EasyChair Preprint

№ 15553

Sharing AES Engine for RISC-V Custom Instructions Performing Encryption and Decryption

Zahra Hojati, Zahra Jahanpeima, Maryam Rajabalipanah, Hossein Ta'Ati, Atefe Rabiei and Zain Navabi

December 11, 2024

Sharing AES Engine for RISC-V Custom Instructions Performing Encryption and Decryption

Zahra Hojati, Zahra Jahanpeima, Maryam Rajabalipanah, Hossein Ta'ati, Atefe Rabiei, Zainalabedin Navabi
School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran
{zahra.hojati, jahanpeima, m.rajabalipanah, hosseintaati89, atefe.rabiei, navabi }@ut.ac.ir

***Abstract*— In light of the critical role of cryptography in safeguarding data and the advantages of AES for hardware implementation, this paper presents a hardware implementation of the 128-bit Advanced Encryption Standard (AES) algorithm. We designed an AES unit that utilizes the same hardware for both encryption and decryption, which we integrated into a RISC-V-based processor, by developing four custom instructions to facilitate this integration. Our methodology leverages the inherent parallelism of the AES algorithm to optimize speed, enhance security features, and ensure low power consumption and efficient resource utilization. A comprehensive performance analysis showed a nearly 19x speedup compared to a software implementation, demonstrating significant performance and efficiency benefits.**

Keywords—AES Encryption, AES Decryption, Custom Instruction, RISC-V processor Security, FPGA Hardware Implementation

I. INTRODUCTION

With the proliferation of data communications and their various applications, there has been an increased demand for enhanced security systems and devices to safeguard individual information transmitted over communication channels. Embedded systems, wireless sensors, and wireless security networks are particularly vulnerable to security breaches and cyber-attacks. Addressing these security concerns is crucial for maintaining the integrity and confidentiality of the data handled by these systems.

RISC-V is a widely adopted instruction set architecture, particularly in embedded systems. The platform for this integration is a processor based on RISC-V, known as AFTAB, which was previously introduced in [1]. Our contribution involves the augmentation of AFTAB's architecture by incorporating four custom instructions and integrating our proposed AES unit into the system. A high-speed hardware implementation of AES on RISC-V that operates at low power and efficiently utilizes resources, by sharing hardware for encryption and decryption, is presented.

Integrating an AES engine within a system facilitates concurrency, enabling the processor to execute additional tasks simultaneously. However, this approach incurs overhead from data transactions between the CPU and the accelerator. Data dependencies among instructions can further degrade performance, as the CPU must wait for the accelerator to complete its operations before continuing dependent tasks.

Moreover, using a BUS for data transmission between the CPU and the AES accelerator presents significant security risks. Data transferred over the BUS can be intercepted, potentially compromising the security of the encrypted data. In contrast, integrating AES directly into the RISC-V processor as custom instructions can mitigate these issues. This integration allows the processor to perform encryption and decryption entirely within the CPU. Data can be accessed securely from the memory and the Register File internally, without external data transactions. This reduces overhead and enhances security by minimizing exposure to potential interception.

By eliminating the means to transfer data between the CPU and an external accelerator, power consumption can be significantly reduced. Data movement is a major contributor to power usage in modern systems.

The rest of this paper is organized as follows. Section II provides an overview of the RISC-V ISA and AES encryption/decryption algorithms. Section III introduces related works compared to this study. Section IV focuses on the methods of developing custom instructions, and the proposed AES hardware architecture. The implementation results and software comparisons are provided in Section V. This research's advancements and contributions are discussed in Section VI. Finally, the conclusion of this work is stated in Section VII.

II. PRELIMINARY REVIEW

This Section reviews the fundamental contents required for understanding the rest of the paper. An introduction to the RISC-V processor is presented and then the AES algorithm is described.

A) *RISC-V:*

RISC-V is an open-source instruction set architecture (ISA) developed at UC Berkeley that is freely available to academia and industry. RISC-V has a base integer ISA, to make RISC-V suitable for research and education, and optional standard extensions to support general-purpose software development. As shown in Fig. 1, The base instruction set has a fixed- length of 32-bit instructions [2].

