

## Research Article

# A Lightweight AES Coprocessor Based on RISC-V Custom Instructions

Lihang Pan,<sup>1,2</sup> Guoqing Tu ,<sup>1,2</sup> Shubo Liu,<sup>3</sup> Zhaohui Cai ,<sup>3</sup> and Xingxing Xiong <sup>4</sup>

<sup>1</sup>Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, Wuhan 430 072, China

<sup>2</sup>School of Cyber Science and Engineering, Wuhan University, Wuhan 430 072, China

<sup>3</sup>School of Computer Science, Wuhan University, Wuhan 430 072, China

<sup>4</sup>School of Information Technology, Jiangxi University of Finance and Economics, Nanchang 330 013, China

Correspondence should be addressed to Guoqing Tu; [tugq@whu.edu.cn](mailto:tugq@whu.edu.cn)

Received 17 September 2021; Accepted 7 December 2021; Published 30 December 2021

Academic Editor: Mian Ahmad Jan

Copyright © 2021 Lihang Pan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

With the increasing popularity of the Internet of Things (IoT), the issue of its information security has drawn more and more attention. To overcome the resource constraint barrier for secure and reliable data transmission on the widely used IoT devices such as wireless sensor network (WSN) nodes, many researcher studies consider hardware acceleration of traditional cryptographic algorithms as one of the effective methods. Meanwhile, as one of the current research topics in the reduced instruction set computer (RISC), RISC-V provides a solid foundation for implementing domain-specific architecture (DSA). To this end, we propose an extended instruction scheme for the advanced encryption standard (AES) based on RISC-V custom instructions and present a coprocessor designed on the open-source core *Hummingbird E203*. The AES coprocessor uses direct memory access channels to achieve parallel data access and processing, which provides flexibility in memory space allocation and improves the efficiency of cryptographic components. Applications with embedded AES custom instructions running on an experimental prototype of the field-programmable gate array (FPGA) platform demonstrated a 25.3% to 37.9% improvement in running time over previous similar works when processing no less than 80 bytes of data. In addition, the application-specific integrated circuit (ASIC) experiments show that in most cases, the coprocessor only consumes up to 20% more power than the necessary AES operations.

## 1. Introduction

With the growing level of automation and intelligence in various industries, numerous IoT devices are getting more and more involved in production and life. The ensuing increase in data security risks is gaining ground. Architecturally, IoT systems can be divided into four layers: smart objects, edge computing, fog computing, and cloud computing [1]. The smart objects (e.g., WSN nodes) give up abundant resources such as computing power, memory, and energy by embracing low-cost deployments. Optimizing communication and security mechanisms for such constrained sensing platforms has been an influential trend in the field in recent years [2]. On the one hand, optimizations for resource-intensive algorithms are being introduced to improve the performance and security of constrained

devices. However, simplified algorithms, such as cryptography [3], may be limited in their scope of application as they do not conform to common standards. On the other hand, to improve the efficiency of the central processing unit (CPU) in solving specific problems, John Hennessy [4] introduced the concept of domain-specific architecture (DSA), which can be seen as a class of processors tailored to a specific group of applications. Likewise, the developers of the RISC-V instruction set architecture (ISA) have reserved space for the implementation of DSA, which is significant for lightweight platforms to break through the limitations. For example, in the information security field, DSAs based on the RISC-V standard will provide a new and flexible security option for IoT devices.

In terms of security, IoT systems can be divided into three layers: perception layer, transportation layer, and

application layer. The perception layer faces many security threats such as eavesdropping, malicious routing, and message tampering [5]. In general, cryptographic technology is still the primary solution to ensure data confidentiality, authenticity, integrity, and address other IoT security challenges [6]. According to the survey of [7], the percentage of cryptographic algorithms used in IoT papers has increased significantly after 2016. The study in [7] also showed that the most frequently used symmetric key algorithm is AES.

Block ciphers such as AES usually have several different modes of operation to cope with various scenarios. With an abundance of sensors connected, IoT devices can collect large amounts of data at high speed, whose value decays over time [8]. Additionally, some multimedia sensors can capture more sensitive data with greater relevance, such as image sensors. To avoid block ciphers exposing associations between data units, we have adopted the cipher block chaining (CBC) mode [9] for handling data in IoT scenarios. Furthermore, we use it for two other reasons:

- (i) It allows easy integration of cipher-based message authentication code (CMAC)
- (ii) Its decryption process can be parallelized on the server side

While CBC has the disadvantage of serial-only encryption, in a single-board environment with limited resources and low-performance requirements, parallel computing is not worth the cost. Similarly, [10] also suggested using CBC mode in WSN nodes to seek low-power operation, and [11] also chose CBC mode in lightweight encryption systems.

Overall, this article presents a lightweight RISC-V coprocessor that supports AES-CBC and AES-CMAC functions. The detailed contributions are summarized as follows:

- (1) We have formulated a RISC-V instruction extension scheme for AES at the application level. Receiving the starting address and length via instruction operands, the coprocessor can perform burst encryption (or authentication) of contiguous memory data. The intermediate results of this process are not visible to the CPU. Besides, the addressing range of the entire memory space and the user-specified workload size enhance the flexibility of memory allocation.
- (2) Using the direct memory access channel provided by *Hummingbird E203*, the proposed coprocessor can access the system memory in parallel while encrypting data, which reduces significantly the running times.
- (3) We have built an experimental prototype on an FPGA platform based on the SoC provided by *Hummingbird E203*. Furthermore, we run applications embedded with RISC-V custom instructions and conducted ASIC evaluation experiments to prove that the proposed coprocessor scheme can achieve high operating speed and low additional power consumption.

The remainder of this article is organized as follows: Section 2 introduces prior knowledge of the RISC-V ISA and AES; Section 3 presents current research on RISC-V coprocessors and instruction extensions and a survey of hardware implementation paths for AES; Section 4 presents the design method of the coprocessor; Section 5 shows the experimental results and evaluates; and Section 6 introduces the future work and concludes.

## 2. Preliminary

**2.1. RISC-V ISA.** RISC-V is a reduced instruction set created at the University of California, Berkeley (UCB) in 2010 [12]. It is not as redundant and complex as x86 nor requires high royalty fees like ARM. Compared to other open-source reduced instruction sets, such as OpenRISC and SPARC, RISC-V is highly simplified, refined, and modular. Developers have the flexibility to add or remove instruction set modules to implement custom processors based on different application scenarios. For example, Western Digital uses it for storage applications, NVidia uses it for graphic processors, and Xiaomi uses it in wearables [13], which means that RISC-V is highly scalable and will be of great use even in the IoT space.

The coprocessor interface is another actual proof of the high scalability of RISC-V. Unlike general-purpose processors, the purpose of a coprocessor or accelerator is not to execute all general-purpose instructions but to specialize in processing a specific type of instruction or custom instructions. A coprocessor design can optimize a particular algorithm as much as possible while maintaining a necessary range of generality. In particular, coprocessors excel at computation-intensive tasks, such as artificial intelligence [14] and information security.

The RISC-V instruction set reserves the coding space for custom instructions. As shown in Table 1 [12], there are four sets of predefined custom instruction operation codes, and all the custom instructions in this article are defined under the custom-0 opcodes.

**2.2. Hummingbird E203.** Various implementations of RISC-V processors are now appearing worldwide, many of which are open-source processor IPs. The design introduced in this article is based on the *Hummingbird E203*, an open-source RISC-V processor IP designed for low-power IoT devices.

