Engineering Science and Technology, an International Journal 60 (2024) 101894

Contents lists available at ScienceDirect

Engineering Science and Technology, an International Journal

journal homepage: www.elsevier.com/locate/jestch

Full length article

AESware: Developing AES-enabled low-power multicore processors leveraging open RISC-V cores with a shared lightweight AES accelerator☆

Eunjin Choi [a], Jina Park [a], Kyuseung Han [b], Woojoo Lee [a,*]

[a] Department of Intelligent Semiconductor Engineering, Chung-Ang University, Seoul, 06974, Korea
[b] AI SoC Research Division, Electronics and Telecommunications Research Institute, Daejeon, 34129, Korea

ARTICLE INFO

Keywords:
Edge device
AES
RISC-V
Embedded processor
Multi-core processor architecture
FPGA prototyping

ABSTRACT

As open-source RISC-V cores continue to be released, the development of low-power multicore processors utilizing these cores is invigorating the edge/IoT device market. Nevertheless, comprehensive research on developing low-power multicore processors with integrated security features using existing open RISC-V cores remains limited. This study addresses this gap by introducing *AESware*, a dedicated lightweight hardware designed for energy-efficient AES (Advanced Encryption Standard) task execution, contributing to the development of AES-specific low-power RISC-V multicore processors. AESware supports variable key lengths and ensures minimal power consumption with a compact design. This standalone IP (Intellectual Property) is compatible with various open RISC-V cores, offering scalability and convenience. And importantly, we propose the most energy-efficient architecture for multicore processors equipped with AESware. Instead of assigning dedicated AESware to each core, we introduce a shared AESware architecture to maximize energy efficiency. We develop an operational algorithm for task scheduling in AESware, achieving maximum utilization and minimal latency while maintaining its lightweight nature. To evaluate our solution, we developed 24 processors into three groups: AESware-equipped, baseline, and those with an external AES accelerator per core. After FPGA (Field-Programmable Gate Array) prototyping for functional verification and power consumption analysis via 45 nm process technology synthesis, our findings revealed significant energy savings. AESware-equipped processors achieved up to 76%, 47%, and 33% energy savings at dual-, quad-, and octa-core configurations compared to baseline, respectively, and were more energy-efficient in running AES applications than those with individual accelerators.

1. Introduction

With the advancement of AI, research is actively conducted in both industry and academia to extract meaningful information from raw data in various fields such as healthcare, facial recognition, and location-based service provision. Consequently, intensive research and development are being carried out on edge devices, where user data is collected from input/output devices such as cameras, radar sensors, healthcare sensors, and microphones [1–4]. The research and development goals for these edge devices can be broadly classified into three primary objectives: (i) how to create an energy-efficient device, (ii) how to ensure the security of the data handled by the edge devices, and (iii) how to secure the price competitiveness of edge devices.

In response to the first question, research is underway to introduce various low-power technologies to edge devices. Since the duration for which an edge device, with its limited power source, can operate is a crucial factor determining product competitiveness, efforts are being made in areas such as the application of low-power hardware design technologies and development of power management algorithms [5–7]. For the second question, technology development for security assurance in data transmission between server clouds and edge devices is being conducted. Especially, the data handled by edge devices often include sensitive user information collected from various sensors and directly input by the user, making security crucial. Therefore, data must be encrypted during transmission. Edge devices are thus required to encrypt data when sending and decrypt upon receiving, supported by

☆ This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00345668), in part by the Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (MOTIE) (P0017011, HRD Program for Industrial Innovation), and in part by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2023-00277060, Development of open edge AI SoC hardware and software platform).

* Corresponding author.
E-mail address: space@cau.ac.kr (W. Lee).

https://doi.org/10.1016/j.jestch.2024.101894
Received 23 July 2024; Received in revised form 10 October 2024; Accepted 1 November 2024
Available online 14 November 2024

various technologies [8,9]. Finally, in the rapidly growing edge device market, the cost and performance of a product are directly tied to its competitive edge. Therefore, finding solutions that can deliver high-performing devices at a lower cost is paramount [10]. The development of processors for edge devices using open RISC-V cores is being spotlighted as the most promising solution to this issue. This is largely due to the significant cost savings potential by reducing the licensing fees associated with commercial processor cores. Furthermore, recent advancements have led to the development of multicore processors using open RISC-V cores to cater to the increasing complexity and diversity of edge device applications [6,11–13].

Despite fervent research and development efforts addressing individual questions, comprehensive studies on the overarching question – how to develop low-power multicore processors equipped with security features based on open RISC-V cores – have been lacking. For instance, the Advanced Encryption Standard (AES), an open-source encryption algorithm, is widely used in security processors, owing to its high security and speed [9,14,15]. In line with this, there have been developments in software that operates AES on RISC-V processors [16], the creation of RISC-V cores supporting AES [17], and research on designing AES accelerator to be embedded in RISC-V processors [18]. When AES processing is conducted solely through software-based methods, it faces clear limitations in terms of energy efficiency, and the use of AES-specific cores restricts the employment of a variety of open cores, confining their usage to that specific core only. Among available solutions, the use of AES accelerators in a software and hardware codesign approach seems to be the only viable option for developing a low-power processor equipped with security features based on open RISC-V cores. Among available solutions, the use of AES accelerators in a software and hardware codesign approach seems to be the only viable option for developing a low-power processor equipped with security features based on open RISC-V cores. Yet, even this has not been studied for low-power multicore platforms.

In addressing the question, we propose a solution that involves building AES-enabled low-power multicore processors with any available open RISC-V cores. To this end, we first design a dedicated lightweight hardware for energy-efficiently performing AES tasks in the RISC-V multicore processor, *AESware*. We design AESware to support three key lengths of 128 bits, 192 bits, or 256 bits that users variably specify and perform AES operations with minimum power consumption in a compact manner. By designing AESware as a separate independent IP located outside the core, we enable the application of AESware even if any of the currently available open RISC-V cores are used to develop a processor, thereby securing the scalability and convenience of the proposed solution.

Next, we propose the most energy-efficient architecture for AESware-equipped multicore processors. Although placing a dedicated AESware in each core in a multicore processor might seem like the most straightforward design and ensures maximum performance, it is an over-specification and not the optimal design in terms of energy efficiency for processors for edge devices. Instead, as illustrated in Fig. 1, we propose an architecture where the cores share a single AESware for executing AES tasks. And we integrate an arbitration logic into the AESware to ensure that the tasks requested by the cores are scheduled with best efficiency. For this scheduling, we develop an operational algorithm that performs efficient task placement to maximize the utilization of AESware and minimize latency, while maintaining the lightweightness of AESware through low complexity. When processing a task in the core software proves more energy-efficient than doing so in the AESware, the algorithm schedules the task to be executed by the core.

We have developed 24 processors to evaluate our proposed solution, conducting intensive experimental work with them. First, we have designed three groups of processors: one equipped with AESware, a baseline group without AESware, and another group with an external AES accelerator for each core to evaluate the performance of our proposed shared architecture. Each processor group consisted of single-, dual-, quad-, and octa-core multicore processors, and we have designed two different types of RISC-V-based processors for each. We have verified the functionality of these designed processors through FPGA prototyping, comparing their FPGA resource consumption. Subsequently, after synthesizing them using 45 nm process technology, we have compared their power consumption. Finally, we have conducted experimental work to assess the increase in energy efficiency offered by our solution. The results demonstrates that processors equipped with AESware achieve up to 76%, 47%, and 33% energy savings at dual-, quad-, and octa-core processor compared to the baseline processor, respectively, and show more energy-efficient in running general-purpose AES applications than the processors with individual AES accelerators.

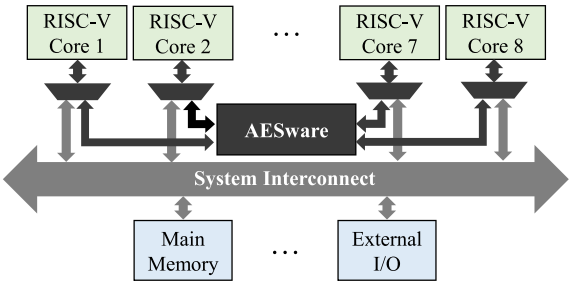

Fig. 1. Processor architecture equipped with a shared AESware.