B) *The AES Encryption and Decryption Algorithm:*

The AES encryption algorithm is an iterative, symmetric-key algorithm. Consequently, the decryption process requires only the ciphertext, following the same steps as encryption but in reverse order, with the same number of rounds, as illustrated in Fig. 2. In our case, a 128-bit key and message length with

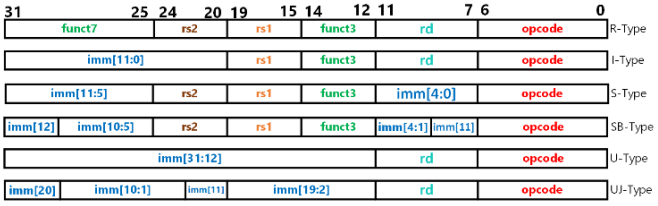

Fig. 1. RISC-V Instruction Format

10 rounds in CBC mode, provide sufficient security.

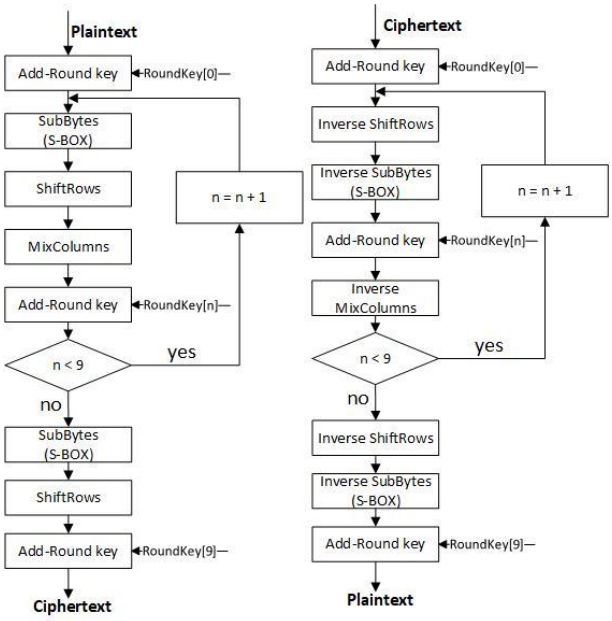

Fig. 2. Encryption and Decryption Flowchart

The AES encryption and decryption algorithms are shown in Fig. 2. Initially, there is a 128-bit plaintext message which is considered 16 bytes and represented in a 4×4 matrix format. All arithmetic computations within these steps are operated byte-wise. Additionally, the AES algorithm includes a key expansion part that generates a new round key for each round. In our design, the key expansion works simultaneously and synchronously with the AES core, categorizing our implementation as an iterative and rolling architecture.

III. Related Works

Implementing an AES coprocessor based on RISC-V introduces different approaches and challenges. RISC-V processors are being implemented globally, with many available as open-source processor IPs

In [3], the authors implement AES on RISC-V using Instruction Set Extensions (ISEs), recommending distinct ISEs for 32 and 64-bit architectures, which yield notable performance gains over software-only approaches. While effective in adding domain-specific functionality, this approach does not optimize for simultaneous encryption-decryption cycles within the processor or integrate AES-specific parallelism, which our work addresses with a unified custom instruction set.

In [4], the NLU-V instruction set extension is introduced for symmetric cryptography on 32-bit RISC-V, adapted from an 8-bit NLU architecture. This extension improves execution time and flash memory use but lacks flexibility for higher bit-width architectures and does not optimize resource usage for both encryption and decryption, a key focus in our RISC-V integration.

The Hummingbird E203 processor core [5] integrates 128-bit AES in CBC mode for IoT, enhancing efficiency and reducing power for cryptographic tasks. However, its design remains limited to specific IoT applications and CBC mode, whereas our work supports a more general-purpose AES integration within RISC-V, providing greater flexibility and broader cryptographic functionality.

[6] presents the design, implementation, and evaluation of Instruction Set Extensions (ISEs) for each of the 10 LWC final round submissions, aiming to enhance cryptographic functionality for resource-constrained devices. The paper emphasizes the importance of careful ISA design and quantified improvements, presenting ISEs as a hybrid approach between hardware and software for workload efficiency. However, while the evaluation of ISEs for LWC provides low-cost cryptographic solutions, it is limited by its generalized approach across algorithms, which may lack specific optimizations for high-performance cryptographic requirements like AES. Additionally, it does not explore custom encryption-decryption cycles or algorithm-specific parallelism, limiting its potential for achieving the high throughput needed in AES implementations.