The *Hummingbird E203* processor (hereafter referred to as E203) is a 32-bit RISC-V architecture IP developed by Nuclei Systems Technology for low-power, small-area scenarios and is open-sourced on GitHub1. As shown in Figure 1, E203 has a two-stage variable-length pipeline architecture, with the first stage completing instruction fetching and the second stage responsible for decoding, execution, and write-back. Memory access instructions and other multicycle instructions (including most custom instructions) are dispatched to different execution subunits.

Similar to the *Rocket Custom Coprocessor* (RoCC) interface [15], E203 also provides an interface called *Nuclei Instruction Co-unit Extension* (NICE) for coprocessor extension. As in

TABLE 1: RISC-V base opcode map,  $\text{inst}[1:0] = 11$ .

$\text{inst}[4:2]$ $\text{inst}[6:5]$	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	$\geq 80b$

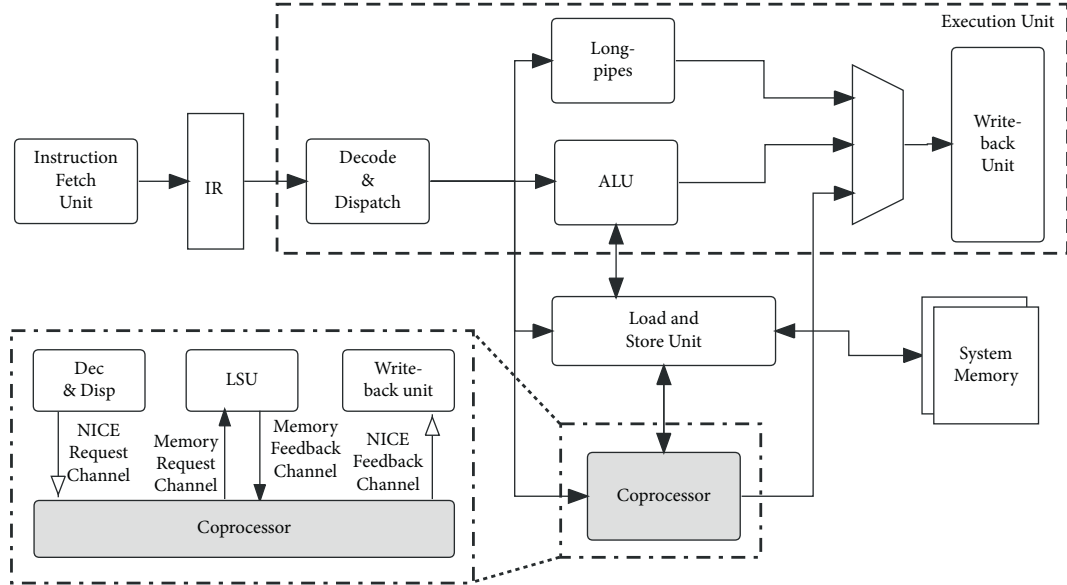


FIGURE 1: E203 pipeline stages and coprocessor location.

Figure 1, custom instructions will be dispatched from the execution unit (EXU) to the coprocessor through the NICE request channel. Then, the coprocessor writes the execution result back to EXU through the NICE feedback channel. Moreover, the separate memory access channels provide the coprocessor with direct access to system memory.

**2.3. AES Algorithm.** AES algorithm is a block cipher algorithm released by the National Institute of Standards and Technology (NIST) [16] in 2001 to replace the DES algorithm. There are three key lengths of AES, 128 bits, 192 bits, and 256 bits, with increasing complexity and security. However, the security of AES-128 is sufficient for low-power IoT, so this article only deals with the study about AES-128 (the following AES specifically refers to AES-128).

AES encryption is divided into two processes: the encryption process of the plaintext  $P$  and the expansion process of the key  $K$ , as shown in Figure 2. The plaintext  $P$  needs to go through ten rounds of calculations to obtain the ciphertext  $C$ . Except for the last round, each round (called round function) consists of 4 steps, including *SubBytes*, *ShiftRows*, *MixColumns*, and *Add-Roundkey*. The key expansion (or key schedule) also has ten rounds of operations, and each round generates a round key for adding to the plaintext in the *Add-Roundkey* step. The key expansion operation requires g-function processing on the last 4 bytes of the input (vector 3 in Figure 2), which includes 3 steps

similar to the round function components (shift, *SubBytes*, and constant vector addition, but no *MixColumns*).

The following will briefly introduce the specific principles of the above steps.

The 128-bit plaintext is organized into a 4-by-4-byte matrix, which is called the state matrix after bit-by-bit addition (XOR) with the seed key. The *SubByte* (or *S-Box*) operation nonlinearly substitutes each byte of the original state matrix into a new byte. It consists of the affine transformation and the modular multiplicative inverse calculation in the Galois field  $\text{GF}(2^8)$  and can usually be expressed as

$$Y = A \cdot X^{-1} + V, \quad (1)$$

where  $A$  is a constant matrix of 8 by 8 bits, and  $V$  is an 8-bit constant vector, collectively called the affine transformation. *ShiftRows* requires the row  $i$  of the input state matrix to move  $i$  bytes to the left. Next, supposing that  $s_{\cdot,j}$  is the column  $j$  of the state matrix output by *MixColumns*, and  $s'_{\cdot,j}$  is the column  $j$  of the input one, the *MixColumns* can be given by

$$s_{\cdot,j} = M \cdot s'_{\cdot,j} (0 \leq j \leq 3), \quad (2)$$

where  $M$  is a constant matrix of 4 by 4 bytes.

All the above mathematical operations are implemented in the  $\text{GF}(2^8)$ , among which the calculation of the modular multiplicative inverse is the most complicated, and the circuit optimization of *S-Box* and *MixColumns* is the focus of this article.

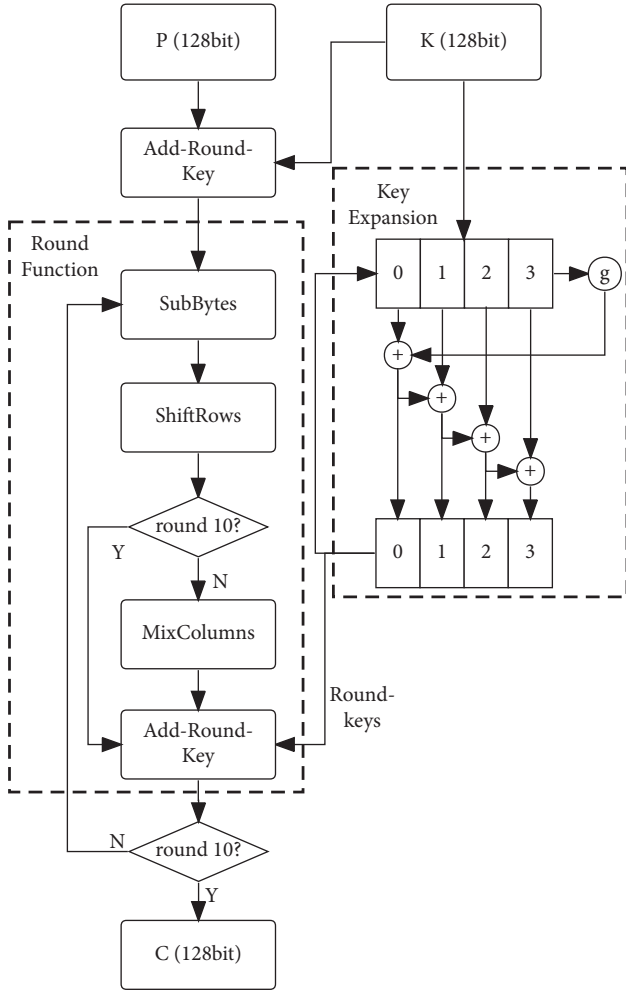


FIGURE 2: AES encryption process diagram.