2. AES algorithm: The background

The AES is a widely adopted symmetric key block cipher algorithm designed for secure data encryption and decryption. Standardized by the National Institute of Standards and Technology (NIST) in 2001 [19], AES uses the same key for both encryption and decryption, classifying it as a symmetric encryption algorithm. The key can be 128, 192, or 256 bits in size, and data is processed in blocks of 128 bits, each divided into four 32-bit words.

AES operates by generating additional keys, known as *round keys*, through a key expansion process that extends the initial key to produce multiple subkeys used throughout the encryption and decryption rounds. The number of generated round keys depends on the size of the initial key: 128-bit keys generate 44 words (10 encryption rounds + 1 initial round key), 192-bit keys generate 52 words (12 rounds + 1 initial round key), and 256-bit keys generate 60 words (14 rounds + 1 initial round key). Initially, the input key is used directly as the first round key, and subsequent round keys are generated using three core operations: Rotate, Subbytes, and Round constant (R_{con}).

- **Rotate**: The last word of the previous block is cyclically rotated to the left, moving the first byte to the last position while shifting the remaining bytes to the left.
- **Subbytes**: The rotated word is substituted byte-by-byte using the Substitution-box (S_{box}).
- **Round Constant**: A unique constant (R_{con}) is XORed with the current word to ensure that each round produces a unique key.

In the key expansion process, Rotate and Subbytes are applied to the last word of the previous round key, and R_{con} is XORed with only the first byte of the resulting word. For 256-bit keys, however, only the Subbytes operation is repeated every eighth word.

A critical component of AES is S_{box}, which introduces nonlinearity to enhance the security of the encryption. S_{box} is a 16×16 matrix with 256 values, each represented by 1 byte. These values are generated using Affine Transformation and the Multiplicative Inverse in Galois Fields, making the S_{box} highly resistant to linear and differential cryptanalysis. Because of its robust cryptographic properties, the same S_{box} is used in other encryption standards such as SM4 [20] and Serpent.

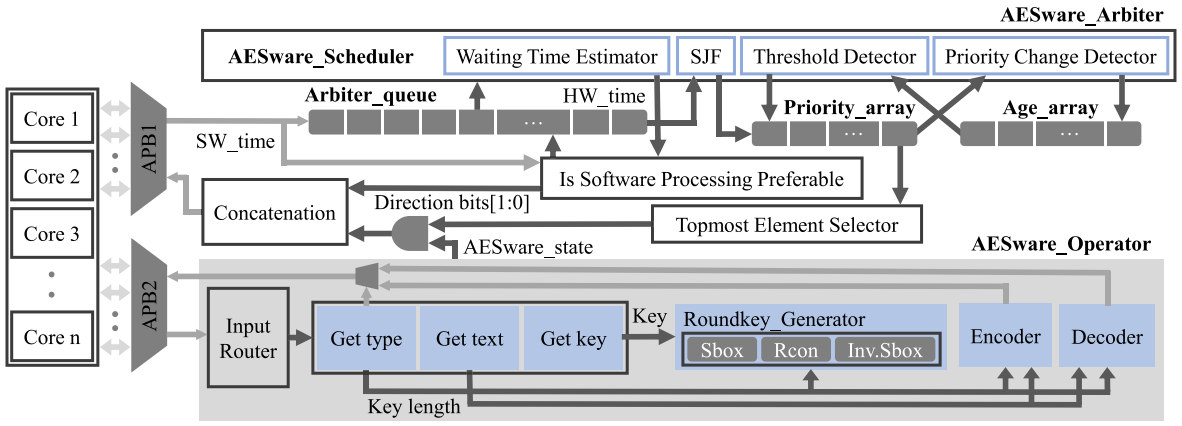

Fig. 2. Block diagram of the proposed AESware.

The AES encryption and decryption process applies multiple rounds of transformations to the data block, depending on the key size: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each round (except for the final round) consists of four operations: AddRoundKey, Subbytes, Shiftrows, and MixColumns. The final round omits MixColumns, performing only Subbytes, Shiftrows, and AddRoundKey to produce the encrypted output. In more detail, the operations in each round are as follows:

- **AddRoundkey** : The current data block is XORed with the corresponding round key.
- **Subbytes** : Each byte of the data block is substituted using the S_{box}. During decryption, the inverse S_{box} is used to reverse this substitution (InvSubbytes).
- **Shiftrows** : The rows of the data block are cyclically shifted to the left by increasing amounts: the second row is shifted by 1 byte, the third row by 2 bytes, and the fourth row by 3 bytes. During decryption, the rows are shifted back to their original positions using the inverse operation (Inv.Shiftrows).
- **Mixcolumns** : Each column of the data block is transformed using matrix multiplication in a Galois Field, which provides diffusion by mixing the bytes in each column. During decryption, the inverse transformation (Inv.MixColumns) is applied to reverse this process.

This iterative structure of transformations is key to AES's security, as it ensures that even a minor change in the input data results in a drastically different output. Furthermore, this design makes AES resistant to various forms of cryptanalysis, providing robust protection against attacks.

3. Aesware: The proposed solution

Despite the high security and utility of the AES algorithm, its implementation through purely software-based methods on energy-hungry embedded processors, such as those used in edge devices, results in significant inefficiencies in terms of energy consumption. Additionally, while the open RISC-V cores have garnered substantial interest from both academia and industry, a limitation arises in developing AES-enabled embedded processors using these cores. Only a few RISC-V cores support AES functionality, leaving the majority of open RISC-V cores unsuitable for developing such secure processors.

To address these limitations, we propose an AESware hardware accelerator and a shared architecture solution that allows any multicore processor built with open RISC-V cores to be AES-enabled and energy efficient. This solution enables each core in the multicore system to offload AES processing tasks to the AESware hardware, which performs the operations more efficiently than software-based methods running on individual cores. Since multiple cores share the AESware, a scheduling mechanism is required to manage cases where several cores simultaneously request AES operations. When one core is already utilizing the AESware, other cores must wait, requiring an efficient scheduling strategy. This scheduling includes an algorithm that evaluates whether it is more energy-efficient for a core to wait for AESware availability or to process the AES task within the core itself in software. If waiting is not energy-efficient due to prolonged delays, the AESware directs the core to handle the AES task in software. Our proposed solution includes this scheduling algorithm, ensuring that AES tasks are managed with optimal energy efficiency across the shared multicore system.

Fig. 2 provides an overview of the structure of the proposed AESware. AESware primarily consists of the *AESware_Arbiter*, responsible for scheduling the AESware utilization of the multiple cores, and the *AES_Operator* that handles actual AES operations. As shown in the figure, both the *AESware_Arbiter* and *AES_Operator* have their dedicated APB ports, *APB1* and *APB2*, respectively. These ports facilitate communication between the cores and the AESware components. When AESware receives an AES operation request from a core, it schedules the order of the AES operation for the respective core via the *AESware_Arbiter*. If the *is_Software_Processing_Preferable* module decides on software processing, considering *SW_time* and *HW_time* (the estimated waiting time calculated by the *Waiting_Time_ Estimator* module in the figure), an output including direction bits and the core tag number is sent to the core. This instructs the core, which has had its tag number outputted from AESware, to handle the AES operation either directly in software or by waiting for AESware as directed by the output. When AESware is ready to begin an operation, a signal is sent through APB1 to indicate this.

Once a core receives AESware readiness signal, it communicates with *AES_Operator* within AESware through APB2 and performs the AES operation. At this point, if the core requires encryption, it inputs plaintext and a key; conversely, if it desires output, it inputs cipher text and a key. Depending on the operation requested, *AES_Operator* uses its internal *Encoder* or *Decoder* module to calculate the cipher text or plain text. This output from AESware is then transferred to the core via APB2. Since there are three types of AES – 128 bit, 192 bit, and 256 bit – comprised of 4, 6, and 8 words (32 bits) respectively, APB2 presents ready signals for 4, 6, and 8 times in each communication, matching the current AES type. Meanwhile, *AESware_Arbiter* monitors the state of the *AES_Operator* and once the final output is successfully delivered and the operator enters an IDLE state, it enables the request from the next core in the *Arbiter_queue*.