In [7], the authors present an FPGA implementation of AES using an iterative approach to optimize resource use, exploring both rolling and unrolling architectures. The rolling architecture, which iteratively transforms data, is area-efficient but low in throughput. The unrolling approach achieves high throughput with pipelining but requires significant area. This work highlights area-throughput trade-offs; however, while the iterative approach conserves area, it sacrifices speed, and the unrolled structure demands large resources, limiting its use in area-constrained systems. The design does not address power efficiency for embedded or processor-integrated applications.

Ref. [8] proposes a non-pipelined AES algorithm aimed at balancing throughput and area with an online key generation method that synchronously produces round keys with encryption. This design allows for high-speed operation despite key changes but lacks the throughput advantages of pipelined architectures. While it supports dynamic key changes, it is less suited to applications needing concurrent encryption-decryption or minimal latency due to its sequential, non-pipelined nature.

In [9], a pipelined partial rolling architecture for AES on FPGA is proposed to achieve high throughput within a small memory area, with comparisons to other AES architectures. This design balances area and throughput but limits parallelization potential with its partially pipelined structure. Moreover, it does not support both encryption and decryption on the same hardware, reducing flexibility for compact implementations requiring dual functionality.

Ref. [10] demonstrates an AES-128 coprocessor for wireless sensor networks, optimized for throughput, resource use, and power consumption on FPGA. This coprocessor performs well in low-power, wireless environments but is primarily tailored to this application, lacking considerations for high-performance systems, dual encryption-decryption functionality, or integration as custom instructions in general-purpose processors.

Ref. [11] discusses RACE, a RISC-V SoC designed for secure, efficient homomorphic encryption (HE) en/decryption on edge devices, achieving energy and area efficiency through memory reuse and a shared Datapath. While RACE is highly optimized for HE, it is not designed for general-purpose cryptographic algorithms like AES or seamless processor integration. In contrast, our work provides dedicated AES operations within a RISC-V core, enabling a substantial speedup over software implementations via custom instruction integration.

IV. HARDWARE ARCHITECTURE

In this section, we present the overall architecture of the AFTAB processor and then we discuss our contribution in developing security custom instructions for AFTAB.

A) AFTAB:

AFTAB, a RISC-V processor introduced in [1], supports the RV32IM ISA, enabling 32-bit base integer and multiplication/division instructions. AFTAB is designed at the Register Transfer Level (RTL) and features a multi-cycle Datapath and Controller. Its Datapath includes several combinational and sequential units, such as an Arithmetic Logic Unit (ALU), Barrel Shifter Unit (BSU), and Add Sub-Unit (ASU) for instruction execution. Data transfer is facilitated by two units: the Data Adjustment Read Unit (DARU) and the Data Adjustment Write Unit (DAWU), which manage handshaking with external memory [1]. The processor also incorporates a Physical Memory Protection (PMP) unit to ensure secure memory access, raising exceptions for illegal access.

To enhance data security, we propose encrypting stored memory data and decrypting it upon loading. This is achieved through an extended instruction scheme for the AES algorithm, utilizing RISC-V custom instructions. The added AES hardware unit and the custom instructions format are detailed in the following sections.

B) AES Unit:

The AES algorithm is described in the previous sections. In this part, we focus on the RTL design of the AES encryption/decryption unit.

As shown in Fig. 3, the AES unit is formed from an AES core, the key expansion unit, input and output buffer units, and the main core controller. These units aim to convert plaintext to ciphertext or ciphertext to plaintext based on a mode input, which is extracted from the instruction and is synchronized together with appropriate handshaking signals, e.g., *start* and *done* signals. The input buffer unit collects 4 sets of 32-bit data, as the plaintext or message, from the memory and provides 128-bit data for the AES core.

The AES core, displayed in Fig. 3, receives a 128-bit message, key, and expanded key as its inputs and transfers a 128-bit *output text* as its output. This unit is composed of proper units compatible with the functionality of appurtenant to the steps explained in the encryption flowchart. They are named Add Round Key, S-box, Shift Rows, Mix Columns, and a local dedicated controller. The four former modules support both encryption and decryption based on the *mode* signal, in our design if the *mode* has the value of 0, the AES core encrypts the data, if the mode has the value 1, the AES core performs the decryption process. The local controller has the status of the current round. It terminates the encryption process after 10^{th} round and notifies the ciphertext is available.