Just like other block ciphers, AES also has several modes of operation. We chose CBC mode from the five modes (ECB, CBC, OFB, CFB, and CTR) recommended by NIST. An additional initial vector (IV) is required for encryption (see Figure 3) in CBC mode. The encryption of each plaintext block depends on the previous ciphertext block, which makes CBC more secure for pattern-free encryption [17], while the server side can still perform CBC decryption in a high-speed parallel.

The CMAC algorithm released by NIST in 2006 is a message authentication algorithm based on the CBC mode. More specifically, CMAC is based on the CBC-MAC message authentication code to overcome the security concerns of the latter in the case of variable-length messages [18]. CBC-MAC directly uses the last block in the CBC mode cipher as the authentication tag. Suppose a message  $M$  consisting of  $n$  blocks ( $M = \{m_1, m_2, \dots, m_n\}$ ) with a CBC-MAC authentication tag  $t = F_k(M)$ . At this point, after obtaining  $M$  and  $t$ , the attacker can still construct  $M' = \{M, m_1 \oplus t, m_2, \dots, m_n\}$  without knowing the key  $k$ , so that the CBC-MAC authentication tag is still equal to  $t$  ( $F_k(M') = t$ ), which is called an extension attack. CMAC differs in that two subkeys ( $k1$  and  $k2$  in Figure 4) are generated using the initial key, and the final message block is

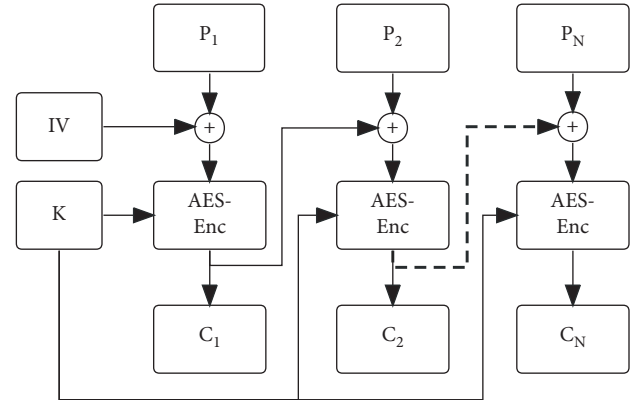


FIGURE 3: AES block encryption in CBC mode.

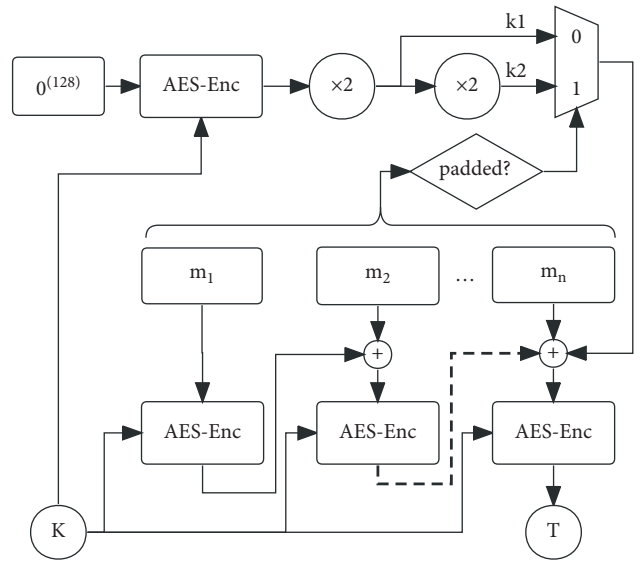


FIGURE 4: AES-CMAC algorithm diagram.

XORed with different subkeys depending on the message length, which avoids extension attacks in this case.

In this design, we have integrated support for CMAC based on AES-CBC.

### 3. Related Work

This section will discuss various AES coprocessor design schemes based on the RISC-V ISA and different approaches to AES hardware implementation. As the RISC-V has grown in popularity, more and more domain-specific coprocessor units have been implemented. Most of them [14, 19–21] are based on custom instructions defined by RISC-V. Not only that, as one of the most famous RISC-V coprocessor interfaces, the RoCC interface provides the coprocessor with access to the system memory (or cache), which is more efficient than general-purpose registers and also provides a larger application space for the coprocessor. The NICE interface derived from RoCC also inherits a similar memory access mechanism.

Ben Marshall [22] et al. explored and proposed six requirements that a standard AES extension scheme should

follow, including supporting all the parameter sets of AES, which has good universality for any design under the RISC-V architecture. However, the cost of being all-inclusive is the sacrifice of some efficiency and safety. The extensions they proposed are implemented separately with the various components of AES to reduce hardware complexity. Still, the CPU and system memory are highly involved in this, resulting in a significant loss of speed and risk on untrusted platforms. In other words, when faced with a white box environment, it will be as fragile as unwhitened software implementations.

The encryption engine proposed by Utsav Banerjee [23] includes an AES black box (or grey box) based on a RISC-V processor, which transmits data through a memory mapping interface. It can perform standalone computations through a dedicated 2-KB RAM and writes back the results via interrupts. However, this solution also has various limitations such as (i) excessive dependence on the processor micro-architecture, (ii) the need for additional memory space, and (iii) the fact that the functionality of the cryptographic engine is fixed in hardware and cannot flexibly respond to the diverse requests from the CPU. On the contrary, the coprocessor proposed is based on custom instructions and system memory access channels. It can implement the same function based on any RISC-V processor IP (such as Rocket) that opens a similar coprocessor interface.

Below we will discuss the existing state of the art on implementation details of the AES circuit. In the current research, there are two main implementation routes: one is to aim at small circuit area and low energy consumption, and the other is to pursue a high operating frequency with a slight circuit delay. No matter which implementation, it is inseparable from the structure design of round function and key expansion. The round function structure is a typical iterative structure, while there are two different structures in hardware implementation: unrolling structure and rolling structure [24]. The former can implement the pipeline mechanism, whereas the latter takes up much fewer resources. As for the key expansion, one of the implementation schemes is to compute all round keys at once and store them [25]. However, sacrificing such storage space is only conducive to long-lived keys. In case that the IoT devices cannot afford the extra storage overhead or use short-term keys, it is more sensible to generate the round keys in real time. Besides, most WSN nodes use dynamic session keys as one of the security measures, which makes the real-time key expansion more advantageous.

Research on the internal structure of the round function has focused on the number of *S-Box* units, and there are three mainstream implementation options, which differ primarily in terms of time and space trade-offs. There are 16 and 4 *S-Box* transformations in the round function and key expansion, respectively. The AES structure proposed in [26] is composed of 4 shared *S-Boxes*, which are used alternately by key expansion and round function for 1 and 4 clock cycles, respectively (see Figure 5(a)). To implement an ultimate small-area circuit, a team from Southeast University [27] proposed a single *S-Box* structure, which requires 20 cycles to complete a *SubBytes* step (see Figure 5(b)).

Despite this, we adopt the same efficient structure as in [23], as shown in Figure 5(c), using 20 *S-Boxes* to complete *SubBytes* within one cycle. Such a structure can spend only 11 clock cycles to complete the encryption of one plaintext block, making full use of the time required for the coprocessor to access the memory.

The *S-Box* transformation is the most complex step in AES and the most effective entry point to reduce the circuit area. Current *S-Box* implementation schemes mainly include look-up table method (LUT) and combinatorial logic implementations, which include composite field mapping (CFM) and DSE *S-Box* [28] techniques. In addition, some scholars have proposed a rotating binary decision diagram (TBDD) [29]. The LUT and TBDD schemes feature short critical paths. Still, they occupy considerable hardware resources. The DSE method is not suitable for low-area implementation despite the merit of the low flip rate. In this article, CFM is used to implement the *S-Box*.

