works, we have identified a smaller sub-set that is more connected to and comparable to our work, thereby distinguishing it from those that may have more distant or peripheral relevance. First, the work [24,25] presented standalone implementations of the PRINCE block cipher in an FPGA, aiming

for low resource usage. For PRESENT-80, the work [26] integrated the crypto co-processor for both AES and PRESENT-80 in an FPGA-based SoC platform with an emphasis on energy efficiency and performance. The implementation of ChaCha [27] by Nurat At et al. enhanced efficiency by interleaving independent tasks and minimizing data dependencies. Concerning the SHA-3 hash functions, since NIST's announcement of SHA-3, there has been a surge in research and numerous hardware implementations dedicated to this algorithm, including [28,29]. The current state-of-the-art, well-optimized ASIC implementation [30] in 7 nm TSMC also falls into this category. Evaluations between these works and our results will be discussed in Section 7.

Similarly to our work, two earlier works [31,32] designed and incorporated algorithm acceleration processing into the SoC, but in two distinct ways: once by optimizing hardware instructions and once by optimizing software instructions.

3. RISC-V Processor

3.1. RISC-V ISA

RISC-V is an open-source ISA which has been widely accepted and adopted in many projects by communities. Many resources and tools, such as compilers and debuggers, have also been developed, forming a diverse open-source ecosystem. The RISC-V project started in 2010 at UC Berkeley. Compared to ARM and x86, an RISC-V processor has the following advantages:

- Free: RISC-V ISA and its surrounding development projects are mostly open-source.
- Simple: RISC-V is much smaller than other commercial ISAs.
- Modular: RISC-V has a small standard base ISA with multiple standard extensions.
- Stable: The base and many extensions of the ISA are already standardized and frozen. No significant changes are expected.
- Extensibility: specialized instructions can be added, based on extensions.

The based instruction sets defined by RISC-V are RV32I, RV64I, and RV128I, supporting 32-bit, 64-bit, and 128-bit, respectively, with around 40 instructions. RISC-V specifies the encoding, control flow, register sizes, memory, addressing, logic manipulation, and ancillaries for the processor. While RV64I is suited for large, sophisticated, complex systems, and RV128I can serve as a theoretical instruction set for a 128-bit processor in the future, RV32I is the most suitable for small embedded systems, such as IoT devices, for example. In addition, the architecture of the processor can be extended, advancing its specialization through standardized extended instruction sets such as M, A, F, D, Q, and G (general purpose—IMAFD). In recent years, many implementations of RISC-V processors have been published in response to the open-hardware movement. Some notable ISA implementation processors supported by communities include the RocketChip [33] from UC Berkeley, the BlackParrot [34], and the SiFive E31 [6].

3.2. VexRiscv Processor

VexRiscv is a 32-bit RISC-V processor implementing RV32IMAC instruction sets and is optimized for FPGA. VexRiscv was written in SpinalHDL, a new open-source high-level digital hardware describing language with primitive's library, which is a domain-specific language (DSL) based on the Scala programing language. Since SpinalHDL allows object-oriented programing and functional programing to elaborate the hardware, VexRiscv has a modular design, with almost all of the components being optional plugins. These include caches for instruction and data and debug extension, as well as the number of pipeline stages, interruptions, exception handling, and a memory management unit. These customization abilities make VexRiscv the ideal platform for developing an SoC with hardware accelerators. Using the LiteX framework, the proposed SoC in this study, which consisted of a VexRiscv processor integrated with lightweight cryptographic accelerator components and other fundamental peripherals, was constructed.

4. Cryptographic Algorithms

4.1. PRINCE

PRINCE [35] is a lightweight symmetric block cipher with a substitution–permutation network (SPN) structure and is based on the so-called FX construction. It was designed to target low latency and unrolled hardware implementations. According to the author, when compared with the advanced encryption standard (AES), PRINCE was able to operate at much higher frequencies while utilizing less area with the same timing constraints and technologies.

The block size of PRINCE is 64 bits, and the key size is 128 bits. The key is split into two 64-bit keys denoted as k_0 and k_1,

$$k = k_0 || k_1 \tag{1}$$

A sub-key k'_0 is then derived from k_0, extending the key to 192 bits.

$$(k_0 || k'_0 || k_1) := (k_0 || (k_0 \ggg 1) \oplus (k_0 \ggg 63) || k_1) \tag{2}$$

The input is XORed with k_0, and then processed via a core function, $\text{PRINCE}_{\text{core}}$, using k_1. The output of the $\text{PRINCE}_{\text{core}}$ function is XORed by k'_0 to produce the final output. The decryption is conducted by exchanging k_0 and k'_0 and using k_1 XORed with a constant denoted as alpha in the core function. This overall procedure is depicted in Figure 1.

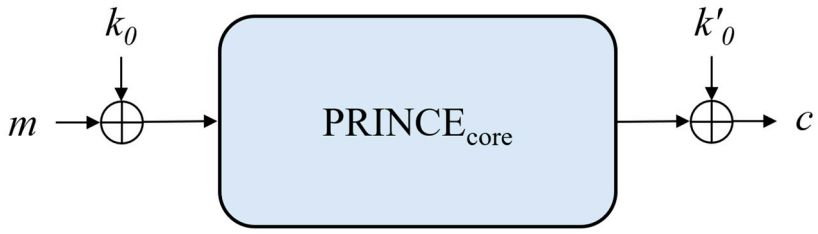

Figure 1. PRINCE encryption process of a plaintext message m to a ciphertext c, using k_0 and k'_0 as whitening keys.

The $\text{PRINCE}_{\text{core}}$ performs the encryption process, depicted in Figure 2, in twelve rounds, divided into five "forward" rounds, five "backward" rounds, and a middle round (that is counted as two). A forward round starts with a non-linear substitution layer S, a linear layer M of permutation, and then the XORed operation with the round constant and k_1. The "backward" rounds are the identical inverse of the "forward" rounds with different round constants.

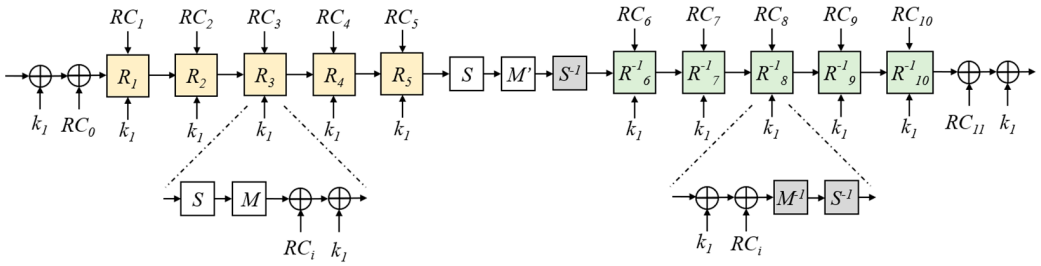

Figure 2. Encryption process of $\text{PRINCE}_{\text{core}}$.

In terms of hardware implementation, the target is to have a pipeline architecture with combinational sub-modules for the S layer and M layer. The algorithm was designed to be unrolled, which means that it should not contain any conditional loops. The key expansion and addition function (XOR) should then be simple to implement.

4.2. PRESENT-80

PRESENT [36] is an ultra-lightweight block cipher, developed as a collaboration between Ruhr-University, Germany, and the Technical University of Denmark back in 2007. The algorithm is also based on the SPN structure, the same as PRINCE, and consists of 31 rounds. The input size is 64 bits and the supported key lengths by PRESENT are 80 bits and 128 bits. In the original paper [36], the 80-bit version was recommended over the 128-bit version, which was more than adequate and suitable for low-security applications such as small sensor systems, hence the name PRESENT-80.