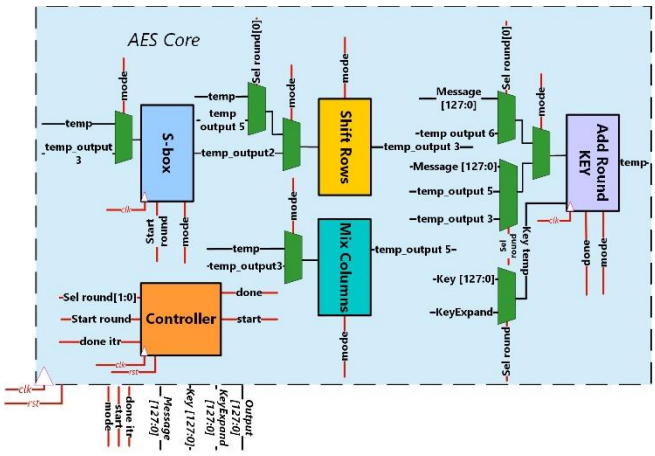

Fig. 3. AES Core Schematic

The optimization in this design consolidates the encryption and decryption processes into a unified unit. As illustrated in Fig. 2, the process begins with the essential Add Round Key step, where the input text and original key serve as inputs. The substitute input for this module varies according to the mode signal: for encryption, it receives output from the MixColumns module, while decryption uses the InverseSbox output. In the final encryption round, the Add Round Key step processes the output from ShiftRows, omitting MixColumns. A signal generated by the local controller manages these variations through multiplexers, adjusting based on the current round.

Upon encryption or decryption, the 128-bit output is segmented byte-wise. The output buffer stores this 128-bit ciphertext and distributes it in four-byte packs.

The key expansion unit generates round keys for each encryption round by applying substitution and permutation to the original key, divided into four 32-bit words. This unit comprises a controller, multiplexer, XOR gates, registers, and a cipher_word0_key module (Fig. 4). For the first word, cipher_word0_key is used, while subsequent words are generated by XORing the input key with the previous key segment, providing the input for each following round.

Fig. 5 illustrates the assembled AES unit. Data is loaded from the memory in 32 bits (Load word), so there is an input buffer to save 4 words from the memory and concatenate them to make the 128-bit input for the AES Core. Since the data is stored in the memory in 32 bits, an output buffer gets the 128-bit result from the AES Core and transfers it as a vector of 4 words, back to memory. All these are wired together, and their sequential behavior is realized through handshaking with the controller. Once the entire 128-bit output text is prepared, it is transferred back to the memory through 4 successive clocks, with each clock transferring 32 bits of data.

C) AFTAB custom instructions:

The complex encryption and decryption tasks of AES can be handled by a hardware unit specifically designed to be initiated and invoked by custom instructions added to the AFTAB processor. RISC-V ISA provides a series of reserved

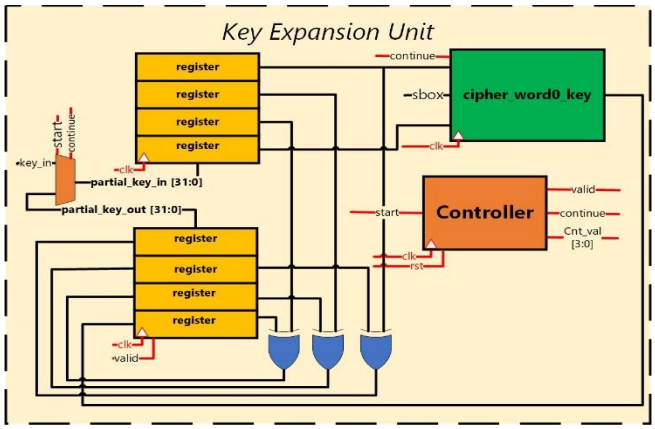

Fig. 4. Key Expansion Unit Schematic

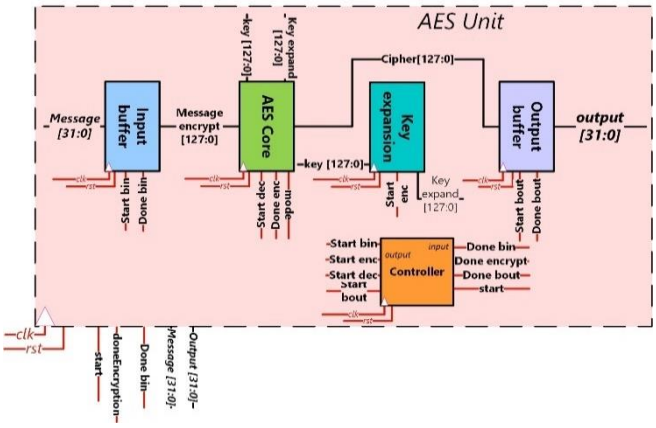

Fig 5. AES Unit Schematic

opcodes for future usage. We exert and customize four reserved instructions for running on the AFTAB processor. Hence, *load-AES* and *store-AES* instructions are customized for AFTAB, each with an additional bit called mode that defines the encryption or decryption process, in our case when the mode value is 0, the unit proceeds with encryption, otherwise it decrypts data.