In the following subsections, we provide a detailed description of the architecture of AESware, along with the design, functionality, and operational mechanisms of its various modules.

Algorithm 1 Key expansion

Input: Key (128 or 192 or 256 bits), S_{box}[7:0][256-1:0], Rcon[7:0][255-1:0], N_b, N_k, N_r
Output: $Roundkey$ consisting of $N_r + 1$ sets of N_b words
Initialization: 1^{st} round in $Rounkey = Key$
while $2 \le i \le N_b(N_r+1)$ **do**
  temp = $(i-1)^{th}$ round in $Roundkey$
  if $i\%N_k = 0$ **then**
    Do_Rotate %The last word is rotated, moving the first to the end.
    Do_Subbytes %The rotated word is replaced using S_{box}.
    Do_Round_Constant %XOR Rcon with the current word.
  else if $N_k = 8$ *and* $i\%N_k = 4$ **then**
    Do_Subbytes
  i^{th} round = $(i-1)^{th}$ round $\otimes$ temp
  $i = i + 1$

3.1. AES_Operator : Handling AES operations

The *AES_Operator* performs AES operations in three primary stages, as illustrated in Fig. 2. Initially, it receives data from the core and conducts preprocessing. This includes determining whether the core's requested operation is encoding or decoding via *Get type* and verifying the bit-length of the key. This key length is parameterized and becomes an input for subsequent modules: *Encoder*, *Decoder*, and *Roundkey_Generator*. In the *Get text* stage, plain text is input for encoding, while cipher text is input for decoding. Following this, the *Get key* step involves receiving the encryption key. The *Input Router* is responsible for identifying whether the input received from the core corresponds to *Get text*, *Get type*, or *Get key*. The received key must then be transformed into an encrypted key, the round key. This transformation process is executed in the *Roundkey_Generator*. Once the key expansion is completed and the round key is generated, an enable signal is dispatched to the *Encoder/Decoder*, signifying that the encoding or decoding process can begin. Finally, the outcomes of the *Encoder/Decoder*'s operation are transmitted back to the core via APB2

We designed each individual block of the *AES_Operator* as follows.

3.1.1. Roundkey_Generator

AES operations employ a symmetric key algorithm that uses the same key for both encryption and decryption processes. Therefore, we designed a single *Roundkey_Generator* module to be shared with both encryption and decryption.

The output of the *Roundkey_Generator* is generated as round key through the key expansion process, as shown in Algorithm 1. This process takes S_{box}, R_{con}, and the number of rows and columns of the state matrix N_b. At the time AES was adopted as a standard, it was designed to use a fixed 128-bit state matrix regardless of the key size. The 128-bit block size can be represented as a 4 × 4 matrix, with each element consisting of 1 byte. Consequently, the parameter N_b, representing the number of columns in the state matrix, is conventionally set to 4 and the number of words in a key N_k and N_r (N_k plus 6) as inputs. This process is well-established in the literature, so we will not delve into detailed explanations here, but rather focus on the unique pattern observed during our design of the *Roundkey_Generator*.

This pattern refers to the preceding operations that the key expansion performs in every round before executing the XOR operation with the previous round. If the current round is a multiple of N_k, the *Roundkey_Generator* precedes with *Rotate*, *Subbytes*, and *Round Constant* operations, performed by the Do_Rotate, Do_Subbytes, and Do_Round_Constant functions, respectively. Notably, when N_k is 8, the Subbytes operation is repeated every fourth round. During the design of the *Roundkey_Generator*, we determined that constantly calculating whether the current round is a multiple of N_k or a multiple of 4 (due to N_k being 8) and deciding which of the four processes to proceed with would introduce significant overhead.

To address this, we developed three $N_{k\_advisor}$ signals for each key type and a Finite State Machine (FSM) that operates using these signals. Each $N_{k\_advisor}$ consists of 20 bits. Each bit is predefined to determine which steps among Rotate, Subbytes, and Round Constant will be executed in that round. Every time a new round is entered, the *Roundkey_Generator* checks the Least Significant Bit (LSB) of $N_{k\_advisor}$ and carries out the operation it directs. After each round is created, the $N_{k\_advisor}$ shifts to the right by 1 bit, updating the LSB.

3.1.2. Encoder

Fig. 3 illustrates the procedure of the AES operation, as it is executed via the FSM of *Encoder* and *Decoder*. The entire operation commences once the top module's *enable_signal* is activated while in the FSM's standby state, referred to as IDLE. In the encoding section, *Get Plain text*, and in the decoding section, *Get Cipher text*, represent the stages where text input from *Get text* is transferred into a 4 × 4 state matrix. Following that, for encoding, four stages, namely AddRoundkey, Subbytes, Shiftrows, and Mixcolumns, are iteratively executed to transform the plain text in the state matrix into cipher text. A bit referred to as *guide* indicates the repetition duration for these stages. This *guide* is governed by a variable named *Count*. These steps are reiterated until every one of the N_r rounds are integrated at the AddRoundKey stage. Therefore, the design increments *Count* by one every time the AddRoundkey step is carried out. The steps Subbytes, Shiftrows, Mixcolums, and AddRoundkey are continually performed to enhance the complexity of the state matrix, as long as *Count* is less than N_r, or in other words, when *guide* equals 1. Once *Count* reaches N_r and *guide* turns 0, after completing Subbytes, Shiftrows, and AddRoundkey (excluding Mixcolumns) just once, the encryption process is considered complete.

3.1.3. Decoder

In the IDLE state, upon recognizing the *enable_signal*, AESware accepts the text to be decrypted, places it into the state matrix during the *Get Cipher text* phase as shown in Fig. 3, and proceeds through four stages: AddRoundkey, Inverse (Inv.) Shiftrows, InvSubbytes, and InvMixcolumns. Like in the *Encoder*, the *guide* bit, which indicates the duration of these steps, is reused, with its value dictated by the variable *Count*. When *Count* reaches N_r (i.e., once all rounds of round key have been employed in the AddRoundkey stage), the decryption process is concluded by once more performing InvShiftrows, InvSubbytes, and AddRoundkey steps. Among these four stages, the AddRoundkey mirrors the same process as in the *Encoder*. Conversely, InvSubbytes, InvShiftrows, and InvMixcolumns each execute the inverse functions of their respective matching stages in the *Encoder*.

3.2. AESware_Arbiter : Scheduling aesware utilization across cores

AESware_Arbiter logic plays a pivotal role in the design of multicore processors equipped with AESware, enabling the design of low-power processors by facilitating the shared use of a single AESware among all cores, instead of dedicating individual AESwares to each core. It serves to prevent potential data collisions and conflicts when multiple cores try to access AESware simultaneously. A data collision could occur if multiple cores attempt to send input data (such as the cipher text, plain text, or key) to AESware at the same time, causing the data streams to become mixed or corrupted. Additionally, a data conflict might arise if one core's AES operation is mistakenly interrupted or overwritten by another core's request, leading to errors such as sending the wrong operation results to the incorrect core. By carefully managing access to AESware, *AESware_Arbiter* ensures efficient scheduling and prevents both data corruption and unnecessary energy waste due to idle waiting.

As depicted in Fig. 2, the *AESware_Arbiter* consists of the *Arbiter_scheduler* and three arrays: *Arbiter_queue*, *Priority_array*, and *Age_array*. The *Arbiter_queue* primarily functions to prevent data collisions. It stores the tag numbers of each core, with each core waiting for its turn until its number is emitted from the *Arbiter_queue*. When

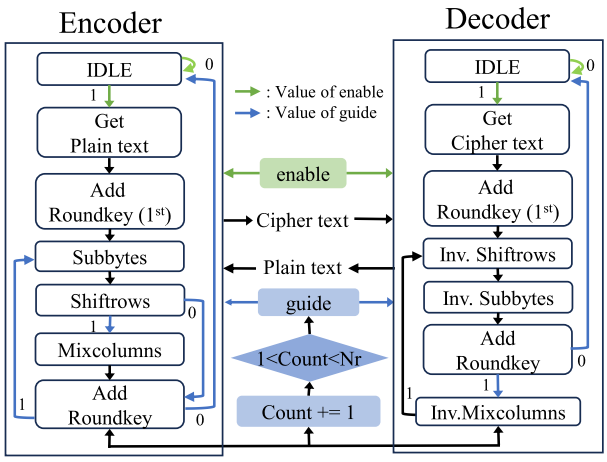

Fig. 3. Operation mechanisms of Encoder and Decoder.

the *AES_Operator* has completed all its stages and signals its IDLE state (i.e., when the *AESware_state* signal in Fig. 2 becomes one), the *AESware_Arbiter* outputs the core number generated by the *Topmost Element Selector*. This selector first searches for the lowest value in the *Priority_array* and then designates the corresponding element in the *Arbiter_queue* as the topmost core number. For example, if the element with the lowest value in the *Priority_array*, indicating the highest priority, is at index 3, the tag number of the core located in the 3rd position of the *Arbiter_queue* will be extracted. The core, upon recognizing its number, initiates communication with the *AES_Operator* via APB2 for AES operations.