CFM maps the input bytes to the composite field for inversion and then maps it back to the original field, avoiding the difficulty of computing the multiplicative inverse directly on the GF ( $2^8$ ). Although the critical path of the CFM implementation is relatively long, it is tolerable considering that the low-power processors of IoT devices will not run at very high frequencies ( $< 100$  MHz). Gaded and Deshpande [30] proposed a new composite field structure that optimizes the path delay to some extent, but it is not the optimal area solution.

## 4. AES Coprocessor

In this section, we describe the design of AES custom instructions and the architecture of the coprocessor. Unlike [22], our AES extended instructions can process tens of bytes of data at a time (there may be hundreds of bytes when processing multimedia data such as images) and will not expose the intermediate value in the encryption to the application. Driven by instructions, our coprocessor can continuously read or write memory from the starting address while performing AES encryption operations to shorten the running time.

*4.1. Custom Instructions Design.* The coprocessor instruction format defined by E203 is shown in Figure 6, where the *opcode* field is determined by the instruction opcode of *custom0-4* defined by RISC-V ISA. Each coprocessor instruction can encode two source register operands (*rs1*, *rs2*), a destination register operand (*rd*), and their enable bits (*xs1*, *xs2*, and *xd*). The *funct7* field allows re-encoding of coprocessor instructions.

Since the E203 is a 32-bit wide design, we cannot pass AES data directly in the form of register addressing or immediate addressing. So we chose the more efficient way of giving the memory address to the coprocessor via registers. The coprocessor can access the system memory directly through the memory access channel provided by E203. For the coprocessor to work continuously, the coprocessor needs the length of the plaintext and the starting address.

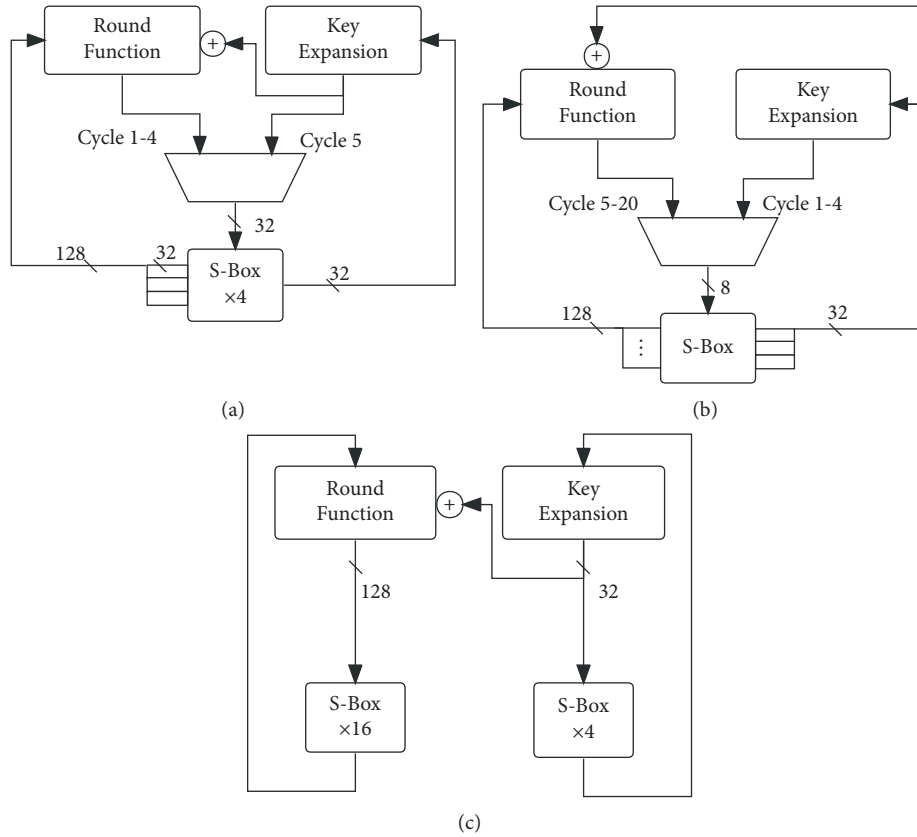


FIGURE 5: Three AES circuit structures with different *S-Box* numbers: (a) four *S-Boxes*, (b) single *S-Box*, and (c) twenty *S-Boxes*.

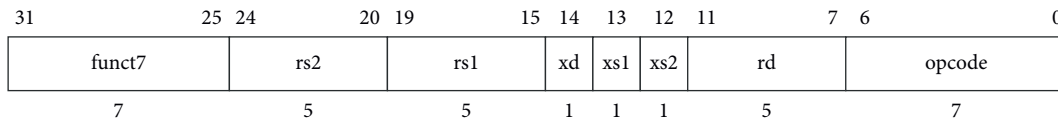


FIGURE 6: Coprocessor instruction encoding format.

Therefore, AES encryption requires at least two coprocessor instructions.

As shown in Table 2, we define three types of custom instructions, KEY-UPDATE for updating the key cache of the coprocessor and INIT and LOOP instruction pairs for completing encryption or authentication. The INIT instruction initializes the control unit of the coprocessor and passes the plaintext read address and the write-back address of the result. LOOP is used to start the coprocessor running and tell it the length of the workload.

**4.2. Coprocessors Architecture.** The AES coprocessor is mainly composed of four parts (see Figure 7): the decoding unit (DEC), the address generation unit (AGU), the load and store unit (LSU), and the AES computation core. When a custom instruction is dispatched to the coprocessor, the DEC and AGU receive instruction information from the NICE request channel (Figure 1). The former redecodes the 32-bit instruction, and the latter holds the values of the two source registers. The DEC outputs different control signals in different functional modes to drive the AES core to do its

corresponding work. The AGU maintains a counter to generate the addresses required for all memory accesses. At the same time, it ends the execution of LOOP instructions when appropriate. The LSU then accesses the memory based on the address information provided by the AGU. Due to the 32-bit width of the memory, accessing a 16-byte block of data takes at least four clock cycles (a total of 5 cycles if the response delay of the memory is included).

The AES core is the primary execution unit of the coprocessor and consists of the control unit (CU), the key expansion unit, and the round function unit. In addition, the AES core has a key cache, which, once the KEY-UPDATE instruction has been executed, provides the AES key for all subsequent encryption or authentication until the next KEY-UPDATE instruction arrives. The central processor does not need to store keys in risk memory but updates the key cache after each key agreement, which malicious applications cannot read again in any way.

Correspondingly, our AES core has a PLT cache, which matches the speed of accessing memory and AES computation. It takes five clock cycles for the coprocessor to access a 128-bit block of data and 11 clock cycles to encrypt or

TABLE 2: The proposed definition of coprocessor instructions.