The *load-AES* loads a set of four consecutive 32-bit data from the memory saves them into the input buffer of the AES core and then activates the AES core. As shown in Fig. 6, *load-AES* instruction's fields use bits 0 to 6 for the opcode and bits 8 to 12 for the source register, which encompasses the memory address. The 7th bit indicates that the memory is the destination for the data. The rest of the bits remain unused.

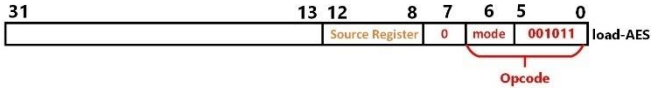

Fig. 6. *load-AES* Instruction Format

As depicted in Fig. 7, the *readReg-AES* instruction shares the same opcode as *load-AES* and transmits the data similarly. The key difference is that the data transfer occurs directly between the Register File and the AES core, as indicated by the 7th bit.

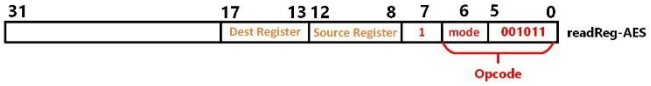

Fig. 7. *readReg-AES* Instruction Format

The *store-AES* stores four consecutive sets of 32-bit data from the output buffer to the memory. As shown in Fig. 8, *store-AES* instruction uses bits 0 to 6 for the opcode and bits 7 to 11 for the source register, which encompasses the memory address. The rest of the bits remain unused.

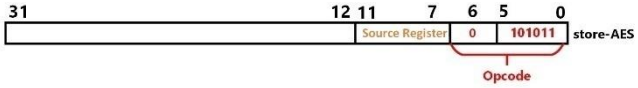

Fig. 8. *Store-AES* Instruction Format

As illustrated in Fig. 9, The *writeReg-AES* also shares the same opcode with *Store-AES* and only differs in the data transaction location. This opcode is intended to store the data in the Register File instead of memory, thereby reducing the overhead for certain tasks.

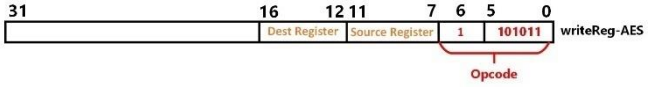

Fig. 9. *writeReg-AES* Instruction Format

D) Integration of AES Encryption Core into AFTAB:

This part attempts to elucidate the process of ASE core integration into the AFTAB processor. As shown in Fig. 10, the AES core is placed in AFTAB, and it has interfaces with DARU and DAWU for reading/writing from/to the memory. The AES core also has connections with the register files and the controller units.

Once AFTAB's controller detects the opcode of *load-AES* or *store-AES* instruction, it traverses to the corresponding state dedicated to these instructions. In *load-AES* execution states, the controller starts reading from memory through DARU and storing the data in the AES core's input buffer, and it also starts the AES core. In *store-AES* execution states, the controller starts writing the AES core's output buffer to the memory through DAWU.

V. Implementation and Result

This section explores both the software and hardware implementation of the AES encryption algorithm, providing a comprehensive comparison between the two approaches.

A) Verification Results:

To verify the encryption and decryption, the unit processed a 32×32-pixel image. The image data, transformed into 32×32 8-bit segments, is stored as eight packs of four 32-bit data, requiring 32 load instructions to fill the input buffer in 8 iterations for encryption. The encrypted data is then fed back to the unit for decryption. The results show that the decrypted image closely matches the original, confirming the accuracy of the AES core. As shown in Fig. 11, the decrypted image closely matches the original, demonstrating high precision and confirming the accuracy and reliability of the AES core integrated into the RISC-V processor.