Importantly, access requests to AESware from the cores are not immediately entered into *Arbiter_queue*, but first pass through *AESware_Scheduler*, which schedules tasks for energy-efficient operation of AESware and the cores. To elaborate further, if the number of cores wishing to utilize AESware simultaneously exceeds AESware's processing capacity, some cores will inevitably have to wait for their turn. This waiting time could potentially exceed the time it takes for a core to execute AES software on its own. If the processor is designed to apply low-power technologies such as dynamic power management (DPM) or dynamic voltage and frequency scaling (DVFS) to each core, this would reduce the power consumption of idle cores, and in most cases, waiting to use AESware would be more energy-efficient than cores running AES software themselves. However, embedded processors typically designed for edge/IoT devices do not incorporate such low-power technologies per core due to design overhead [21]. Therefore, waiting for an extended period to utilize AESware is not always the most energy-efficient option.

Arbiter_scheduler aims to maximize the utilization of AESware through efficient task placement, minimizing latency. It determines whether processing a task in the core software is more energy-efficient than processing it in the AESware, and if so, schedules the task to be executed by the core. The operational procedure of *Arbiter_scheduler* is illustrated in Algorithm 2. At its core, it employs a *Shortest Job First* (SJF) and *Priority* algorithm to minimize the queue waiting time. In detail, the algorithm responds to two primary scenarios:

1. New Request Arrival: When a new request is detected, *Arbiter_scheduler* assigns it the lowest priority and enqueues it into *Arbiter_queue*. With the inclusion of the anticipated duration of the new element, the SJF algorithm is freshly executed based on the estimated execution time of each element on *AES_Operator*. This estimation is obtained by substituting predefined parameters based on the key bit count of the element. After running the SJF algorithm, if any element has been downgraded in priority due to the entry of the new element, the Age of that element increments by one. This Age attribute plays a pivotal role in determining request priority. If the Age accrues to a third of the total core count, the priority of the element increases.
   For reference, in Fig. 2, the Age and $priority$ correspond to the *Age_array* and *Priority_array*, respectively. These arrays are interrelated and updated through the New Request Arrival algorithm, which is implemented via the *Threshold Detector* and *Priority Change Detector* in the figure. The *Threshold Detector* is a module that monitors the *Age_array* and modifies the *Priority_array*, whereas the *Priority Change Detector* is a module that monitors the *Priority_array* and modifies the *Age_array*. When the age exceeds the threshold, the *Threshold Detector* changes the *Priority_array*, and the age of the element with the changed priority is reset to 0 by the *Priority Change Detector*.
2. *AES_Operator* Readiness: If the $AESware\_state$ signals a value of 1, indicating the *AES_Operator* is ready for a new request, *Arbiter_scheduler* pops the highest priority element. The core, which then verifies its identification, initiates communication with the *AES_Operator* via APB2.

Algorithm 2 Energy-Efficient Scheduling Procedure for AESware Utilization.

N_{core} : the number of core number; SJF : method aligning the jobs in order of the shortest time to process; Age_threshold:1/3 of the number of cores; .put : putting the argument to the least prior queue; .pop : getting the argument out to *AESware_Operator*; .remove : sending the argument back to the core for software processing

```
procedure Arbiter_scheduler is
    while True do
        if new request arrived then
            newRequest.Age = 0  Arbiter_queue.put(new request)
            SJF(Arbiter_queue) for i in 1 to N_core do
                if Arbiter_queue[i].previous_priority <
                Arbiter_queue[i].current_priority then
                    AESware_queue[i].Age += 1
                if Arbiter_queue[i].Age > Age_threshold then
                    Arbiter_queue[i].priority− =1
            does_Core_Need_Software_Processing()
        if AESware_state is 1 then
            Arbiter_queue.pop(the most prior element) for i in 1 to N_core do
                Arbiter_queue[i].priority− = 1
            is_Software_Processing_Preferable()
        if all requests are completed then
            Break
function is_Software_Processing_Preferable:
    for i in 1 to N_core do
        HW_time = 0 for j in 1 to i do
            HW_time+ = Arbiter_queue[j].estimated_time
        if HW_time>SW_time then
            Arbiter_queue.remove(i^th element)
```

For both scenarios, *is_Software_Processing_Preferable* function is executed last to determine whether it is better for the request to be processed in software by the core instead. This function assesses whether waiting in the *Arbiter_queue* is indeed the most energy-efficient action for each element. Given the frequent simultaneous presence of multiple requests in the queue in multicore processors, this function critically shapes the HW-SW intermediate character of the AESware. If it determines that a specific element would execute faster by immediately running the AES operation in software than by waiting for its preceding requests, it removes that element from the *Arbiter_queue*. Additionally, the direction bits, allocated to the lower 2 bits of the data sent via APB1, are used to signal the core that initiated the request to commence software processing. This strategy not only reduces the waiting time for the removed core but also decreases the wait time for lower-priority requests.

Table 1
Device utilization and execution time comparison with previous works.

(a) Device utilization comparison with previous works.

	AESware	Xilinx [22]	[23]	[24]	[25]	[26]
LUT	4476	16 252	15 376	8129	3582	16 732
FF	2009	9934	2309	–	2580	–

(b) Execution time comparison with previous works.

	AESware	[22]	[23]	[24]	[25]	[26]
f_{clk} (MHz)	50	250	237	–	33	100
t_{exec} (ns)	4880	140	142.5	–	2667	1630

Table 2
List of cores explored for multicore processor development and their respective power consumption when synthesized using a 45 nm process technology (see [27–31]).

	Power (μW)
AESware	4438
ORCA [27]	2110
e203 [28]	4781.3
biRISC-V [29]	5623
Rocket [30]	6700
cv32e40p [31]	6879.7

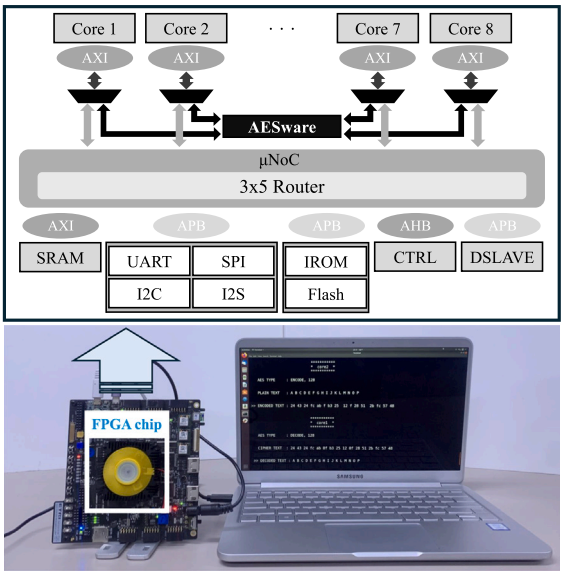

Fig. 4. The designed multicore processor architecture and its FPGA prototype.