Each round of PRESENT-80 consists of an XOR operation with the round key K_i (for $1 \leq i \leq 32$), a substitution process with a non-linear layer of S-box, and a linear bitwise permutation process using the pLayer. For the XOR operation, the round keys are generated from the 80-bit supplied key, which was designed to be stored in the key register, denoted as $K = k_{79}k_{78} \ldots k_0$. The round key K_i at the round i is the 64 leftmost bits of the $K = k_{79}k_{78} \ldots k_0$. After the extraction of the round key, the $K = k_{79}k_{78} \ldots k_0$ is updated as follows and demonstrated in (3).

- Rotate the key register K by 61-bit positions to the left.
- The four leftmost bits are substituted using the S-box.
- The bits of $[k_{19}k_{18}k_{17}k_{16}k_{15}]$ are XORed with the current round number counter (5 bits).

$$\begin{array}{ll} 1. & [k_{79}k_{78} \ldots k_1k_0] = [k_{18}k_{17} \ldots k_{20}k_{19}] \\ 2. & [k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}] \\ 3. & [k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus round\_counter \end{array} \quad (3)$$

PRESENT utilized a 4-bit to 4-bit substitution S-box, with the hexadecimal notation of input x and output of the function $S(x)$ given by Figure 3. To fully substitute the 64-bit cipher state, 16 of these S-boxes were utilized in parallel to generate the output.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 3. The substitution function of PRESENT.

The bitwise permutation of PRESENT is given in Figure 4, where i is the bit position of the cipher state being moved to the new position of $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51	4	20	36	52	5	21
i	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
$P(i)$	37	53	6	22	38	54	7	23	39	55	8	24	40	56	9	25	41	57	10	26	42	58
i	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63		
$P(i)$	11	27	43	59	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63		

Figure 4. The permutation position table of PRESENT.

The simplicity of PRESENT and intention to be implemented in hardware were cited as the key design goals. PRESENT would demand roughly the same hardware resources for both encryption and decryption. The S-box, the permutation pLayer, and the FSM for round key scheduling are the three primary components that need to be designed in the implementation. The S-box and pLayer can be implemented in hardware as a bit manipulation module, and the S-box can also be utilized again throughout the round key generation process. For the decryption process, the inverted S-box and inverted pLayer also need to be designed.

4.3. ChaCha

ChaCha [37] is a stream cipher, more specifically, a family of stream ciphers based on a variant of Salsa20 [38,39], with the modified round function increasing the amount of diffusion per round. Both ChaCha and Salsa20 are built on a pseudorandom function, based on ARX (Add-Rotate-XOR) operations: 32-bit addition, rotation operations, and bitwise addition (XOR). The core function maps a 256-bit key, a 64-bit nonce, and a 64-bit counter into a randomly accessible 512-bit block of the keystream. The internal state of ChaCha is formed by 16 32-bit words arranged as a 4×4 matrix, denoted as S.

$$\mathrm{S} = \begin{pmatrix} 61707865 & 3320346E & 79622D32 & 6B206574 \\ \mathrm{key}[0] & \mathrm{key}[1] & \mathrm{key}[2] & \mathrm{key}[3] \\ \mathrm{key}[4] & \mathrm{key}[5] & \mathrm{key}[6] & \mathrm{key}[7] \\ \mathrm{counter}[0] & \mathrm{counter}[1] & \mathrm{nonce}[0] & \mathrm{nonce}[1] \end{pmatrix} \tag{4}$$

The state consists of four words of the constant "expand 32-byte k". The following eight words are for the key, with two for the block counter and the last two for the nonce. Theoretically, ChaCha can generate the keystream up to 2^{70} bytes or 2^{64} blocks of the 512-bit key. The algorithm implemented in this work optimized the original paper of ChaCha (and Salsa20), using two words for the counter and two words for the nonce, as mentioned above. This differs from the ChaCha20 variant used in the standardized IETF protocol in RFC 8439 [40], with the state implementing one word for the counter and three for the nonce.

Depending on the even number of applied rounds, 8 and 20, for example, the respective names would be ChaCha8 and ChaCha20. For each round in ChaCha, the core operation is the quarter round "$QR(a, b, c, d)$" that takes a four-word input and produces a four-word output from the state S. The quarter round performs the ARX operation on the 32-bit words (a, b, c, and d), with "<<<" as the notation for bitwise left rotation, as shown below.

$$\begin{aligned} &\mathrm{a} += \mathrm{b};\ \mathrm{d} \hat{}= \mathrm{a}\ ;\ \mathrm{d}\ \lll = 16; \\ &\mathrm{c} += \mathrm{d};\ \mathrm{b} \hat{}= \mathrm{c}\ ;\ \mathrm{b}\ \lll = 12; \\ &\mathrm{a} += \mathrm{b};\ \mathrm{d} \hat{}= \mathrm{a}\ ;\ \mathrm{d}\ \lll = 8; \\ &\mathrm{c} += \mathrm{d};\ \mathrm{b} \hat{}= \mathrm{c}\ ;\ \mathrm{b}\ \lll = 7; \end{aligned} \tag{5}$$

Four of these quarter rounds performed together on S would then form or be defined as one round of the algorithm. Depending on the round count number, starting from one, odd-numbered rounds apply the quarter round function to each of the four columns in the 4×4 state matrix, and even-numbered rounds apply it to each of the four diagonals. Figure 5 lists the state S (4) and its 32-bit words as a 4×4 table, ranging from 0 to 15. Four quarter rounds in an odd round will operate on those words as defined in (6).

$$\begin{aligned} &\mathrm{QR}(0,\ 4,\ 8,\ 12); \\ &\mathrm{QR}(1,\ 5,\ 9,\ 13); \\ &\mathrm{QR}(2,\ 6,\ 10,\ 14); \\ &\mathrm{QR}(3,\ 7,\ 11,\ 15); \end{aligned} \tag{6}$$

Additionally, four other quarter rounds in an even round will manipulate those words, as in (7). Two consecutive rounds of this odd round and even round is called a double round.

$$\begin{aligned} &\mathrm{QR}(0,\ 5,\ 10,\ 15); \\ &\mathrm{QR}(1,\ 6,\ 11,\ 12); \\ &\mathrm{QR}(2,\ 7,\ 8,\ 13); \\ &\mathrm{QR}(3,\ 4,\ 9,\ 14); \end{aligned} \tag{7}$$

For hardware implementation, the design of the quarter round module alone would significantly speed up the algorithm compared to software implementation. This is because each addition, XOR, and rotation operation would already cost multiple circles when being executed in the processor using standard instructions. The complexity of one round in

the accelerator is expected to be low as it only has 16 additions, 16 XORs, and 16 constant distance rotations of 32-bit words.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 5. The indexed representation of the 4 × 4 matrix state S, ranging from 0 to 15.

4.4. SHA3-512

A cryptographic hash function is a mathematical algorithm that maps "message" data of arbitrary size to a bit array of a fixed size "digest" output message. It is a one-way function and it is practically infeasible to invert or reverse the computation to obtain the original message, except for brute-forcing it. SHA-3 [41] is the latest member of the Secure Hash Algorithm family of standards by NIST, with the predecessors being SHA-2 and SHA-1. The NIST defined four instances in the SHA-3 standard for different digest lengths, including SHA3-224, SHA3-256, SHA3-384, and SHA3-512.

SHA-3 is a sub-set of a broader cryptographic primitive family called Keccak and is based on a new design approach called sponge construction, which is a comprehensive collection of random functions or permutations. This allows inputting, or "absorbing", any amount of data and outputting, or "squeezing", any amount of data. Figure 6 below describes the sponge construction in the SHA-3 hash functions.