	KEY-UPDATE	CBC-INIT	INIT	CMAC-INIT	Loop
funct7[2:0]	000	001		101	011
rs1	Read address for key	Read address for plaintext			Length of plaintext
rs2	(Disable)	Write address for result			(Disable)
rd	(Disable)	(Disable)			(Disable)

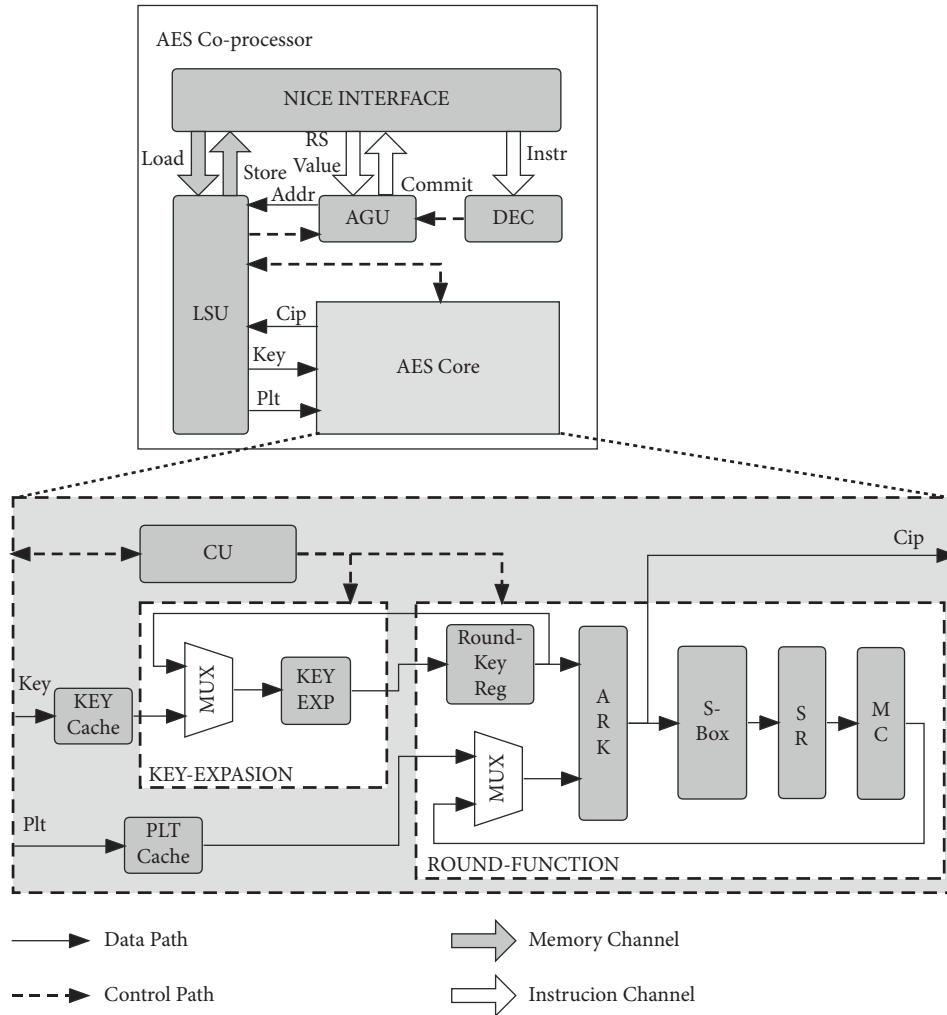


FIGURE 7: Structure of the proposed AES coprocessor.

authenticate a block of data. To maintain the uninterrupted operation of the AES core, we designed a finite-state machine to control memory access and AES calculations.

As shown in Figure 8, the coprocessor enters the *Load-key* state from *IDLE* when the KEY-UPDATE instruction arrives, and it returns to *IDLE* after instruction committing. Then, the coprocessor will enter different states according to the decoded information of the INIT instruction: prereading the first plaintext block (*Load-block-no.1, LB1*) in CBC mode or skipping *LB1* in CMAC mode. The latter is because the AES core first needs to encrypt an all-0 block to obtain the CMAC subkeys. Finally, the *Wait4Loop* state will remain after committing INIT.

The coprocessor will enter the read/write memory loop when the LOOP instruction arrives and resume *IDLE* when it is finished. If the LOOP instruction does not come after the first block of data has completed its AES computation, the coprocessor will advance to the *Store* state and return to *Wait4Loop* again when the write memory has finished, as shown in Figure 9.

The AES encryption process can be divided into three steps as shown in Figure 9, namely, PREPARATION, SET-UP, and LOOP-COMPUTATION. PREPARATION is the process of updating the key before encryption or authentication, and it is not required when the key is unchanged. SET-UP and LOOP-COMPUTATION are always executed

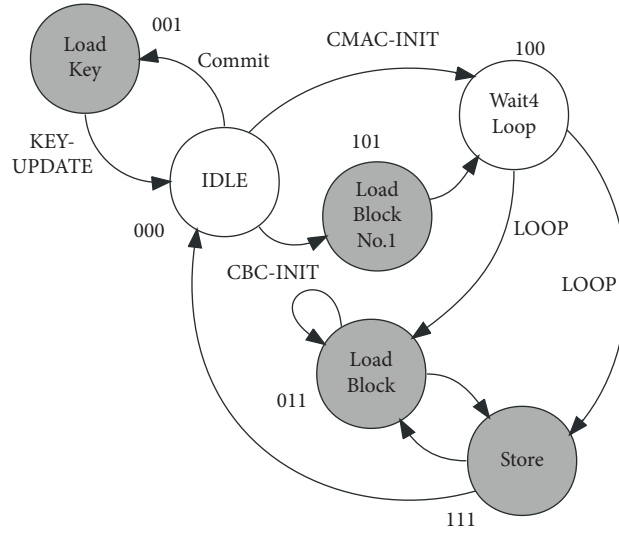


FIGURE 8: Finite-state machine for memory access control.

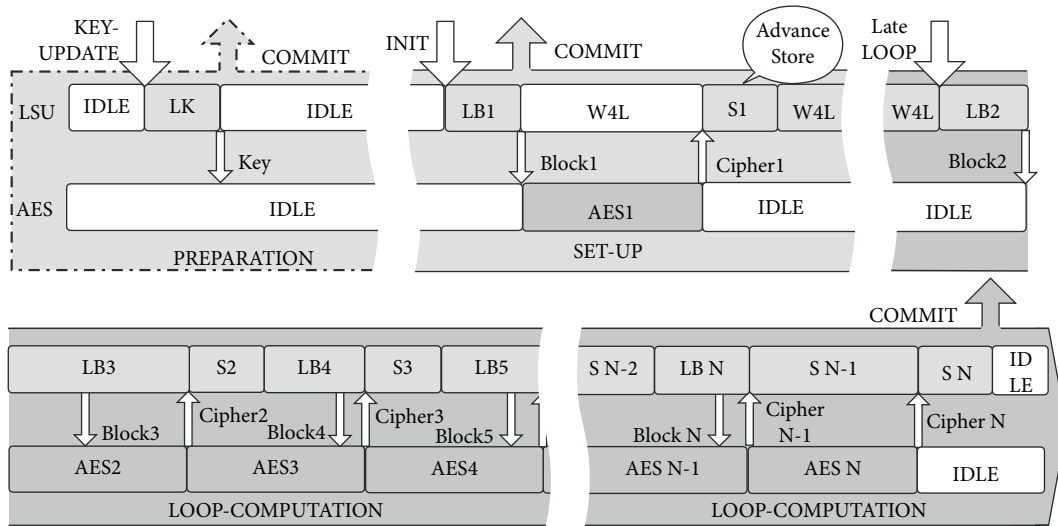


FIGURE 9: AES coprocessor instruction execution flowchart (CBC mode).

sequentially in pairs, and the timing of the arrival of the LOOP instruction determines the exact execution flow. In any case, however, the coprocessor reads the plaintext block in advance to maintain the continuous operation of the AES core.

The authentication process is similar to encryption, except that the coprocessor does not read the plaintext block in the SET-UP step, but instead generates two CMAC subkeys. Another difference is that only the last cipher block will be written back as an authentication tag.

**4.3. Hardware and Software Co-Design.** This part will describe the software and hardware codesign scheme when the coprocessor runs in CBC mode. This solution entrusts part

of the calculations to the CPU to maintain the simplicity of the coprocessor structure and the compatibility of different modes of operation.