4. Design and implementation of multicore processor prototypes

4.1. Lightweightness evaluation of AESware

Before integrating the AESware into embedded processor design, we conducted a preliminary evaluation of its lightweight characteristics. We compared it with Xilinx's standard AES IP [22] and four types of IPs developed in previous research [23–26], especially focusing on those that support all three types of operations while considering area efficiency. Table 1(a) reports the results of this comparison. AESware consumes a total of 4476 Look-Up Tables (LUTs) and 2009 Flip-Flops (FFs). Of these, *AESware_Operator* uses 3802 LUTs and 952 FFs, while *AESware_Arbiter* uses 674 LUTs and 1057 FFs. Notably, the *AESware_Arbiter* is an addition to AESware when the target processor is multicore. While it is about 4.9% larger than the smallest IP in the comparison group [25], considering the inclusion of scheduling logic for multicore systems (AESware without the *AESware_Arbiter* consumes approximately 29.6% less resources than [25]), we can confirm that it still meets its lightweight design goals. Furthermore, the lightweight nature of AESware becomes even more pronounced in a multicore context. In our proposed shared AESware architecture, the resource consumption of AESware remains constant at 4476 FFs and 2009 LUTs regardless of the number of cores, whereas the resource usage of the other IPs in the comparison group increases multiplicatively with the number of cores, thus widening the gap significantly.

Additionally, beyond comparing FPGA resource consumption, we also compared the AES execution time of AESware with previous works, and the results are provided in Table 1(b) for reference. In this table, f_{clk} represents the operating clock frequency of each AES IP, and t_{exec} indicates the execution time for performing AES-128 decryption. As expected, given AESware's design priority is focused on achieving lightweight characteristics, its processing speed is lower compared to previous works that prioritize performance, such as Xilinx's standard IP. While AESware's performance appears slightly behind that of [25], which also emphasizes efficiency, this difference arises from a key design variation: unlike AESware, which implements the complete AES operations, the previous work employs a Pseudo-Random Number Generation (PRNG) algorithm to replace the key expansion phase, thereby inherently reducing execution time.

4.2. Multicore processor implementations

To verify the functionality and evaluate the performance of AESware in multicore processors, we need to develop various prototype processors integrated with AESware. First, we selected the two most suitable RISC-V cores for use in multicore processors for edge/IoT devices from the numerous publicly available cores. To do this, we explored various RISC-V cores, as listed in Table 2. We focused particularly on the power consumption and performance of the cores, which could directly impact the energy-saving evaluation of AESware. We synthesized these cores using the 45 nm 1.1 V supply FreePDK [32] and measured their respective power consumption, as reported in the table. We also estimated the performance of each core based on the size of its ALU and the presence of an FPU. As a result, we selected the ORCA core to represent low-power, low-performance cores and the Rocket core to represent high-power, high-performance cores.

We then, designed four different types of AES-specific RISC-V multicore processors, based on the processor architecture depicted in Fig. 1: single-core, dual-core, quad-core, and octa-core processors, each based on the ORCA core. We utilized an EDA tool, RISC-V eXpress [33], for the processor RTL designs and prototyped them using Vivado [34] on the Xilinx Kintex Ultrascale+ FPGA board. We engineered all processors to operate at a 50 MHz clock frequency.

Fig. 4 presents one of the architecture we developed (i.e., octa-core processor), and its FPGA prototype. The IPs, excluding the cores, were designed identically across all processors. As seen in the figure, these include a 64 K SRAM as the main memory, Flash as a non-volatile memory, a lightweight Network-on-Chip (μNoC [10]) for system interconnect, a control module (CTRL) for boot and reset, and standard external I/O interface modules such as UART, SPI, I^2S, and I^2C for connecting various embedded/IoT sensors. In the prototyped processor architecture, the instructions and data needed by the cores are stored in SRAM. Communication between the cores and SRAM, when exchanging instructions and data, is handled via AXI protocol-supported modules and the μNoC. Similarly, when exchanging data with external I/O devices, the cores communicate via protocol support modules (UART, SPI, I^2C, I^2S), also through the μNoC. However, to avoid latency and congestion caused by the system interconnect, communication with AESware is handled directly, bypassing the μNoC. As illustrated in the figure, each core is connected to either the μNoC or AESware via a Mux, ensuring efficient communication with AESware for handling AES tasks.

To verify the proper operation of the processors, we connected various wearable sensors (e.g., camera, accelerometer, temperature sensor, PPG sensor) to the prototypes. We developed several simple applications that combine the data from these sensors to assess the user's

Table 3
Power consumption results of the multicore processors synthesized with 45 nm technology.

(a) Power consumption breakdown of AESware (*AWR* for abbreviation).

Module name			Power (μW)		
AESware	Arbiter		445		4438.0
	Operator	Encoder	234.2	3993.0	
		Decode	233.2		
		Roundkey_generator	3208		
		char2hex	15.6		
		other	202		

(b) Power consumptions of 12 different ORCA-based multicore processors.

Core: ORCA	Single-core	Dual-core	Quad-core	Octa-core
P_{w/o_AWR} (μW)	8514.3	11 298.9	15 029.2	24 915.8
P_{each_AWR} (μW)	12 507.2	19 284.7	31 001.0	56 859.7
P_{shared_AWR} (μW)	12 507.2	15 736.8	19 467.1	29 353.8
OH_{shared_AWR}	46.8%	39.2%	29.5%	17.8%

(c) Power consumptions of 12 different Rocket-based multicore processors.

Core: Rocket	Single-core	Dual-core	Quad-core	Octa-core
P_{w/o_AWR} (μW)	22 910.8	40 091.1	74 451.6	143 182.8
P_{each_AWR} (μW)	26 903.7	48 077.0	90 423.5	175 126.7
P_{shared_AWR} (μW)	26 903.7	44 529.1	78 889.7	147 620.7
OH_{shared_AWR}	17.4%	11.1%	5.9%	3.1%

condition and then process the results using AES. We allocated these applications to different cores of each processor and conducted experiments to execute them concurrently. This process was replicated across all processors, confirming their normal operation in handling multiple sensor inputs and AES data processing simultaneously. Regarding the resource consumption of the prototypes, the single-, dual-, quad-, and octa-core processors used 20 494, 29 093, 43 670, and 75 019 FPGA resources, respectively. Among these, the proportion of resources utilized by AESware was low, accounting for just 21.8%, 15.3%, 10.2%, and 6.0%, respectively.

Next, we designed and prototyped single-, dual-, quad-, and octa-core processors using the Rocket core. The rest of the IPs were kept identical. We verified that AESware functioned correctly in these processors as well. For the single-, dual-, quad-, and octa-core Rocket processors, the FPGA resource consumption for each number of cores was 23 349, 49 877, 87 640, and 162 531, respectively. The share of AESware within the processor was 19.2%, 8.9%, 5.1%, and 2.7%, respectively. Due to the larger proportion occupied by the Rocket core in the overall processor, it resulted in a remarkably low proportion of AESware within the processors.

5. Experimental results

5.1. Power consumption and overhead analysis

We synthesized the 24 different single, dual, quad, and octa-core processors based on both ORCA and Rocket cores using 45 nm 1 V supply FreePDK [32]. Table 3 presents the synthesis simulation results using Synopsys Design Compiler [35]. Initially, Table 3(a) reports the power consumption of individual modules within AESware.

Subsequently, Table 3(b) displays the power consumption of ORCA-based multicore processors. Here, P_{w/o_AWR} represents the power consumption of processors processing AES solely in software without using AESware, P_{each_AWR} denotes the power for processors with a dedicated AES accelerator for each core (i.e., each core has an independent *AES_Operator*), and P_{shared_AWR} is for processors with our proposed shared single AESware. In the table, for single-core processors, since they do not include the *AESware_Arbiter*, P_{each_AWR} is the same as P_{shared_AWR}. However, as expected, for other configurations, P_{shared_AWR} is less than P_{each_AWR}, and this difference becomes more pronounced with an increasing number of cores. For example, in octa-core processors, the increase in P_{shared_AWR} compared to P_{w/o_AWR} reaches 128%, while the increase of P_{each_AWR} compared to P_{w/o_AWR} is only 18%, clearly demonstrating the benefits of a shared AESware architecture. Additionally, we have reported the power overhead due to the shared AESware, $OH_{shared\_AWR} = \frac{P_{shared\_AWR} - P_{w/o\_AWR}}{P_{w/o\_AWR}} \cdot 100\ (\%)$ in the table. For single-core processors, the power overhead due to AESware was about 46.8%, which reduces as the number of cores increases, being only 17.8% for octa-core processors.

Table 4
Comparison of energy consumption between performing AES tasks in software on the ORCA core and using AESware. Results from the single-core processors.