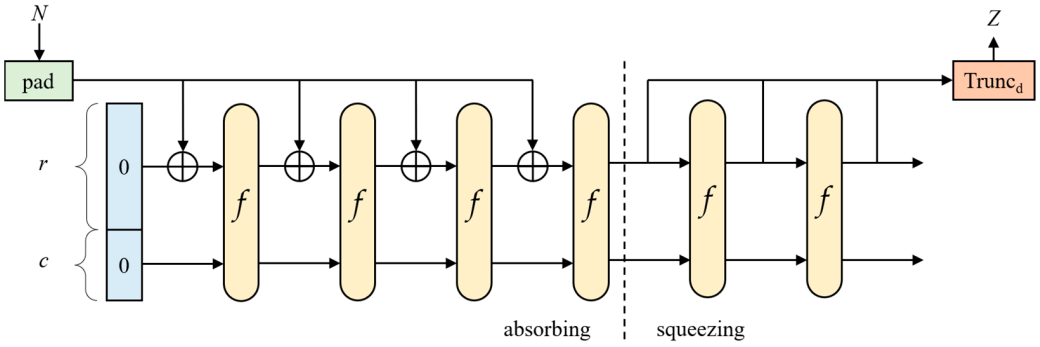

Figure 6. The sponge construction, SHA-3 standard: permutation-based hash and extendable output functions.

In Figure 6, for SHA3-512 to output 512 bits of digest, the input message in the form of a bit string N was padded using the *pad10*1* pattern padding functions to a multiple of 576 (bits), which is called the rate, denoted as r. The c is called the capacity and the state of SHA-3 is defined as a bit string with the length of $b = r + c$, containing all zero bits in it at the beginning. The c value in SHA3-512 is 1024 (bits), which makes b 1600 (bits). The padded input is then "absorbed" and fed into 24 rounds of block transformation f, which is *Keccak[1024](M || 01, 512)*. The number of "absorbing" times is dependent on the length of the input, specifically, the number of 576-bit segments of the padded input. For example, if the input bit string message is 500 bits, then it will be padded to the length of 576 bits, and only one "absorbing" operation will be needed. In the "squeezing" stage, segments with the length of r are collected to the bit string Z until the number of output bits is met, which is 512 bits for SHA3-512. Z will then be truncated to the output length, and will then be the digest of the message, completing the hash operation.

A small note on SHA-3 is that it is not considered as a lightweight hash function, and the standardization publication from NIST does not address this either. However, there have been some works [42] on lightweight cryptographic algorithms implemented based on the sponge construction and instances of Keccak, which are parts of SHA-3. The SHA3-512 hash function accelerator core was integrated into the SoC to diversify the type of algorithms that can be applied.

5. Implementations

5.1. The PRINCE Accelerator Core

The integrated PRINCE cipher accelerator core in the SoC was designed using Verilog HDL. Figure 7 presents the overall internal architecture of the implementation with a simple but flexible interface, and the connection diagram with the Configuration/Status Register (CSR) peripheral bus of the proposed SoC.

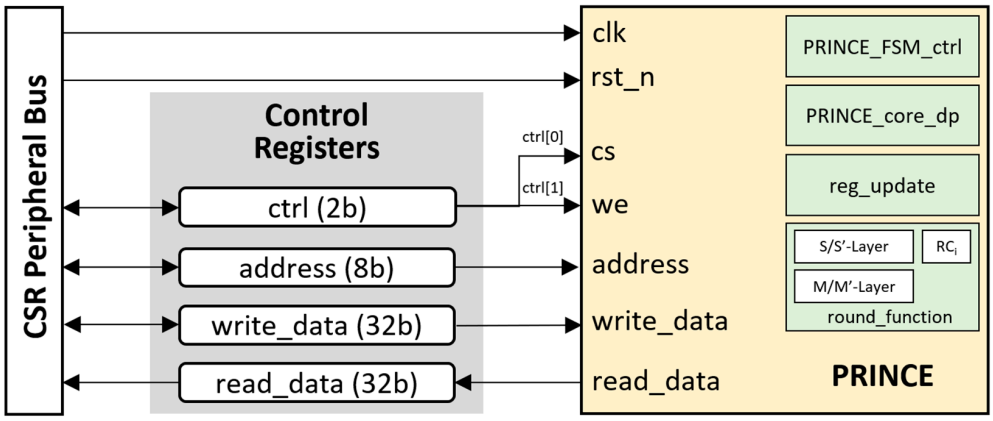

Figure 7. The PRINCE accelerator architecture with bus interface.

Accessing and communicating with the hardware core is achieved through the process of reading and writing data values to registers with specified addresses in the peripheral bus, represented and connected to the data ports of the accelerator. The PRINCE accelerator core possesses the base address of 0x3000 in the CSR peripheral bus, assigned by the SoC builder. By implementing this simple "register-controlled interface" scheme, two dedicated functions to control the accelerator are formed:

- "prince_write_to_address(address, data)": This function prepares the "write_data" and "address" for the accelerator core and then simulates the core with "cs" and "we" for the writing process. This function writes 32-bit data or configuration values for the 8-bit address in the PRINCE accelerator core.
- "prince_read_from_address(address)": The function simulates the core with "cs" and "we" for the reading process and reads the output from the accelerator core after preparing the "address" and ignoring the "write_data". This function reads 32-bit data from an 8-bit address in the PRINCE accelerator core.

The internal register address map of the accelerator used in the software interface is shown in Table 1.

Table 1. Internal register address mapping for the PRINCE accelerator.

Name	Address	Description
Key Input	0x10–0x13	128-bit key input registers
Block Input	0x20–0x21	64-bit message input registers
Result	0x30–0x31	64-bit cipher output registers
Control	0x08	Accelerator control bit
Status	0x09	Accelerator status bit
Configuration	0x0A	Accelerator configuration bit

5.2. The PRESENT-80

The integrated PRESENT-80 accelerator core of the SoC was designed and configured the same way as the PRINCE core. The connection diagram between the PRESENT-80 accelerator and the SoC is shown in Figure 8. Internally, the accelerator consists of a key scheduler for generating round keys and two separate sub-modules for the encryption and decryption process.

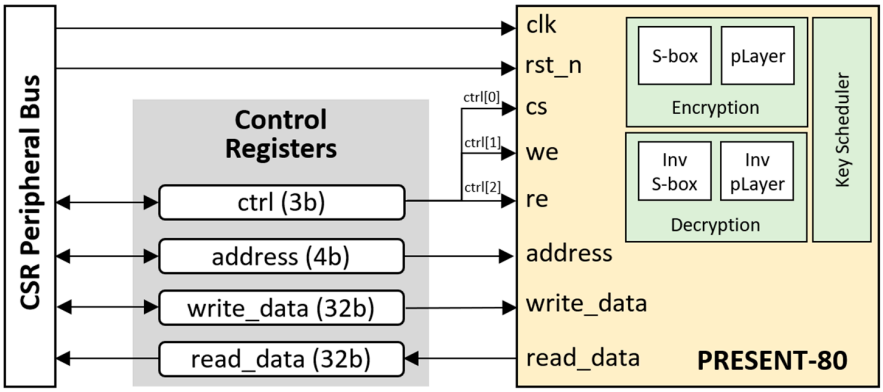

Figure 8. The PRESENT-80 accelerator architecture with bus interface.

This hierarchical structure simplified the hardware design of the accelerator, which could then be accessed and configured at the address of 0x2800 in the CSR peripheral bus after being integrated. The offset addresses are listed in Table 2.

Table 2. Internal register address mapping for the PRESENT-80 core.

Name	Address	Description
Configuration	0x00	Configuration of encryption or decryption
Key Input	0x01–0x03	80-bit key input register
Data Input	0x04–0x05	64-bit data input register
Cipher Output	0x06–0x07	64-bit cipher output register

5.3. The ChaCha Accelerator Core

The architecture of the accelerator core for ChaCha used in the SoC was also written in Verilog HDL and connected using the same principle. The connection diagram between ChaCha core and the SoC bus is shown in Figure 9. This implementation provides some configurable elements, such as the round number, the optional 128-bit key input, and the direct encryption output of a 512-bit message block (XORed with the 512-bit block of the keystream). The accelerator core can be accessed at the address of 0x0200 in the CSR peripheral bus. The offset addresses are listed in Table 3.