Fig. 12 presents an example of an 8-bit 512×512 example for encryption results. In this example, the image was gray-

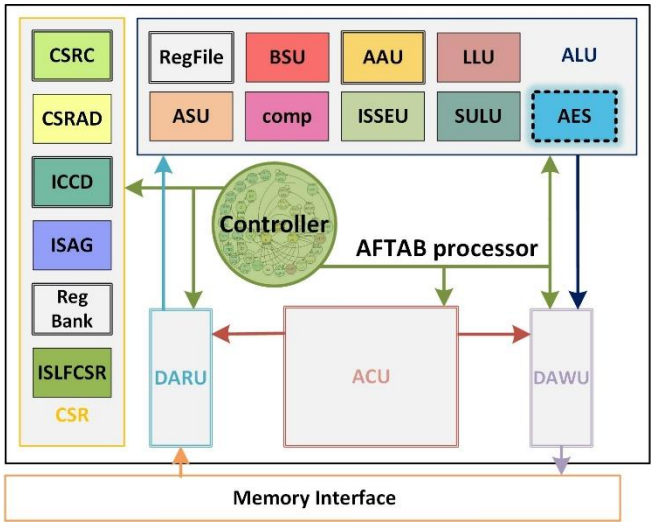

Fig. 10. AES in AFTAB

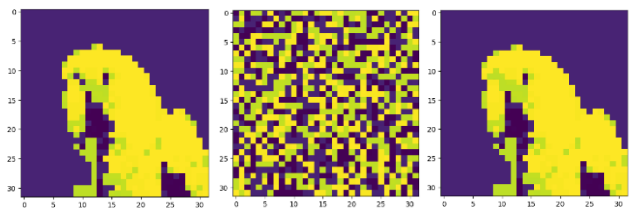

Fig. 11. Original Image, Encrypted Image, and Decrypted Image in order

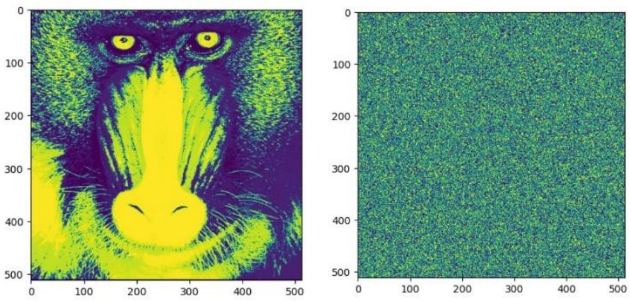

Fig. 12. Original image, Encrypted Image

scaled and fed to the unit in 128-bit packs. The encrypted output was later partitioned in its original form, and the right image was obtained, indicating a thorough encryption. Its decrypted image was identical to the original picture proving its high accuracy. Demonstrating the process across multiple image processing case studies and decrypting them showed that the AES Unit exhibits high precision with negligible error.

B) Simulation and FPGA Results:

The AES core, described in Verilog HDL, is integrated within the AFTAB processor. A 128-bit AES algorithm, including the two developed custom instructions, is simulated on AFTAB. The AES Core and AES unit were synthesized and implemented on a Xilinx FPGA "*7vx485tffg1157-1*". The results are presented in Table I. The number of slices reported for our proposed AES design includes both encryption and decryption, whereas other entries in the table reflect only encryption or decryption.

TABLE I. FPGA RESULTS COMPARISON

Implementation	**FPGA**		
Architecture	Number of Slices	$F_{max}(MHz)$	Latency (Clock Cycles)
Our proposed AES Unit	3751*	163.63	65
Our proposed AES Core	2271*	200.12	20
FPGA-based AES[3] (encryption)	10773	10.81	20
FPGA-based AES[3] (decryption)	15240	7.017	30
NLU-V-only [4]	5608*	-	19056
[12](encryption)	2444	456.00	-
[13](encryption)	15612	14.69	-

*This includes hardware for both encryption and decryption.

The AES core is synthesized and implemented on *"Cyclone IV E: EP4CE115F29I8L"*. The compilation is configured to map onto Logic Elements only, excluding the use of any embedded multipliers in the device. Table II compares our proposed non-pipelined architecture with other related works. The maximum frequency of the AES core is reported as 101MHz, with results available within 65 clock cycles. The number of logic elements reported for our proposed AES design reflects both encryption and decryption, while other entries in the table pertain solely to encryption.

During the fabrication of AFTAB, it is essential to ascertain the area occupied by the AES unit relative to the entire processor. Table III compares the ASIC implementation result of our proposed core to [11].