	128 bit	192 bit	256 bit
E_{w/o_AWR} (mJ)	524.6	628.5	735.1
E_{shared_AWR} (mJ)	61.0	80.3	97.6
ES (%)	88.4	87.2	86.7

Similarly, following the FPGA prototype development, we also conducted power analysis for multicore processors using Rocket cores, with the results reported in Table 3(c). The Rocket core, known for its higher performance, inherently consumes more power than the ORCA core. Consequently, the OH_{shared_AWR} due to the addition of AESware is relatively small, amounting to just 17% in single-core processors and a mere 3.1% in octa-core processors, which is significantly lower compared to that in ORCA processors. Meanwhile, the difference between P_{shared_AWR} and P_{each_AWR} is also not as dramatic as in the case of ORCA processors. This indicates that in cases like the Rocket core, which is more suitable for high-performance processors rather than IoT/Edge device processors, the benefits of sharing AESware are still positive but may not have a significant impact on the overall power consumption.

This subsection has analyzed the power consumption of AESware and its overhead in processors employing it. Moreover, we included an analysis of power consumption for processors with a dedicated AESware for each core, to examine the advantages of the shared single AESware architecture. The findings indicate that AESware's overhead is quite small, and the benefits of a shared architecture are clear. However, this analysis alone is insufficient as a measure of the effectiveness of our proposed solution: it lacks consideration of time, specifically the performance improvements offered by our solution. Hence, in the next section, we will analyze the effect of our proposed solution on the energy efficiency of multicore processors.

5.2. Evaluation of energy efficiency

To evaluate the energy-saving (ES) effect of our developed AESware, we first investigated the increase in energy efficiency due to its lightweight hardware design. For this purpose, we derived experimental values for the energy consumption when AES operations are processed in software on a single ORCA core, E_{w/o_AWR}, and when processed using AESware, E_{shared_AWR}. The results are reported in Table 4. As indicated in the table, we calculated the ES for three types of AES operations (i.e., 128-, 192-, and 256-bit key type AES operations), and observed that the ES is slightly better for the 128-bit operation. It is noteworthy that the processing times for each operation in software were 3081, 3691, and 4317 clock cycles, respectively, while the processing times using AESware were 244, 321, and 390 clock cycles. This demonstrates that our proposed hardware offers an acceleration effect of approximately 11 times. Coupled with the acceleration performance and the lightweight design of AESware, we confirmed that the resultant ES could reach up to 88.4%.

Next, to thoroughly evaluate the energy-saving effects of our proposed solution in multicore processors, we compared E_{w/o_AWR} and E_{shared_AWR} in dual-, quad-, and octa-core processors. Additionally, in this experimental work, we included multicore processors equipped with individual dedicated AES accelerators for each core in our comparison group, measuring their energy consumption $E_{per-Core}$. Notably, to focus more on the advantages of the proposed shared architecture, we

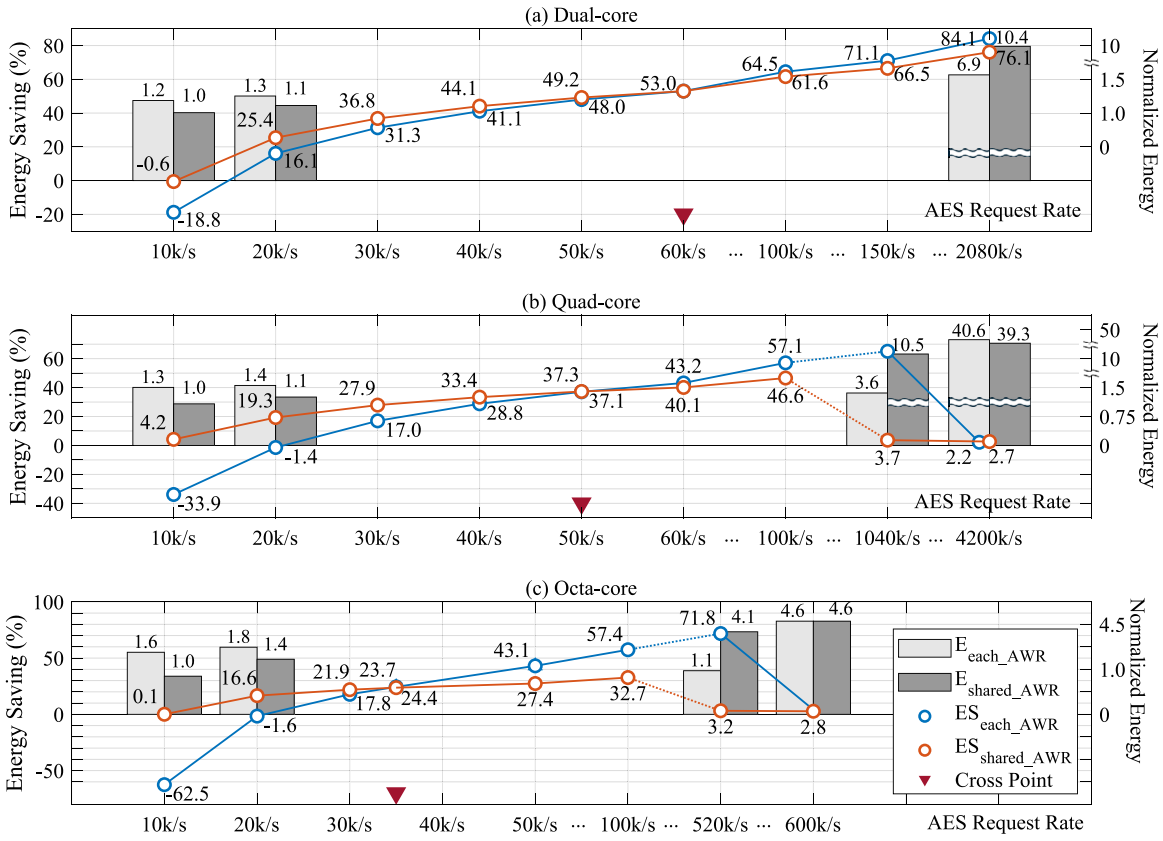

Fig. 5. Energy consumption and efficiency results of dual-, quad-, and octa-core processors without AESware, with shared AESware, and with individual AESware for each core across various AES request rates.

used our lightweight accelerators (i.e., the *AES_Operator* of AESware) to measure $E_{per-Core}$, thus $E_{per-Core} = E_{each\_AWR}$. These evaluations encompassed processors based on both ORCA and Rocket architectures. However, despite the parallel nature of outcomes and variations in magnitude, this paper primarily reports findings relevant to ORCA cores, considering their suitability for edge/IoT devices.

Fig. 5 presents the results of the energy comparison for multicore processors. The graphs within the figure represent the outcomes for (a) dual-, (b) quad-, and (c) octa-core processors. Each graph's left vertical axis denotes the ES, while the right vertical axis represents the normalized energy values based on the smallest one of the measured E_{w/o_AWR}'s (i.e., the energy consumed by the dual-core processor processing the lowest frequency of AES operations in software, corresponding to the leftmost point in graph (a)). The x-axis of the graphs indicates the AES Request Rate (*RR*), which is the number of times a core requires AES operations per real-time second. For instance, with an *RR* of 10 k/s, the shared AESware would receive 4000 AES operation requests per second. Accordingly, by varying the *RR*, we reflected the changing frequency of AES operations required by applications, allowing us to evaluate how energy-efficiently the requested AES operations were processed under various scenarios.

In Fig. 5, the red line with the red points represent the energy savings of processors with shared AESware compared to those processing AES in software, $ES_{shared\_AWR} = \frac{E_{w/o\_AWR} - E_{shared\_AWR}}{E_{w/o\_AWR}} \cdot 100$ (%). The blue line and the blue points indicate the energy savings of processors where each core possesses its AESware, again compared to software-based AES processing, $ES_{each\_AWR} = \frac{E_{w/o\_AWR} - E_{each\_AWR}}{E_{w/o\_AWR}} \cdot 100$ (%). We categorized the range of the x-axis, *RR*, into three sections: *low-*, *mid-*, and *high-RR*. Firstly, the *low-RR* section is where no benefit is gained from AESware, i.e., $ES_{shared\_AWR} \leq 0$. This section corresponds to scenarios where the AES operations demanded are relatively very minor compared to other computations, resulting in the power overhead of AESware, as reported in Table 3(b), outweighing the temporal benefits of rapid AES processing. As depicted in Fig. 5, this section ends for all processors when *RR* reaches approximately 10 k/s.