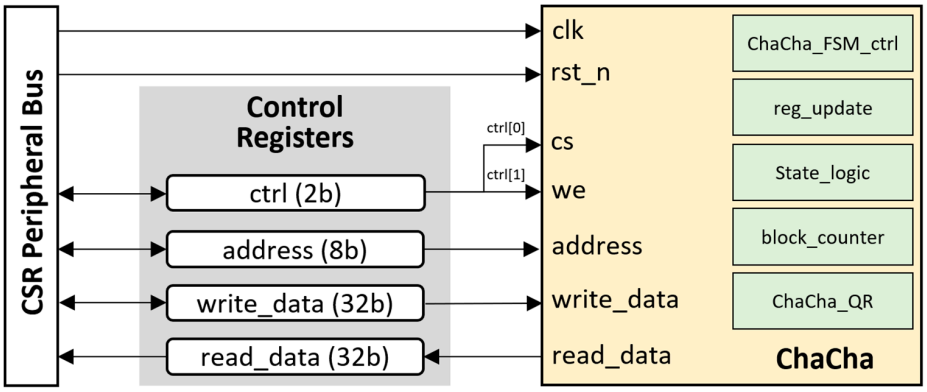

Figure 9. The ChaCha accelerator architecture with bus interface.

Table 3. Internal register address mapping for the ChaCha accelerator.

Name	Address	Description
Key Input	0x10–0x17	256-bit key input registers
Nonce Input	0x20–0x21	64-bit number-use-once registers
Data Input	0x40–0x4F	Optional 512-bit message input registers
Data Output	0x80–0x8F	512-bit keystream output registers
Control	0x08	Accelerator control bit
Status	0x09	Accelerator status bit
Configuration	0x30	Accelerator configuration bit

5.4. The SHA3-512 Accelerator Core

The SHA3-512 accelerator core used in this work was slightly improved from an open-source project [43]. The modification was achieved for the padding module of the accelerator. Since the [43] core was developed for Keccak, the padding specification was to add to the input message a 1 bit, followed by a pre-determined amount of 0 bit, and to end with a 1 bit. This padding scheme changed when Keccak was standardized to SHA-3, with the pattern of adding, in hexadecimal form, a 0x06 byte, followed by numbers of 0x00, and ending with a 0x80 byte. There was no change in terms of resources after this modification. The connection diagram between the SHA-3 cores and the SoC is shown in Figure 10. A wrapper was then created using a simplified interface similar to the PRINCE and ChaCha cores and written in Verilog HDL. The core was assigned to the address of 0x4000 in the CSR bus and the internal register address mapping is shown in Table 4 below.

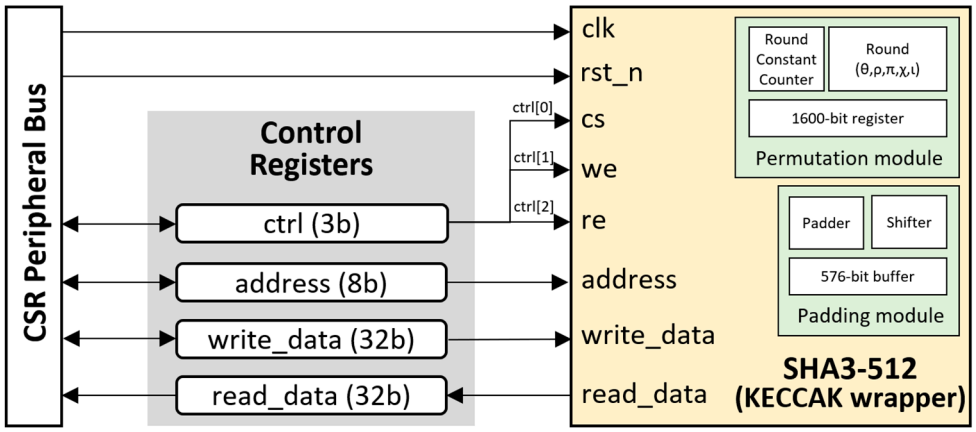

Figure 10. The SHA3-512 accelerator architecture with bus interface.

Table 4. Internal register address mapping for the SHA3-512 core.

Name	Address	Description
Reset	0x00	Set and reset input
Input	0x01	32-bit input register
Byte Number	0x02	Set index of last byte
Input Last	0x03	Set the last input and start
Status	0x09	Accelerator status bit
Hash Output	0x10–0x1F	512-bit hash output registers

5.5. The System-on-Chip

In this paper, LiteX [44] was used as an SoC building framework, interconnecting components and invoking suitable toolchains to synthesize and deploy the design in an FPGA. The VexRiscv processor core with the Wishbone bus configuration was the first component to be initialized in the system, along with other peripherals, forming a basic SoC. All accelerator core designs in Verilog HDL were added later in the process. LiteX also

supports generating essential project files and constraints for the respective FPGA platform toolchain, which was Vivado in this work.

Utilizing Migen, a Python-based fragmented hardware description language, the LiteX framework enables hardware cores and SoC systems to be designed with ease, experimenting with various digital design architectures and implementing them in various FPGA hardware platforms. By using the framework alongside the hardware cores library provided by the LiteX open-source community, including VexRiscv, a more-accessible building process of large and complex SoCs can be made, improving the portability and flexibility of the design process.

The SoC was generated with the following configuration after calling the SpinalHDL generation process for the VexRiscv processor; the framework then started to compile and elaborate the system design based on the building script, and then started the FPGA synthesizing process, building the bitstream.

- Single-core 32-bit VexRiscv processor (RV32IM), 32-bit Wishbone Bus with 4 GB address space, 8 KB L1 cache (4 KB data cache and 4 KB instruction cache), and 8 KB L2 cache.
- Peripherals: UART, SPI, GPIO.
- Custom cryptographic accelerator cores for PRINCE, PRESENT-80, ChaCha, and SHA3-512.

The proposed SoC architecture composed of a VexRiscv processor and lightweight cryptographic cores is presented in Figure 11. As briefly mentioned for the PRINCE accelerator core in Section 4.1, the Configuration/Status Register bus in LiteX is a mechanism for reading and writing values for the configuration registers of various intellectual property (IP) cores within the FPGA. It should not be conflated with CSRs as defined by the RISC-V ISA. This bus is used to control and monitor the behavior of the IP cores, such as setting up clock frequencies, enabling or disabling features, and reading status information. The CSR bus is an important feature in the LiteX SoC builder tool in building the proposed SoC, as it allows for a simple and flexible interface with the IP cores, making it easy to configure and control various components, including all of the lightweight cryptographic accelerator cores.

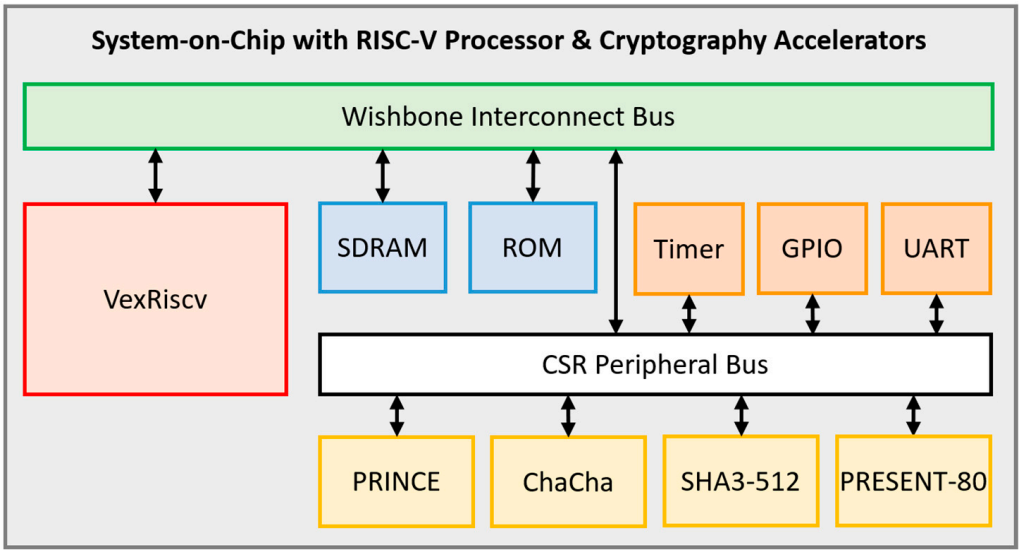

Figure 11. SoC architecture with lightweight cryptographic and hash function accelerator cores.