The AES algorithm in CBC mode requires an additional initial vector (IV), which is only used for XOR with the first plaintext block. However, the involvement of the IV into the coprocessor will inevitably increase the complexity of the RTL design and instruction system to the detriment of timing convergence and application. Therefore, we can complete the XOR step in software (i.e., using general instructions) after obtaining the IV, thus avoiding the coprocessor from directly processing it. The RISC-V general instructions *Load* and *XOR* are executed in the LSU and ALU of E203, respectively. Together with the custom instructions, they complete the entire AES-CBC encryption.



Under the software and hardware codesign, we can easily program to achieve AES encryption and authentication functions. In Algorithm 1, we show how to use custom instructions to implement CBC encryption and CMAC authentication. The CBC encryption is outlined as follows:

- (i) After completing the key agreement with the server, update the coprocessor's key (line 17)
- (ii) In the CBC encryption function, the first thing to do is to preprocess the plaintext, that is, XOR with IV (line 3–5)
- (iii) After executing the INIT and LOOP instructions, if we need to save the plaintext, we can perform XOR again to restore it (lines 8–10)

Since IV is not required, we only need to execute custom instructions in the CMAC function.

## 5. Implementation and Evaluation

This section presents our approach to AES circuit area optimization and the validation and evaluation of the experimental prototype. Our experimental results demonstrate the speed advantage of the direct memory access-based coprocessor when processing larger data volume and the small additional power consumption it brings.

**5.1. Circuit Area Optimization.** The following will briefly introduce the area optimization method we implemented in this work, mainly embodied in *S-Box* and *MixColumns* parts.

The *S-Box* expression (1) based on composite field mapping can be rewritten as

$$Y = A \cdot \delta^{-1} (\delta X)^{-1} + V, \quad (3)$$

where  $\delta$  is the composite field mapping matrix, and  $\delta X$  is the mapping of  $X$  on the composite field  $\text{GF}((2^4)^2)$ . At this time, the multiplicative inverse  $(\delta X)^{-1}$  is also defined on the composite domain.  $\delta^{-1}$  is the inverse mapping matrix, which can map the multiplicative inverse back to the  $\text{GF}(2^8)$ . Here, the choice of composite field and the  $\delta$  determines the complexity of hardware implementation.

The *MixColumns* operation expression is in the form of equation (2). Fujii et al. [31] proposed a compact multiplexing form to reduce the circuit area, which is given as

$$\begin{cases} s_{0,j} = 2A_{01} + s'_{1,j} + A_{23}, \\ s_{1,j} = 2A_{02} + s_{0,j} + A_{01}, \\ s_{2,j} = 2A_{23} + s'_{3,j} + A_{01}, \\ s_{3,j} = 2A_{02} + s_{2,j} + A_{23}, \end{cases} \quad (0 \leq j \leq 3). \quad (4)$$

where  $A_{xy} = s'_{x,j} + s'_{y,j}$ ,  $0 \leq x, y \leq 3$ . This formula converts matrix multiplication into vector addition and reduces the circuit area from 108 XOR gates in the traditional method to 97 by multiplexing logic gates at the expense of lengthening the critical path.

**5.2. Experimental Results and Analysis.** Based on the E203 SoC, we have completed a prototype that can execute custom instructions on the Atrix-7 FPGA platform (Experimental Development Board from Nuclei DDR200T 2) to verify the functionality of the AES coprocessor IP.

The RISC-V compiler can compile custom instructions by embedding insn pseudo-instructions, which can be described as

$$\text{.insnopcode, func3, func7, rd, rs1, rs2,} \quad (5)$$

where the *func3* is a 3-bit field consisting of  $x1, x2, x d$  (see Figure 6).

Based on the hardware-software codesign, we wrote test programs to verify the functionality of the coprocessor in different modes. We built a RISC-V compiled environment under Linux to complete the system verification at an operating frequency of 60 MHz. Also, we measured the coprocessor operation speed in the program by reading the clock cycle counter, one of the control and status registers (CSR), defined in the RISC-V architecture.

The design takes direct memory access to get data from system memory, reducing the risk of exposing the encrypted intermediate results to untrusted platforms. Also, the parallel access design has a significant speed advantage in processing data structures stored continuously, such as arrays. In the literature [23], the AES core has the same processing speed of 11 cycles/block. Still, due to the use of memory mapping, the 32-bit CPU needs to spend at least eight additional clock cycles for memory reads and writes for every 16-byte workload, thus requiring at least 19 cycles per block on average. In contrast, our proposed coprocessor already spends less than 19 cycles/block when encrypting more than two plaintext blocks (32 bytes) and close to 11 cycles/block when encrypting more than 20 plaintext blocks (320 bytes) (see Figure 10). The time savings relative to the memory mapping method range from 25.3% to 37.9% for typical workloads of 5–20 blocks of data.

Below, we consider a specific situation. Suppose that there is a large amount of external data to be encrypted (data volume of tens to hundreds of KB). Due to the large data volume, overall encryption will be an unbearable memory burden for resource-constrained devices. One of the solutions here is to encrypt data by segments of different sizes depending on the memory margin. Moreover, the last 16 bytes of the ciphertext can be used as the IV in the next encryption to achieve continuous CBC mode. Since the coprocessor can share all memory space and handle workloads of different lengths, there is no need for memory copy except for IV protection (if full CBC encryption is required). In this scenario, the flexible allocation of memory space for encryption provided by the coprocessor will reduce the memory burden of the device. Not only that, such coprocessor IP is suitable for all RISC-V cores equipped with similar interfaces, such as Rocket core and E203.

We have performed segmentation encryption on the bitmap data of a 40 020-byte BMP image at a frequency of 60 MHz, and Table 3 shows the results. In the case of

**Input:** The address of all parameters, including the key, the plaintext (or message), the ciphertext (or tag) and the IV. Length of the plaintext (or message).

```

(1) Function AES_CBC_ENCRY(addrp, addrc, lenp, addriv):
(2)   if len ≥ 0 then
(3)     for i = 0 → 15 do
(4)       addrp[i] ← (addrp[i] ⊕ addriv[i])
(5)     end
(6)     CBC_INIT (addrp[i], addrc[i]) //The embedded custom instruction
(7)     LOOP (lenp) //The embedded custom instruction
      /* If the plaintext needs to be protected, then */
(8)     for i = 0 → 15 do
(9)       addrp[i] ← (addrp[i] ⊕ addriv[i])
(10)    end
(11)   end
(12) Function AES_CMAC_AUTHEN (addrm, addrt, lenm):
(13)   if len ≥ 0 then
(14)     CMAC_INIT (addrm, addrt) //The embedded custom instruction
(15)     LOOP (lenm) //The embedded custom instruction
(16)   end
      /* The encryption procedure is as follows */
      /* Here is key agreement */
(17) KEY_UPDATE (addrk) //The embedded custom instruction
      /* Here is IV synchronisation and plaintext padding */
(18) AES_CBC_ENCRY((addrp, addrc, lenp, addriv)
      /* Next is message authentication */
      /* Here is message padding */
(19) AES_CMAC_AUTHEN (addrm, addrt, lenm)

```

ALGORITHM 1: Software and hardware codesign.