The sections where energy gains are realized from our proposed solution are identified as the *mid-RR* and *high-RR* sections. We define the *mid-RR* section as the range where processors based on the shared AESware architecture are more energy-efficient than those with per-core AESware, and the *high-RR* section as the range where they are not. In other words, as illustrated in the figure, these sections are distinguished based on the cross point (*CP*), where ES_{shared_AWR} and ES_{each_AWR} converge. This implies that at lower *RR*s, the shared AESware could immediately handle all requested AES operations on its own. However, as *RR* increases, AES operations gradually begin to accumulate in the queue, leading to performance degradation. By the time *RR* reaches the *CP*, the processing speed of shared AESware has already slowed down compared to per-core AESware, to the extent that the reduced power overhead achieved by the shared architecture is offset.

In dual-, quad-, and octa-core processors, the *CP* occurs at 60 k/s, 50 k/s, and 35 k/s, respectively. Furthermore, the maximum ES_{shared_AWR} in the *mid-RR* range for these processors is 49.2%, 37.3%, and 23.7%, respectively. This implies that as the number of cores increases, the *CP* decreases and the efficacy of the shared AESware architecture also tends to diminish. The reason is the increasing workload that the shared AESware must handle alone as the number of cores increases. Interestingly, as the number of cores increases, the difference in ES_{shared_AWR} compared to ES_{each_AWR} in the *mid-RR* range becomes more pronounced, with the shared architecture showing better results by 15.3% in dual-core, 28.4% in quad-core, and 38.5% in octa-core processors compared to the per-core architecture. This trend is attributed to the power consumption of AESware increasing significantly in the per-core architecture, while remaining consistent in the shared

architecture as the number of cores increases. From these results, we can conclude that, except in extreme cases where the number of cores in embedded processors is very high and each core demands a substantial AES workload – which is not typical – the shared AESware architecture is the most suitable for multicore processors.

Meanwhile, the *high-RR* range includes scenarios where the *AES_Arbiter* begins scheduling operations by sending back requests to cores that were waiting for AES operation processing, especially when the number of requests in the *Arbiter_queue* exceeds a certain threshold. In this range, although our proposed solution shows slightly less energy savings compared to the per-Core AESware architecture, it is crucial to note that the savings remain positive. As seen in Fig. 5, the magnitude of savings continues to increase until it reaches the maximum ES. In other words, even in the atypical *high-RR* scenario, while the gains are less compared to the per-Core AESware architecture, our solution still demonstrates significant energy efficiency improvements, reaching maximums of 76.1%, 46.6%, and 32.7% in dual-, quad-, and octa-core processors, respectively.

Observing the graphs for quad- and octa-core processors in the figure, it is evident that ES_{shared_AWR} begins to decrease once *RR* surpasses a certain threshold. This trend is observed when *RR* becomes significantly high, leading to the majority of AES operation requests being processed in software by individual cores. Notably, in quad-core processors, ES_{shared_AWR} falls to single digits when *RR* reaches 1040 k/s, and in octa-core processors, this occurs at 520 k/s. A similar pattern is seen in processors with a per-Core AESware architecture. Following our proposed scheduling method, if the dedicated AES accelerators are unable to handle all the required AES operations, the cores are instructed to process them in software. Consequently, although the ES_{each_AWR} remains higher than the points for shared AESware architecture, we observed a significant decrease in ES_{each_AWR} when *RR* exceeds 4200 k/s in quad-core processors and 600 k/s in octa-core processors.

The key results from the experimental work on the energy-saving effectiveness of AESware can be summarized as follows:

- In the *mid-RR* range where the shared AESware architecture proves to be the best solution, the ES_{shared_AWR} for dual-, quad-, and octa-core processors are 49.2%, 37.3%, and 23.7%, respectively. In this range, the shared AESware architecture demonstrates superior performance over the per-Core AESware architecture, with maximum improvements of 15.3%, 28.4%, and 38.5% for dual-, quad-, and octa-core processors, respectively.
- In the *high-RR* range, the energy-saving effect of our proposed solution continuously increases until the volume of AES operations requested becomes too large compared to the number processed by the shared AESware, leading to the majority being handled in software. Consequently, the maximum ES_{shared_AWR} for dual-, quad-, and octa-core processors reach 76.1%, 46.6%, and 32.7%, respectively.

6. Discussion and future work

In this paper, we have presented AESware, a lightweight hardware accelerator tailored for AES operations in embedded processors, and have proposed a shared architecture and scheduling algorithm to ensure its energy-efficient use in multicore processors. We have developed a processor prototype incorporating the proposed techniques to validate the efficacy of our approach. Our solution is not tied to any specific core or processor, offering both scalability to adapt to any available open RISC-V core and ease of development for creating AES-enabled multicore processors.

Regarding our research on designing a lightweight AES accelerator, like our AESware, there is a recent study that similarly places AES hardware externally to the core [36]. However, that study focuses on developing a co-processor optimized for a single RISC-V Rocket core using the RoCC (Rocket Custom Coprocessor) interface and TileLink protocol. As a result, unlike AESware, it may require substantial design modifications to add a new interface to other existing open RISC-V cores, which is a highly complex task. In contrast, AESware's strength lies in its compatibility across various cores without such modification challenges. Nevertheless, inspired by this research, we plan to include the development of a performance-enhanced version of AESware that supports RoCC as part of our future expansion plans for AESware.

Next, regarding the AESware shared architecture, we built upon recent research that proposed low-power design techniques for sharing specific IPs across multiple cores [37,38]. In those studies, we had developed a multicore system with integer cores and enhanced processor performance by implementing a separate FPU (Floating Point Unit) as a hardware IP, designed to accelerate floating-point operations. The FPU had been shared among multiple cores, thereby improving the processor's energy efficiency. Motivated by this prior work, we have introduced the concept of shared IP in this study, but advanced it by addressing the limitations of the simple round-robin scheduling used in the previous research. Specifically, we have developed a new scheduling algorithm and hardware logics that enable priority-based resource allocation among cores, allowing for more efficient sharing of resources. Moreover, in the previous architecture, the relationship between the shared IP and the cores was relatively straightforward, where the cores would request access to the shared IP, and the IP would automatically accept these requests. In contrast, our shared architecture incorporates more advanced functionality, where AESware can assess the situation and, if necessary, request that the core itself handle the AES operations directly. The intensive experimental work conducted on the shared architecture in this study has demonstrated its significant potential for low-power design. Based on these findings, we plan to further refine and advance the shared architecture as part of the future work for AESware.

Additionally, beyond improvements in hardware design, advances in semiconductor device research could further enhance the efficiency of AESware-equipped processors if low-power/high-performance transistors are used in future manufacturing of AESware. Recent studies on transistors for high-efficiency, low-power processor design [39–43] have been active. Notably, transistors typically used in biosensing applications, such as ISFETs [44], offer the potential to rapidly perform real-time AES encryption on biometric data. Incorporating these emerging technologies into AESware processor designs is expected to further maximize performance and energy efficiency. In our future work to advance AESware, we plan to explore cross-layer optimization across device, circuit, and architecture levels, leveraging these innovations.

7. Conclusion

This study has addressed the critical challenges in the development of energy-efficient and secure multicore processors for edge devices using open RISC-V cores. Through the proposed solution, we have introduced *AESware*, a dedicated and lightweight hardware optimized for AES (Advanced Encryption Standard) tasks, significantly enhancing the energy efficiency of multicore processors. Contrary to the conventional methods of processing AES tasks in software within cores of existing embedded processors, or the straightforward approach of embedding individual AES accelerators in each core for improvement, our shared AESware architecture has proven to be a more efficient approach in terms of energy consumption. Under the proposed precise measurement strategy, the comprehensive experimental work involving the development and testing of 24 processors clearly demonstrated the advantages of our approach. Processors equipped with AESware achieved up to 76%, 47%, and 33% energy savings in dual-, quad-, and octa-core configurations, respectively, and showed greater efficiency in running general AES applications compared to processors with individual AES accelerators. Additionally, the solution's compatibility with various open RISC-V cores offers flexibility and scalability, making it

an attractive option for a wide range of edge devices. Ultimately, this research aims to make a meaningful contribution to the field of secure and energy-efficient embedded processor design. We hope that our efforts will assist in advancing the development of sophisticated edge devices.