6. Experimental Results

6.1. FPGA Implementation Results

IO configuration and project constraints for the Nexys4 DDR FPGA development board were generated alongside the Verilog RTL for the SoC as a ".xdc" file. The design was synthesized and implemented using Vivado 2021.2, with the resource utilization

shown in Table 5. There were no DRC violations, and the STA (static timing analysis) report was cleaned. The clock source for the Nexys4 DDR was from an onboard 100 MHz crystal, whereas the VexRiscv core SoC was driven by an MMCM (Xilinx Mixed-Mode Clock Manager) at 75 MHz. The choice of using a frequency of 75 MHz in the SoC, rather than aiming for the highest frequency, was made with the goal of better representing the system as an IoTs device. This frequency was chosen since it offers a compromise between efficiency and performance, which is crucial for Internet of Things devices. By limiting the operation to 75 MHz, the SoC can already perform the necessary computations, which will be discussed later, while still being able to conserve energy if needed.

Table 5. Synthesis result for Xilinx Nexys4 DDR XC7A100TCSG324-1 FPGA.

	Top Design *	VexRiscv	PRINCE	PRESENT-80	ChaCha	SHA3-512
LUTs	11,830	2025	1561	489	2890	3003
Logic LUTs	11,684	2025	1561	489	2890	3003
LUTs as RAMs	144	0	0	0	0	0
SRLs	2	0	0	0	0	0
FFs	9552	1279	648	901	1960	2291
RAMB	36	9	0	0	0	0
DSP	4	4	0	0	0	0

* The total utilized resources consist of accelerator cores, also shown in the table above, and many other components that make up the system (e.g., the internal bus, basic peripherals).

6.2. Accelerator Core Function Verifications

All cryptographic accelerator cores connected to the SoC through the peripheral bus were evaluated using custom firmware, developed and compiled using the standard and open-source GNU Compiler Collection (GCC) RISC-V toolchain.

The firmware/software interface for each cryptographic component was developed using the mapping address information from Tables 1–4. The complete cipher functions and verification test cases for the PRINCE, PRESENT-80, ChaCha, and SHA3-512 algorithms are also included. The verification results of the PRINCE, PRESENT-80, ChaCha, and SHA-3 cores are displayed in Figures 12–16, respectively. This evaluation affirmed that all functional components were in the proper operation condition.

```
[LITEX-TERM] Done.
Executing booted program at 0x40000000

--============= Liftoff! ===============--
     ___ _____ ____ _        _      ____
    |  _ \| ____/ ___|| |      / \  | __ )
    | | | |  _| \___ \| |     / _ \ |  _ \
    | |_| | |___ ___) | |___ / ___ \| |_) |
    |____/|_____|____/|_____/_/   \_\____/

      RISC-V 32-bit LiteX SoC on FPGA(s)
   with Lightweight Cryptography Accelerators

RISC-V SoC minimal firmware with cipher functions - Feb 3 2023 23:06:08

Available commands:
help               - Show this help command
reboot             - SoC Reboot
led                - LED GPIO test
--
chacha             - ChaCha Cipher test
prince             - PRINCE Cipher test
sha3               - SHA3-512 Hash Algorithm test
present            - PRESENT Cipher test
--
chachacompare      - ChaCha Cipher implementations comparison
princecompare      - PRINCE Cipher implementations comparison
sha3compare        - SHA3-512 Hash Algorithm implementations comparison
presentcompare     - PRESENT Cipher implementations comparison
litex-vexriscv-soc>
```

Figure 12. Custom firmware with interactive console, running in the SoC.

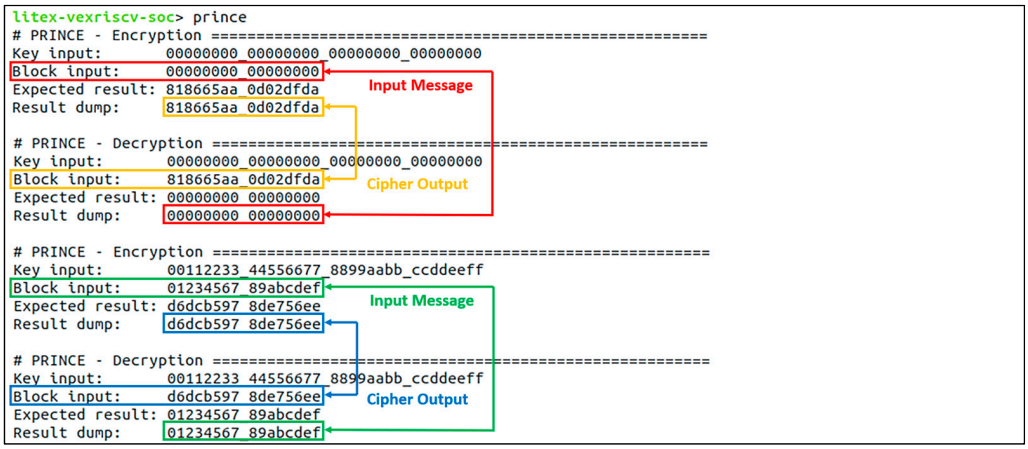

Figure 13. Verification result for the block cipher PRINCE. Two simple test cases in the electronic codebook (ECB) mode for the algorithm were evaluated. Each case verified the encryption and decryption function of the accelerator core and correctly outputted the expected data.

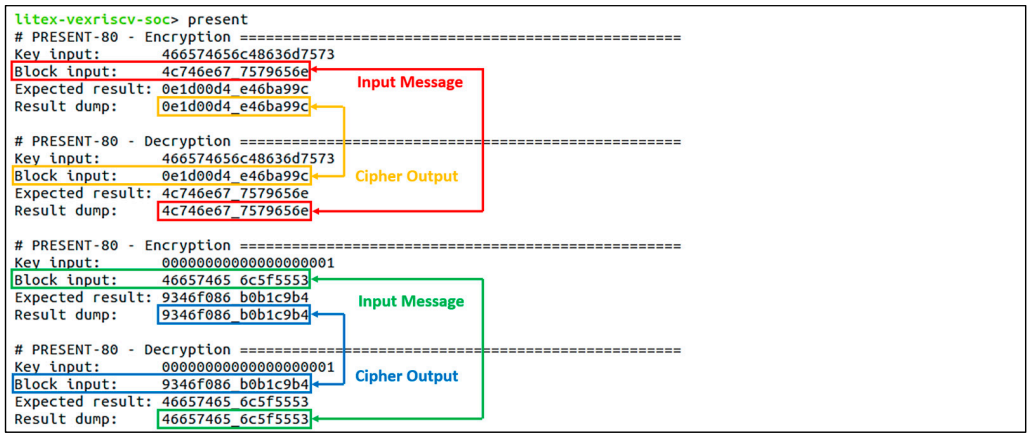

Figure 14. Verification result for the PRESENT-80 block cipher. Two simple test cases verified the encryption and decryption function of the accelerator core and correctly returned the expected data.