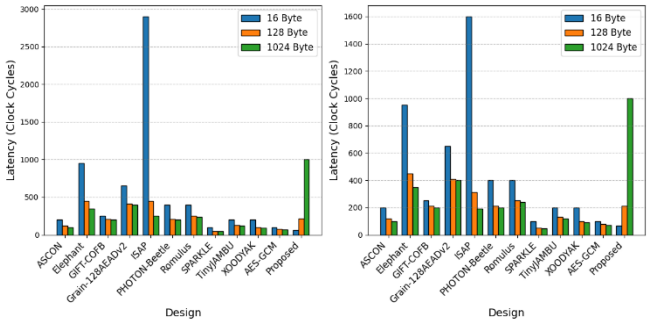

Fig 13. Latency Comparison with [6]

TABLE II. FPGA RESULTS COMPARISON

Implementation	**FPGA**		
Architecture	Logic Elements	$F_{max}(MHz)$	Latency (Clock Cycles)
Our proposed AES Unit	9326*	163.63	65
Our proposed AES Core	8063*	200.12	20
Aes 128 arch1[7]	75840	-	-
Aes 128 arch2[7]	80829	-	-
Aes 128 arch3[7]	75147	-	-
Harshali Zodpe [8]	4089	495.32	-
AES-4SM [9]	10530	49.8	21
AES-8SM [9]	10730	98.31	21
AES Coprocessor [10]	3047	-	-

*This includes hardware for both encryption and decryption.

TABLE III. ASIC RESULTS COMPARISON

Implementation	**ASIC**	
Architecture	Area (mm^2)	Power (mW)
Our proposed AES Unit	0.0296	1.31
(1024, 27) [11]	0.1569	0.35
(2048, 30) [11]	0.1638	0.58
(4096, 90) [11]	0.1756	1.47
(8192, 130) [11]	0.1996	3.78
(16384, 390) [11]	0.2509	11.51

Fig. 13 illustrates the latency of our design compared to the encryption/decryption units proposed in [6]. This comparison highlights the efficiency of our design in 16- and 128-byte transactions. Our design outperforms all others in 16-byte data transmissions and most in 128-byte transmissions. However, in 1024-byte data transmissions, our performance is slightly lower than some other designs due to the overhead associated with larger data blocks. Despite this, for small to moderately sized data transmissions, our design demonstrates superior performance, making it highly suitable

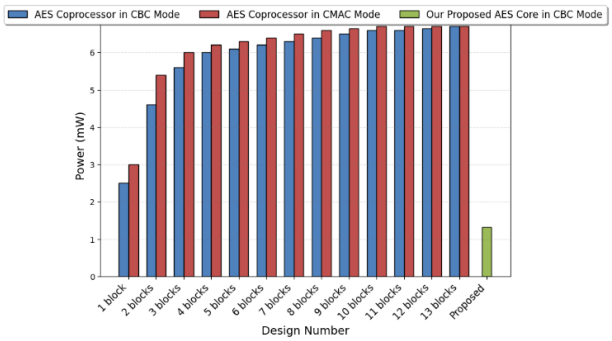

Fig 14. Power Comparison with [5]

for applications requiring efficient, small-scale data encryption.

Fig. 14 illustrates an average power comparison between our proposed AES core and the AES coprocessors from [5]. In this comparison, our AES core operating in CBC mode exhibits the lowest power consumption among all resources, in both CBC and CMAC modes.

C) Software-based Simulation:

To validate THE effectiveness of implementing an AES core, we conducted a test wherein the 128-bit AES algorithm was executed solely using the original RISC-V ISA on the processor. The AES program was compiled using the RISC-V GNU toolchain [13]. This simulation showed that the software-based implementation of AES requires 1219 clock cycles to complete each encryption and decryption process. In contrast, our hardware-based implementation completes the same task in 65 clock cycles. This significant difference in the number of clock cycles required for encryption between the software-based and hardware-based implementations is noteworthy. Our hardware implementation is approximately 18.75 times faster than the software approach This enhanced speed is crucial for time-sensitive operations and large-scale data encryption.

VI. Advancements and Contributions

A) Power Efficiency and Resource Utilization

Our implementation excels in power efficiency and resource utilization, particularly by merging encryption and decryption functions—a feature that sets it apart from state-of-the-art solutions. Leveraging the RISC-V architecture's characteristics and optimized implementation techniques, we achieve notable power reductions without compromising encryption strength, making it ideal for energy-constrained environments like IoT devices and battery-powered systems.