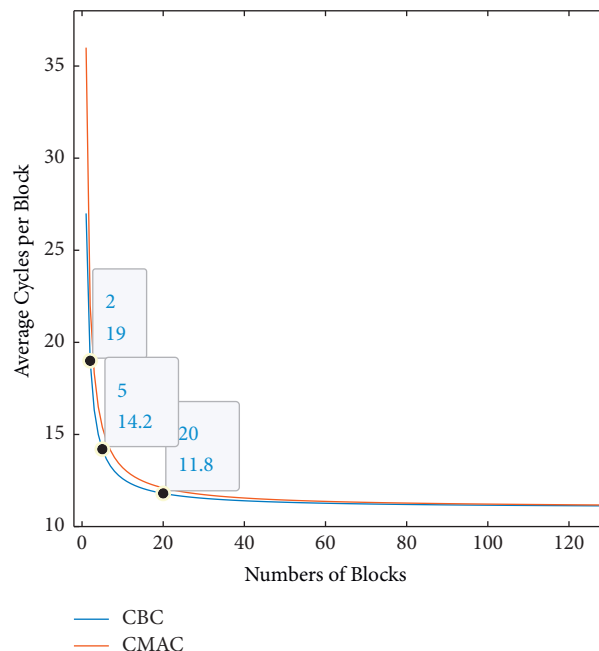



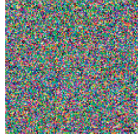
FIGURE 10: Changing trend of the average cycle as the workload increases.

relatively abundant memory space, the encryption throughput rate has reached around 600Mbps.

Furthermore, we also completed postsynthesis simulations using instructions in CBC mode as test vectors at 1.2 V

and 25 °C process conditions in TSCM-65-nm technology (typical case) and clock frequency up to 140 MHz. In addition, we evaluated the area and power consumption of the AES core and the entire AES coprocessor at a 100 MHz

TABLE 3: Onboard operation result of segmented image encryption.

Segment size/bytes	Original	Cipher	Total cycles <sup>a</sup>	Throughput (Mbps) <sup>a</sup>
256			40 736	471.56
512			34 267	560.59
1024			30 988	619.90
40 020 <sup>b</sup>			27 632	695.19

<sup>a</sup>Data excluding the cycle cost of communicating with peripherals. <sup>b</sup>No segmentation.

TABLE 4: Comparison of AES circuits with different implementation routes.

	[27]	[32]	[23]	AES core	Proposed Coprocessor
Technology	28 nm	65 nm	65 nm		65 nm
Voltage (V)	0.5	1.1	0.8		1.2
Frequency (MHz)	50	100	16		100
Cycles (block)	213	527	11	12 <sup>c</sup>	11 27
Area (mm <sup>2</sup> )	0.002 8	0.005 4	0.015	0.014 7	0.031 6(2.150 ×)
Power (mW)	0.045	0.246	—	6.07 <sup>c</sup>	7.19 <sup>c</sup> (1.184 ×)
Energy efficiency (pJ · bit <sup>-1</sup> ) (norm to 1.2 V)	1.5 <sup>a</sup> (4.31)	10.13 (12.05)	4.08 <sup>b</sup> (9.18)	5.69	—

<sup>a</sup>Data at TSMC-28-nm technology. <sup>b</sup>Measured energy. <sup>c</sup>Data obtained by encrypting 16 blocks of plaintext in CBC mode.

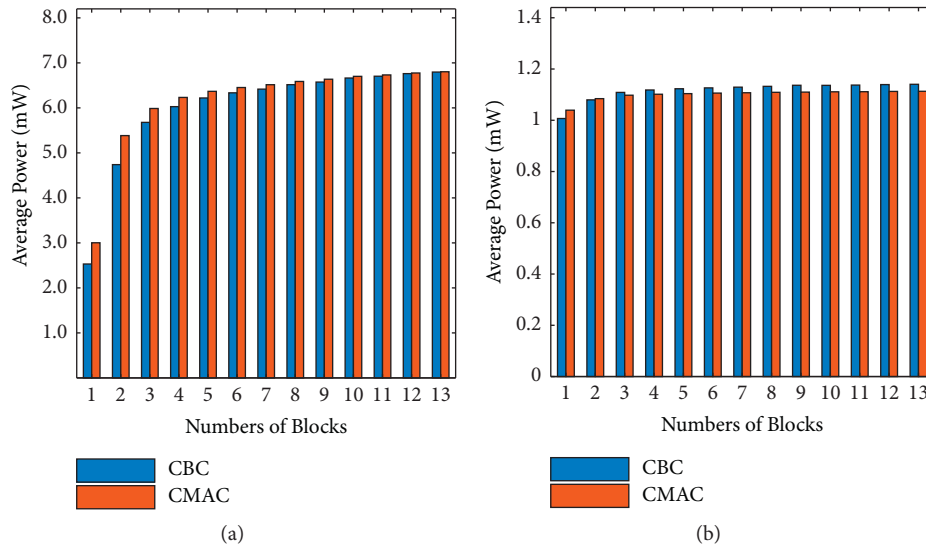


FIGURE 11: Average power consumption of AES core (a) and other modules (b) at different encryption workloads.

clock. Table 4 shows the comparison with the existing state of the art, from which we can see that when comparing the single AES core, the implementation that goes the small-area route [27, 32] has lower power consumption and significantly insufficient throughput. However, in terms of energy efficiency, the proposed design yields performance close to [27]. Compared with [23], which has the same processing speed, our proposal is almost the same in terms of area and better in energy efficiency. Moreover, the experiment also shows that the direct memory access-based coprocessor

proposed in this article does not lead to excessive additional power consumption, which is only 1.184 times that of the AES core when processing 16 blocks of data.

As shown in Figure 11, the power consumption of the coprocessor tends to stabilize when processing more than ten blocks of data. The power consumption of the modules other than the necessary AES core does not vary significantly with the size of the workload. Figure 12 shows that the additional power consumption caused by the coprocessor when encrypting more than 10 data blocks in CBC mode is

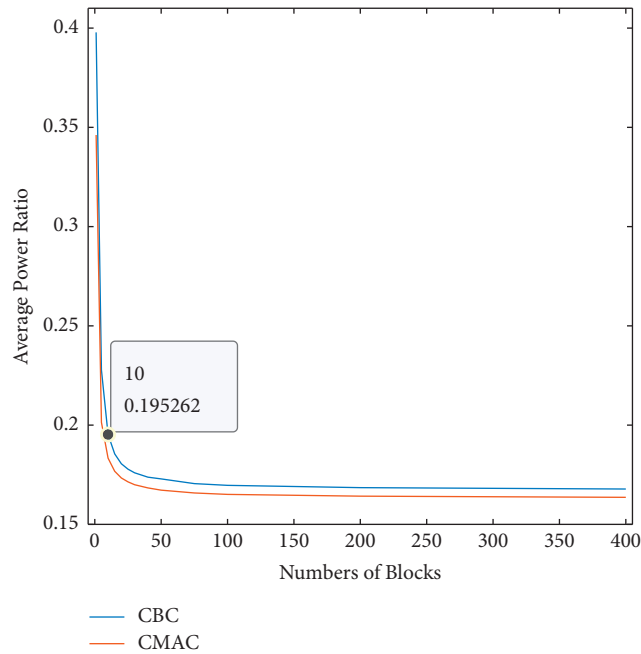


FIGURE 12: The power ratio of other parts to the AES core.

less than 20% of the power consumption of the AES core. Besides, this value is even lower in CMAC mode, where the coprocessor does not need to write to memory frequently.

## 6. Conclusion and Future Work

This article presents an implementation of an AES coprocessor that takes advantage of the direct memory access channels of RISC-V processors (including E203 and Rocket cores) to achieve independent access to memory data for high utilization of AES cores. We also designed a set of AES custom instructions within the RISC-V framework, with the specified data length as the processing unit, providing a developer-friendly programming interface. Our solution is more specialized and efficient than the existing RISC-V AES standard extension solutions.

Our experimental prototype on an FPGA platform shows that the coprocessor scheme proposed in this article has a speed advantage in large workload scenarios. Our ASIC postsynthesis simulation experiments show that the coprocessor extension scheme does not introduce significant additional power consumption.