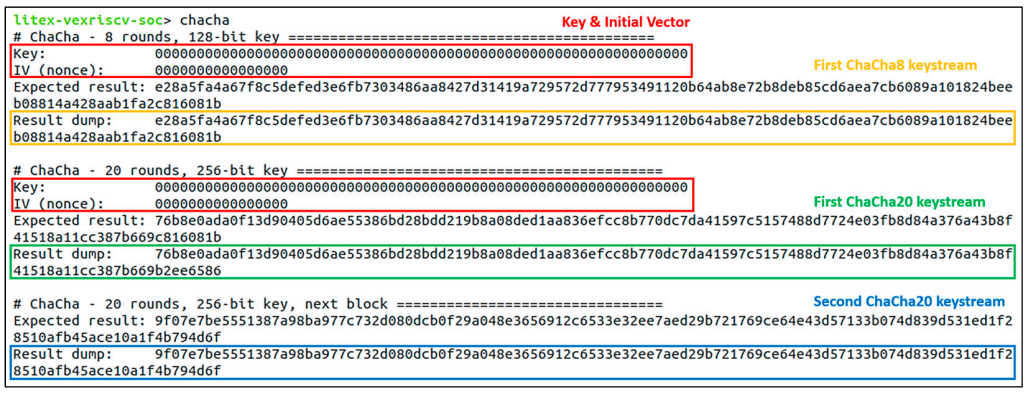

Figure 15. Verification result for the stream cipher ChaCha with 8 and 20 rounds of configuration, using the same 256-bit key and 64-bit initial vector, outputting 512-bit keystreams (blocks), which met the expected results.

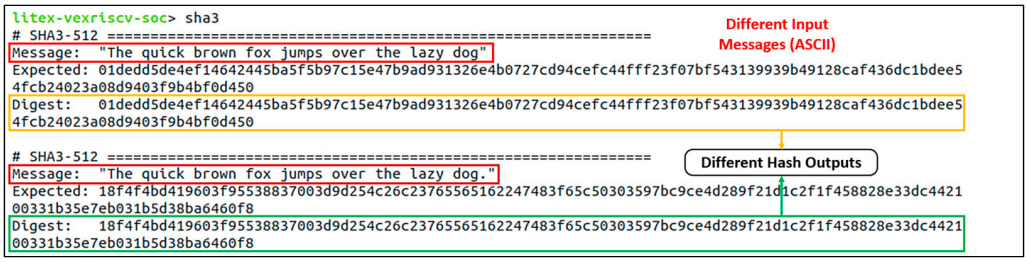

Figure 16. Verification result for the SHA3-512 hash function. Two test cases of a small different ASCII input message (inside the quotation marks) led to two completely different but expected hash output results.

6.3. Cryptographic Processing Speed

Using the SoC's internal timer, the number of executing cycles consumed by functions executed in the SoC's hardware accelerator core and software implementation can be determined. This was accomplished by modifying the verification function for each accelerator core to use the timer and begin counting each system clock cycle until the result from the test case was returned from the accelerator. For example, the PRINCE encryption test case was measured, as shown in Figure 17. Similar approaches were also created for the software implementation of each algorithm, except for PRESENT-80, which was also executed in the SoC, measuring only one simple test case of each algorithm.

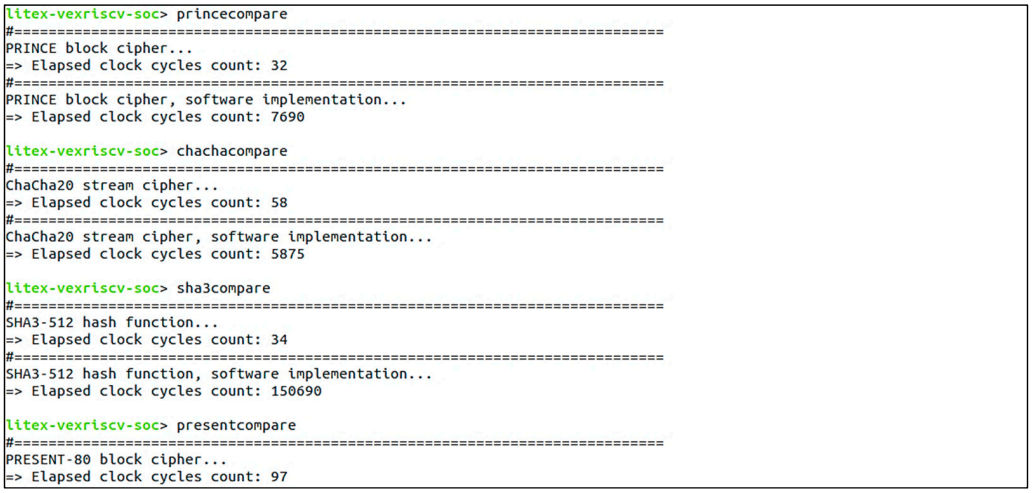

Figure 17. Hardware and software implementations executing cycle comparison.

In comparison to the software implementation of all of the algorithms, the hardware accelerator cores had a much faster processing speed, with the fastest taking 4400 times fewer processing cycles. Table 6 compares the number of cycles required to execute a single test case in the hardware and software implementation of four algorithms. These performance gains demonstrated the efficacy of using a hardware accelerator for a cipher instead of executing it in software, allowing small devices, such as Internet of Things devices, to perform other tasks. The absence of a software implementation for PRESENT-80 was the case since the PRESENT block cipher was designed to be hardware-optimized in the first place, and a software implementation would provide no comparison value. Prior to this, numerous software implementations were attempted, and the processing clock cycle count varied between a few hundred and a few million [26].

Table 6. Number of average execution cycles from 100 iterations, rounded up to whole numbers.

	Hardware	Software	Gains
PRINCE	32 cycles	7682 cycles	240
PRESENT-80	97 cycles	-	-
ChaCha20	58 cycles	5876 cycles	101
SHA3-512	34 cycles	150,693 cycles *	4432

* An open-source software implementation [45] of SHA3.

7. Evaluations and Discussions

As a comparison to other works, shown in Table 7, the proposed system implementations stand out in some areas. Firstly, the lower resource usage can be seen as an advantage with PRINCE, using only 392 slices compared to 956 and 539 slices in [24] and [25], respectively. The same lower slice usage can also be seen with the PRESENT-80 core, using 123 slices compared to 460, and with the SHA3-512 core: only 750 compared to 3117 and 1192 slices. The downside in terms of resource usage can only be seen with the ChaCha core when compared to [27], which was designed to be a compact and high-performance implementation. However, all four implementations of PRINCE, PRESENT-80, ChaCha, and SHA3-512 with the proposed system occurred without employing any RAM blocks. From the power perspective, lower power dissipation results were obtained alongside the synthesis report, with the smallest value of 8 mW for the ChaCha core. Other implementation results were not available for further comparison.

Table 7. Summary and comparison result with other standalone cipher implementations.

Algorithm	Output Size (Bit)	Device	Max. Frequency (MHz)	Throughput (Mbps)	Slices	Block RAM	Power (Watts)
PRINCE on SoC	**64**	**Artix-7 XC7A100T**	**75** [1]	**149**	**392**	**0**	**0.054** [4]
PRINCE [24]	64	Virtex-4 FF668	31.765	2032	956	-	0.165
PRINCE-BB84 [25]	64	Kintex-7	61.42	3931	539	-	0.045
PRESENT-80 on SoC	**64**	**Artix-7 XC7A100T**	**75** [1]	**49**	**123**	**0**	**0.003** [4]
PRESENT-80 [26]	64	Spartan-3E	50	6.25 [2]	460	-	0.044
ChaCha20 on SoC	**512**	**Artix-7 XC7A100T**	**75** [1]	**662**	**723**	**0**	**0.008** [4]
ChaCha20 [27]	512	Virtex-6 XC6VLX75T	345	254	77	2	-
SHA3-512 on SoC	**512**	**Artix-7 XC7A100T**	**75** [1]	**1129**	**750**	**0**	**0.211** [4]
SHA3-512 [28]	512	Virtex-5	223	5350	1192	-	-
SHA3-512 [29]	512	Arria 10	312	22,360	149,500	787	-
SHA3-512 [30]	512	ASIC—TSMC 7nm	5100	115,200	30,740 [3]	-	0.025

[1] The accelerator is connected to the clock signal from the system bus. [2] Estimated results based on system operating frequency and number of clock cycles from [26]. [3] The ASIC implementation reported in GE (gate equivalent). [4] Power dissipation obtained via synthesis report.