B) Speed Enhancement

One of the key contributions of our work is the substantial speed improvement compared to software-based implementations of AES. Our hardware implementation operates at a remarkable speed, achieving a performance increase of 19x compared to its software counterpart. This enhancement ensures rapid encryption and decryption, making our solution suitable for real-time applications where both speed and efficiency are critical. *C) Security Assurance*

In addition to its efficiency and speed, our implementation guarantees a high level of security. Through image processing testing, we have demonstrated high-precision encryption and correct decryption, ensuring the confidentiality of sensitive data. The presented figures in the results section affirm the robustness of our encryption scheme, highlighting its effectiveness in securing data against unauthorized access and malicious attacks.

VII. Conclusion

This paper presents an enhanced AES encryption implementation on the RISC-V processor, AFTAB, aimed at improving security for data transmission in safety-critical hardware platforms. By integrating an AES unit with four custom AES instructions, we achieved a nearly 19x speed increase in executing a 128-bit AES program compared to the standard implementation. Our design leverages an innovative approach that combines encryption and decryption using resource sharing and mode bit differentiation, optimizing logic element utilization during SoC fabrication. This work represents a significant advancement in cryptographic implementations on RISC-V, balancing power efficiency, resource utilization, speed, and security, making it well-suited for applications ranging from resource-constrained embedded systems to high-performance computing.

References

[1] Rajabalipanah, M., Roodsari, M. S., Jahanpeima, Z., Roascio, G., Prinetto, P., & Navabi, Z. (2021, September). AFTAB: A RISC-V Implementation with Configurable Gateways for Security. In 2021 IEEE East-West Design & Test Symposium (EWDTS) (pp. 1-6). IEEE.

[2] Waterman, A., Lee, Y., Patterson, D. A., & Asanovic, K. (2011). The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, *116*, 1-3.

[3] M arshall, B., Newell, G. R., Page, D., Saarinen, M. J. O., & Wolf, C. (2020). The design of scalar AES Instruction Set Extensions for RISC-V. *Cryptology ePrint Archive*.

[4] Uzuner, H., & Kavun, E. B. (2024). NLU-V: A Family of Instruction Set Extensions for Efficient Symmetric Cryptography on RISC-V. *Cryptography*, *8*(1), 9.

[5] Pan, L., Tu, G., Liu, S., Cai, Z., & Xiong, X. (2021). A lightweight AES coprocessor based on RISC-V custom instructions. *Security and Communication Networks*, 2021, 1-13.

[6] Cheng, H., Großschädl, J., Marshall, B., Page, D., & Pham, T. (2023). RISC-V instruction set extensions for lightweight symmetric cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 193-237.

[7] Arrag, S., Hamdoun, A., & Tragha, A. (2012). Design and Implementation A different Architectures of mixcolumn in FPGA. *arXiv preprint arXiv*:1209.3061

[8] Zodpe, H., & Sapkal, A. (2020). An efficient AES implementation using FPGA with enhanced security features. *Journal of King Saud University-Engineering Sciences*, 32(2), 115-122.

[9] Qin, H., Sasao, T., & Iguchi, Y. (2006). A design of AES encryption circuit with 128-bit keys using look-up table ring on FPGA. *IEICE transactions on information and systems*, 89(3), 1139-1147

[10] Abdelmoghni, T., Mohamed, O. Z., Billel, B., Mohamed, M., & Sidahmed, L. (2018, November). Implementation of AES coprocessor for wireless sensor networks. In *2018 International Conference on Applied Smart Systems (ICASS)* (pp. 1-5). IEEE.

[11] Azad, Z., Yang, G., Agrawal, R., Petrisko, D., Taylor, M., & Joshi, A. (2022, August). Race: Risc-v soc for en/decryption acceleration on the edge for homomorphic computation. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (pp. 1-6).

[12] Hussain, U., & Jamal, H. (2012, December). An efficient high throughput FPGA implementation of AES for multi-gigabit protocols. In *2012 10th International Conference on Frontiers of Information Technology* (pp. 215-218). IEEE.

[13] Wang, Y., & Ha, Y. (2013). FPGA-based 40.9-Gbits/s masked AES with area optimization for storage area network. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(1), 36-40.

[14] https://github.com/riscv-collab/riscv-gnu-toolch