In future work, we will add a true random number generator, asymmetric encryption, and hash operation unit to the existing AES coprocessor to achieve the goal of building a complete data transmission security architecture similar to [23]. The difference is that we are targeting an instruction-driven cryptographic coprocessor. It will implement functions including session key generation, certificate verification, session key encryption, and data encryption through custom RISC-V instructions. It does not need to obtain the symmetric key from the outside, nor does it provide any interface to access the original key, which will dramatically reduce the risk of session key leakage.

## Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant no. 41971407) and the Wuhan Frontier Project on Applied Foundations (Grant no. 2020020601012266).

## References

- [1] M. N. Khan, A. Rao, and S. Camtepe, "Lightweight cryptographic protocols for iot-constrained devices: a survey," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4132–4156, 2020.
- [2] J. Granjal, E. Monteiro, and J. Sa Silva, "Security for the internet of things: a survey of existing protocols and open research issues," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1294–1312, 2015.
- [3] A. Hafsa, A. Sghaier, J. Malek, and M. Machhout, "Image encryption method based on improved ECC and modified aes algorithm," *Multimedia Tools and Applications*, vol. 80, no. 13, pp. 19769–19801, 2021.
- [4] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.
- [5] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu, "Security of the internet of things: perspectives and challenges," *Wireless Networks*, vol. 20, no. 8, pp. 2481–2501, 2014.

- [6] A. B. Pawar and S. Ghumbre, "A survey on iot applications, security challenges and counter measures," in *Proceedings of the n2016 International Conference on Computing, Analytics and Security Trends (CAST)*, pp. 294–299, IEEE, Pune, India, December 2016.
- [7] S. K. Mousavi, A. Ghaffari, S. Besharat, and H. Afshari, "Security of internet of things based on cryptographic algorithms: a survey," *Wireless Networks*, vol. 27, no. 2, pp. 1515–1555, 2021.
- [8] S. Qi, Y. Lu, W. Wei, and X. Chen, "Efficient data access control with fine-grained data protection in cloud-assisted IIOT," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp. 2886–2899, 2020.
- [9] P. Panagiotou, N. Sklavos, E. Darra, and I. D. Zaharakis, "Cryptographic system for data applications, in the context of internet of things," *Microprocessors and Microsystems*, vol. 72, Article ID 102921, 2020.
- [10] F. Alsayid, H. Armoush, and K. A. Darabkh, "An experimental evaluation of the advanced encryption standard algorithm and its impact on wireless sensor energy consumption," in *Proceedings of the 2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*, pp. 1–6, IEEE, Sakheer, Bahrain, December 2020.
- [11] M. E. Hameed, M. M. Ibrahim, N. A. Manap, and A. A. Mohammed, "A lossless compression and encryption mechanism for remote monitoring of ECG data using Huffman coding and CBC-AES," *Future Generation Computer Systems*, vol. 111, pp. 829–840, 2020.
- [12] A. Waterman and K. Asanović, "The risc-v instruction set manual, volume i: user-level isa, document version 2.2," 2017, <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.
- [13] H. Legenvre, P. Kauttu, M. Bos, and R. Khawand, "Is open hardware worthwhile? Learning from thales' experience with RISC-V," *Research-Technology Management*, vol. 63, no. 4, pp. 44–53, 2020.
- [14] N. Wu, T. Jiang, L. Zhang, F. Zhou, and F. Ge, "A reconfigurable convolutional neural network-accelerated coprocessor based on risc-v instruction set," *Electronics*, vol. 9, no. 6, p. 1005, 2020.
- [15] K. Asanovic, R. Avizienis, J. Bachrach et al. "The rocket chip generator," Tech. Rep. UCB/EECS, EECS Department, University of California, Berkeley, CA, USA, 2016.
- [16] J. Daemen and V. Rijmen, "Reijndael: The advanced encryption standard," *Dr. Dobbs' Journal of Software Tools for the Professional Programmer*, vol. 26, no. 3, pp. 137–139, 2001.
- [17] C.-W. Huang, C.-L. Yen, C.-H. Chiang, K.-H. Chang, and C.-J. Chang, "The five modes aes applications in sounds and images," in *Proceedings of the 2010 Sixth International Conference on Information Assurance and Security*, pp. 28–31, IEEE, Atlanta, GA, USA, August 2010.
- [18] M. Bellare, J. Kilian, and P. Rogaway, "The security of the cipher block chaining message authentication code," *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 362–399, 2000.
- [19] A. De, A. Basu, S. Ghosh, and T. Jaeger, "Hardware assisted buffer protection mechanisms for embedded risc-v," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4453–4465, 2020.
- [20] A. De, A. Basu, S. Ghosh, and T. Jaeger, "Fixer: Flow integrity extensions for embedded risc-v," in *Proceedings of the 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 348–353, IEEE, Florence, Italy, March 2019.
- [21] X. Xue, C. Wang, W. Liu, H. Lv, M. Wang, and X. Zeng, "An RISC-V processor with area-efficient memristor-based in-memory computing for hash algorithm in blockchain applications," *Micromachines*, vol. 10, no. 8, p. 541, 2019.
- [22] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, "The design of scalar aes instruction set extensions for risc-v," *Cryptology ePrint Archive*, vol. 2021, 2021.
- [23] U. Banerjee, A. Wright, C. Juvekar, M. Waller, A. P. Arvind, and A. P. Chandrakasan, "An energy-efficient reconfigurable DTLS cryptographic engine for securing internet-of-things applications," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 8, pp. 2339–2352, 2019.
- [24] H. Qin, T. Sasao, and Y. Iguchi, "A design of AES encryption circuit with 128-bit keys using look-up table ring on FPGA," *IEICE - Transactions on Info and Systems*, vol. E89-D, no. 3, pp. 1139–1147, 2006.
- [25] T. Abdelmoghni, O. Z. Mohamed, B. Billel, M. Mohamed, and L. Sidahmed, "Implementation of aes coprocessor for wireless sensor networks," in *Proceedings of the 2018 International Conference on Applied Smart Systems (ICASS)*, pp. 1–5, IEEE, Medea, Algeria, November 2018.
- [26] M.-H. Dao, V.-P. Hoang, V.-L. Dao, and X.-T. Tran, "An energy efficient aes encryption core for hardware security implementation in iot systems," in *Proceedings of the 2018 International Conference on Advanced Technologies for Communications (ATC)*, pp. 301–304, IEEE, Ho Chi Minh City, Vietnam, October 2018.
- [27] M. Lu, A. Fan, J. Xu, and W. Shan, "A compact, lightweight and low-cost 8-bit datapath aes circuit for iot applications in 28 nm CMOS," in *Proceedings of the 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 1464–1469, IEEE, New York, NY, USA, August 2018.
- [28] G. Bertoni, M. Macchetti, L. Negri, and P. Fragneto, "Power-efficient asic synthesis of cryptographic Sboxes," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp. 277–281, New York, NY, USA, April 2004.
- [29] S. Morioka and A. Satoh, "A 10-Gbps full-aes crypto design with a twisted BDD s-box architecture," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 12, no. 7, pp. 686–691, 2004.
- [30] S. V. Gaded and A. Deshpande, "Composite field arithmetic based s-box for aes algorithm," in *Proceedings of the 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 1209–1213, IEEE, Coimbatore, India, June 2019.
- [31] H. Fujii, F. C. Rodrigues, and J. López, "Fast aes implementation using armv8 ASIMD without cryptography extension," in *Proceedings of the 2019 International Conference on Information Security and Cryptology*, pp. 84–101, Springer, Seoul, South Korea, February 2019.
- [32] K. Shahbazi and S.-B. Ko, "Area-efficient nano-aes implementation for internet-of-things devices," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 29, no. 1, pp. 136–148, 2020.