The main difference that can be extracted from Table 7 is the throughput of each implementation. Compared to [24,25], these standalone implementations have much higher throughput with the largest gap of 26 times compared to the integrated PRINCE accelerator. Compared to [26], the integrated PRESENT-80 accelerator in this work has higher performance with a throughput gain of 7.84 times while using less resources and less power dissipation. For ChaCha and SHA3-512, much higher operating frequencies were used to evaluate the others' performance, while the implementations in this work used the same clock frequency from the SoC, only at 75 MHz. Compared to [27], the integrated ChaCha accelerator has 2.6 times higher throughput, operating at 4.6 times lower frequency.

The integrated SHA3-512 accelerator also reached a throughput of 1.129 Gbps. Compared to [28,29], these SHA3-512 standalone implementations had higher throughput but with the cost of much higher operating frequencies and a higher usage of resources, which was not adequate for small and medium systems. We also added the state-of-the-art result of [30] into the comparison. The proposed work supports SHA3-512 and was verified in an FPGA before being synthesized in ASIC using TSMC 7 nm technology. The high through-

put once again shows the benefit of using a dedicated accelerator for specific applications. When scaling down the throughput using the same operating frequency in our system, the throughput was also comparable at 1.694 Gbps with well-optimized architecture in ASIC. The overhead when integrated into the SoC can be used to explain why the throughput is reduced when compared to certain other works. Using the simple register-controlled interface meant that the number of clock cycles to configure, execute the cipher, and acquire the output became costly. Thus, operating the SoC at a higher frequency may increase the throughput drastically.

A further comparison is made with the work of [31] in Table 8, which developed PRINCE and PRESENT-80 ciphers as part of the RISC-V Extension for Lightweight Cryptography, including custom hardware instructions. The results for PRINCE were equivalent, with approximately the same resource use and execution latency, while the PRESENT-80 in this work was able to execute the cipher with less time. Other relevant works, such as [26], implemented the PRESENT-80 lightweight cryptographic block cipher and the AES-128 classical block cipher as system accelerators. The results showed a longer execution time with increased resource and power consumption. The comparable work of [32] was also examined, which established the E31 (RV32IMAC) with optimized instructions for ChaCha20 and Keccak-f[1600] (SHA-3). The results demonstrated that even with optimized and customized instructions, which were an advantage of RISC-V, the number of cycles required to encrypt and decrypt the ChaCha20 stream cipher remained very high. The number of execution cycles for Keccak-f[1600] seems to be an improvement, but it was still unable to match the benefit of using the integrated system accelerator.

Table 8. Comparison to other relevant system implementations.

	This Work	**[31]**	**[32]**	**[26]**
Device	**Artix-7 XC7A100T**	Artix-7 XC7A100T	FE310-G000 SoC	Spartan-3E
Processor core	**VexRiscv**	VexRiscv	SiFive E31	MicroBlaze
Operation frequency	**75 MHz**	-	-	50 MHz
Implementation type	**SoC Accelerator**	HW instructions	SW instructions	SoC accelerator
		Algorithm performance (cycles)		
PRINCE	**31**	35 [1]	-	-
ChaCha20	**58**	-	1787 [2]	-
SHA3-512 (Keccak)	**34**	-	13,774	-
PRESENT-80	**97**	160 [1]	-	514 [3]

[1] The work of [31] reported the results as the number of instructions. [2] The estimated result of [32] for processing 64 bytes originally reported processing 4096 bytes. [3] The estimated average result for [26] originally reported 100 processing iterations.

8. Conclusions

In this paper, the proposed SoC design, based on a 32-bit VexRiscv processor and cryptographic accelerator cores for PRINCE and PRESENT-80, two lightweight block ciphers, ChaCha, a lightweight stream cipher, and SHA3-512, a hash function, was successfully deployed with fewer implementation clock cycles in the Nexys4 DDR FPGA development board. In addition to the lightweight cryptographic accelerator cores, the VexRiscv SoC's functionality was validated and demonstrated to be stable, functional, and to possess high throughput. The number of utilized resources by the system in an FPGA was also relatively low, 11,830 LUTs and 9552 FFs, meaning it is eligible for compact Internet of Things application systems, with the advantage of significantly reducing the execution time resulting in low latency of lightweight cryptographic algorithms. Specifically, the PRINCE core requires 31 cycles, PRESENT-80 requires 97 cycles, Chacha20 needs 58 cycles, and SHA-3 takes 34 cycles to be implemented in the proposed SoC. Low latency enables the system to process with minimal delay and improves system performance.

In conclusion, this work has introduced a highly customizable platform using an RISC-V processor. The highlighted contribution was the usage of the simple peripheral bus

in LiteX, based on a register-controlled interface, and the Configuration/Status Register to connect customized and optimized cores, such as PRINCE, PRESENT-80, ChaCha, and SHA-3, with VexRiscv to construct a high-performance SoC with low-implementation clock cycles and low-logic resources. This allows not only cryptographic accelerators but also various hardware cores to be integrated, forming a simple and custom SoC. The proposed SoC can be applied in small and medium systems such as Internet of Things systems, customized security IoTs devices, compact root-of-trust systems, etc. It can also be used to deploy and develop more customized and complicated systems.

Author Contributions: Writing—original draft preparation, methodology, software, K.-M.M.; conceptualization, validation, supervision, research administration, D.-H.L.; review, C.-K.P.; data analysis, review and editing, T.-T.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the University of Science, VNU-HCM, under grant number ĐT-VT 2023-01.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

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

References

1. Waterman, A.; Lee, Y.; Patterson, D.A.; Asanovic, K. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*; EECS Department, University of California: Berkeley, CA, USA, 2016.
2. Calicchia, L.; Ciotoli, V.; Cardarilli, G.C.; di Nunzio, L.; Fazzolari, R.; Nannarelli, A.; Re, M. Digital Signal Processing Accelerator for RISC-V. In Proceedings of the 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genoa, Italy, 27–29 November 2019; pp. 703–706. [CrossRef]
3. Zhang, G.; Zhao, K.; Wu, B.; Sun, Y.; Sun, L.; Liang, F. A RISC-V based hardware accelerator designed for Yolo object detection system. In Proceedings of the 2019 IEEE International Conference of Intelligent Applied Systems on Engineering (ICIASE), Fuzhou, China, 26–29 April 2019; pp. 9–11. [CrossRef]
4. Holler, R.; Haselberger, D.; Ballek, D.; Rossler, P.; Krapfenbauer, M.; Linauer, M. Open-Source RISC-V Processor IP Cores for FPGAs—Overview and Evaluation. In Proceedings of the 2019 8th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 10–14 June 2019; pp. 1–6.
5. RISC-V Summit 2022: All Your CPUs Belong to Us. Available online: https://www.eetimes.com/risc-v-summit-2022-all-your-cpus-belong-to-us/ (accessed on 12 May 2023).
6. A Winning Processor Portfolio. Available online: https://www.sifive.com/risc-v-core-ip (accessed on 12 May 2023).
7. Harris, S.L.; Chaver, D.; Piñuel, L.; Gomez-Perez, J.I.; Liaqat, M.H.; Kakakhel, Z.L.; Kindgren, O.; Owen, R. RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. In Proceedings of the 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021; pp. 145–150. [CrossRef]
8. Wu, W.; Su, D.; Yuan, B.; Li, Y. Intelligent Security Monitoring System Based on RISC-V SoC. *Electronics* **2021**, *10*, 1366. [CrossRef]
9. Lee, D.; Moon, H.; Oh, S.; Park, D. mIoT: Metamorphic IoT Platform for On-Demand Hardware Replacement in Large-Scaled IoT Applications. *Sensors* **2020**, *20*, 3337. [CrossRef] [PubMed]
10. Zhang, H.; Wu, X.; Du, Y.; Guo, H.; Li, C.; Yuan, Y.; Zhang, M.; Zhang, S. A Heterogeneous RISC-V Processor for Efficient DNN Application in Smart Sensing System. *Sensors* **2021**, *21*, 6491. [CrossRef] [PubMed]
11. Lim, S.-H.; Suh, W.W.; Kim, J.-Y.; Cho, S.-Y. RISC-V Virtual Platform-Based Convolutional Neural Network Accelerator Implemented in SystemC. *Electronics* **2021**, *10*, 1514. [CrossRef]
12. Gamino del Río, I.; Martínez Hellín, A.; Polo, Ó.R.; Jiménez Arribas, M.; Parra, P.; da Silva, A.; Sánchez, J.; Sánchez, S. A RISC-V Processor Design for Transparent Tracing. *Electronics* **2020**, *9*, 1873. [CrossRef]
13. Lee, J.; Chen, H.; Young, J.; Kim, H. RISC-V FPGA Platform Toward ROS-Based Robotics Application. In Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 31 August–4 September 2020; p. 370. [CrossRef]
14. Santos, D.A.; Luza, L.M.; Zeferino, C.A.; Dilillo, L.; Melo, D.R. A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems. In Proceedings of the 2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS), Marrakech, Morocco, 1–3 April 2020; pp. 1–5. [CrossRef]
15. Andersson, J. Development of a NOEL-V RISC-V SoC Targeting Space Applications. In Proceedings of the 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Valencia, Spain, 29 June–2 July 2020; pp. 66–67. [CrossRef]

16. El-hajj, M.; Mousawi, H.; Fadlallah, A. Analysis of Lightweight Cryptographic Algorithms on IoT Hardware Platform. *Future Internet* **2023**, *15*, 54. [CrossRef]
17. Cheng, H.; Großschädl, J.; Marshall, B.; Page, D.; Pham, T. RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography. *TCHES* **2022**, *2023*, 193–237. [CrossRef]
18. Wei, M.; Yang, G.; Kong, F. Software Implementation and Comparison of ZUC-256, SNOW-V, and AES-256 on RISC-V Platform. In Proceedings of the 2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE), Chengdu, China, 19–21 March 2021; pp. 56–60. [CrossRef]
19. 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]
20. Marshall, B.; Newell, G.R.; Page, D.; Saarinen, M.-J.O.; Wolf, C. The design of scalar AES Instruction Set Extensions for RISC-V. *Cryptol. Eprint Arch.* **2020**, 1–28. [CrossRef]
21. Pojoga, G.; Papagiannopoulos, K. Low-Latency Implementation of the GIFT Cipher on RISC-V Architectures. In Proceedings of the 19th ACM International Conference on Computing Frontiers, Turin, Italy, 17–19 May 2022; pp. 287–295. [CrossRef]
22. Altınay, Ö.; Örs, B. Instruction Extension of RV32I and GCC Back End for Ascon Lightweight Cryptography Algorithm. In Proceedings of the 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS), Barcelona, Spain, 23–25 August 2021; pp. 1–6. [CrossRef]
23. Zheng, X.; Wu, J.; Lin, X.; Gao, H.; Cai, S.; Xiong, X. Hardware/Software Co-design of Cryptographic SoC Based on RISC-V Virtual Prototype. *IEEE Trans. Circuits Syst. II Express Briefs*, 2023; *Early Access*. [CrossRef]
24. Abbas, Y.A.; Jidin, R.; Jamil, N.; Z'aba, M.R.; Rusli, M.E.; Tariq, B. Implementation of PRINCE algorithm in FPGA. In Proceedings of the 6th International Conference on Information Technology and Multimedia (ICMI), Putrajaya, Malaysia, 8–10 December 2014; pp. 1–4. [CrossRef]
25. Abdullah, A.A.; Obeid, N.R. Efficient Implementation for PRINCE Algorithm in FPGA Based on the BB84 Protocol. *J. Phys. Conf. Ser.* **2021**, *1818*, 012216. [CrossRef]
26. Guo, X.; Chen, Z.; Schaumont, P. Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*; Bereković, M., Dimopoulos, N., Wong, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5114, pp. 106–115.
27. At, N.; Beuchat, J.L.; Okamoto, E.; San, İ.; Yamazaki, T. Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2014**, *61*, 485–498. [CrossRef]
28. Sundal, M.; Chaves, R. Efficient FPGA Implementation of the SHA-3 Hash Function. In Proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Bochum, Germany, 3–5 July 2017; pp. 86–91. [CrossRef]
29. Bensalem, H.; Blaquière, Y.; Savaria, Y. An efficient OpenCL-Based implementation of a SHA-3 co-processor on an FPGA-centric platform. *IEEE Trans. Circuits Syst. II Express Briefs* **2022**, *70*, 1144–1148. [CrossRef]
30. Nannipieri, P.; Bertolucci, M.; Baldanzi, L.; Crocetti, L.; Di Matteo, S.; Falaschi, F.; Fanucci, L.; Saponara, S. SHA2 and SHA-3 Accelerator Design in a 7 nm Technology within the European Processor Initiative. *Microprocess. Microsyst.* **2021**, *87*, 103444. [CrossRef]
31. Tehrani, E.; Graba, T.; Merabet, A.S.; Danger, J.-L. RISC-V Extension for Lightweight Cryptography. In Proceedings of the 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 26–28 August 2020; pp. 222–228. [CrossRef]
32. Stoffelen, K. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology—LATINCRYPT 2019*; Schwabe, P., Thériault, N., Eds.; Springer: Cham, Switzerland, 2019; Volume 11774, pp. 323–340.
33. RISC-V Foundation. Rocket Chip Generator. 2019. Available online: https://github.com/chipsalliance/rocket-chip (accessed on 7 January 2023).
34. Petrisko, D.; Gilani, F.; Wyse, M.; Jung, D.C.; Davidson, S.; Gao, P.; Zhao, C.; Azad, Z.; Canakci, S.; Veluri, B.; et al. BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs. *IEEE Micro* **2020**, *40*, 93–102. [CrossRef]
35. Borghoff, J.; Canteaut, A.; Güneysu, T.; Kavun, E.B.; Knezevic, M.; Knudsen, L.R.; Leander, G.; Nikov, V.; Paar, C.; Rechberger, C.; et al. PRINCE—A Low-Latency Block Cipher for Pervasive Computing Applications. In *Advances in Cryptology—ASIACRYPT 2012*; Wang, X., Sako, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7658, pp. 208–225.
36. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.; Seurin, Y.; Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007: 9th International Workshop, Vienna, Austria, 10–13 September 2007. Proceedings 9*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466.
37. Bernstein, D.J. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*; The University of Illinois at Chicago: Chicago, IL, USA, 2008; Volume 8, pp. 3–5.
38. Bernstein, D.J. The Salsa20 family of stream ciphers. In *New Stream Cipher Designs*; Robshaw, M., Billet, O., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 4986, pp. 84–97.
39. Bernstein, D.J. Salsa20 Specification. 2005. Available online: http://www.ecrypt.eu.org/stream/salsa20pf.html (accessed on 24 December 2022).
40. Nir, Y.; Langley, A. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439. June 2018. Available online: https://www.rfc-editor.org/rfc/rfc8439 (accessed on 24 December 2022).

41. Dworkin, M.J. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Standard NIST FIPS-202. August 2015. Available online: https://doi.org/10.6028/NIST.FIPS.202 (accessed on 24 December 2022).
42. Kavun, E.B.; Yalcin, T. A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 258–269.
43. OpenCores. SHA3 (KECCAK). Available online: https://opencores.org/projects/sha3 (accessed on 24 December 2022).
44. EnjoyDigital. LiteX. Available online: https://github.com/enjoy-digital/litex (accessed on 19 February 2022).
45. Saarinen, M.-J.O. Very Small, Readable Implementation of the SHA3 Hash Function. Available online: https://github.com/mjosaarinen/tiny_sha3 (accessed on 24 December 2022).