CRediT authorship contribution statement

Eunjin Choi: Writing – original draft. **Jina Park:** Writing – original draft, Software. **Kyuseung Han:** Methodology, Formal analysis. **Woojoo Lee:** Writing – original draft, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Ji, F. Wu, Z. Zhu, L.-P. Chang, H. Liu, W. Zhai, Memory-efficient deep learning inference with incremental weight loading and data layout reorganization on edge systems, J. Syst. Archit. 118 (2021) 102183.

[2] H. Jiang, R. Chakravarthy, V.C. Ravikumar, A task parallelism runtime solution for deep learning applications using MPSoC on edge devices, in: Asia and South Pacific Design Automation Conference, ASPDAC, 2022, pp. 1–2.

[3] İ. Yazici, I. Shayea, J. Din, A survey of applications of artificial intelligence and machine learning in future mobile networks-enabled systems, Eng. Sci. Technol. Int. J. 44 (2023) 101455.

[4] K. Lee, S. Jeon, K. Lee, W. Lee, M. Pedram, Radar-PIM: Developing IoT processors utilizing processing-in-memory architecture for ultra-wideband radar-based respiration detection, IEEE Internet Things J. (2024) 1.

[5] A. Suyyagh, Z. Zilic, Energy and task-aware partitioning on single-ISA clustered heterogeneous processors, Parallel Distrib. Syst. 31 (2020) 306–317.

[6] H. Jang, S. Lee, H.-I. Park, W. Lee, Development of a low-power touch-based lifelogging processor, IEEE Trans. Circuits Syst. II 71 (8) (2024) 3910–3914.

[7] Y. Abid, A. Shuja, M. Ali, I. Murtaza, Output current boosting in triboelectric nanogenerators for applications in self-powered energy systems, Eng. Sci. Technol. Int. J. 55 (2024) 101749.

[8] Y. Jeon, E. Ham, J. Lim, J.-H. Kim, Hardware-software co-design of AES-CCM for Bluetooth LE security, in: International Conference on Electronics, Information, and Communication, ICEIC, 2023.

[9] U. Lee, H.K. Kim, J. Lee, M.H. Sunwoo, Area-efficient intellectual property (IP) design of advanced encryption standard, in: IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 70, 2023, pp. 3797–3801.

[10] K. Han, S. Lee, J.-J. Lee, W. Lee, M. Pedram, TIP: A temperature effect inversion-aware ultra-low power system-on-chip platform, in: IEEE/ACM International Symposium on Low Power Electronics and Design, ISLPED, 2019.

[11] H. Jang, K. Han, S. Lee, J.-J. Lee, S.-Y. Lee, J.-H. Lee, W. Lee, Developing a multicore platform utilizing open RISC-V cores, IEEE Access 9 (2021) 120010–120023.

[12] J. Park, E. Choi, K. Lee, J.-J. Lee, K. Han, W. Lee, Developing an ultra-low power RISC-V processor for anomaly detection, in: Design, Automation & Test in Europe Conf. & Exhib., DATE, 2023, pp. 1–2.

[13] E. Choi, J. Park, K. Lee, J.-J. Lee, K. Han, W. Lee, Day–night architecture: Development of an ultra-low power RISC-V processor for wearable anomaly detection, J. Syst. Archit. 152 (2024) 103161.

[14] M. Wazid, A.K. Das, R. Hussain, G. Succi, J.J. Rodrigues, Authentication in cloud-driven IoT-based big data environment: Survey and outlook, J. Syst. Archit. 97 (2019) 185–196.

[15] R. Mondal, H. Ngo, J. Shey, R. Rakvic, O. Walker, D. Brown, Efficient architecture design for the AES-128 algorithm on embedded systems, in: ACM International Conference on Computing Frontiers, 2020, pp. 89–97.

[16] M. Wei, G. Yang, F. Kong, Software implementation and comparison of ZUC-256, SNOW-V, and AES-256 on RISC-V platform, in: IEEE International Conference on Information Communication and Software Engineering, ICICSE, 2021, pp. 56–60.

[17] A. Zgheib, O. Potin, J.-B. Rigaud, J.-M. Dutertre, Extending a RISC-V core with an AES hardware accelerator to meet IOT constraints, in: SMACD / PRIME 2021; International Conference on SMACD and 16th Conference on PRIME, 2021, pp. 1–4.

[18] C. Duran, E. Roa, A 10pJ/bit 256b AES-SoC exploiting memory access acceleration, in: IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 69, 2022, pp. 1612–1616.

[19] National Institute of Standards and Technology (NIST), Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, National Institute of Standards and Technology, 2001.

[20] S. Yang, L. Shao, J. Huang, W. Zou, Design and implementation of low-power IoT RISC-V processor with hybrid encryption accelerator, Electronics 12 (20) (2023).

[21] W. Lee, Y. Wang, M. Pedram, VRCon: Dynamic reconfiguration of voltage regulators in a multicore platform, in: Design, Automation & Test in Europe Conf. & Exhib., DATE, 2014.

[22] Xilinx, AES, 2024, https://www.xilinx.com/products/intellectual-property/ef-di-aes.html. (Accessed 10 October 2024).

[23] N.S.S. Srinivas, M. Akramuddin, FPGA based hardware implementation of AES rijndael algorithm for encryption and decryption, in: International Conference on Electrical, Electronics, and Optimization Techniques, ICEEOT, 2016, pp. 1769–1776.

[24] T.M. Kumar, K.S. Reddy, S. Rinaldi, B.D. Parameshachari, K. Arunachalam, A low area high speed FPGA implementation of AES architecture for cryptography application, Electronics 10 (16) (2021).

[25] A.M. Ruby, S.M. Soliman, H. Mostafa, Dynamically reconfigurable resource efficient AES implementation for IoT applications, in: 2020 IEEE International Symposium on Circuits and Systems, ISCAS, 2020, pp. 1–5.

[26] E. Selvapriya, L. Suganthi, Design and implementation of low power advanced encryption standard cryptocore utilizing dynamic pipelined asynchronous model, Integration 93 (2023) 102057.

[27] ORCA, ORCA-risc-v, 2024, https://github.com/kammoh/ORCA-risc-v. (Accessed 10 October 2024).

[28] SI-RISCV, E203-risc-v, 2024, https://github.com/SI-RISCV/e200_opensource.(Accessed 10 October 2024).

[29] ultraembedded, biRISC-V, 2024, https://github.com/ultraembedded/biriscv. (Accessed 10 October 2024).

[30] Rocket, Rocket-risc-v, 2024, https://github.com/chipsalliance/rocket-chip. (Accessed 10 October 2024).

[31] ultraembedded, cv32e40p, 2024, https://github.com/openhwgroup/cv32e40p.git. (Accessed 10 October 2024).

[32] NCSU, FreePDK45, 2024, https://eda.ncsu.edu/freepdk/freepdk45. (Accessed 10 October 2024).

[33] K. Han, S. Lee, K.-I. Oh, Y. Bae, H. Jang, J.-J. Lee, W. Lee, M. Pedram, Developing TEI-aware ultralow-power SoC platforms for IoT end nodes, IEEE Internet Things J. 8 (6) (2021) 4642–4656.

[34] Xilinx, Vivado 2021.2, 2024, https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2021-2.html. (Accessed 10 October 2024).

[35] Synopsys, Design compiler, 2024, https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. (Accessed 10 October 2024).

[36] T. Gomes, P. Sousa, M. Silva, M. Ekpanyapong, S. Pinto, FAC-V: An FPGA-based AES coprocessor for RISC-V, J. Low Power Electron. Appl. 12 (4) (2022).

[37] J. Park, K. Han, E. Choi, S. Lee, J.-J. Lee, W. Lee, M. Pedram, Florian: Developing a low-power RISC-V multicore processor with a shared lightweight FPU, in: 2023 IEEE/ACM International Symposium on Low Power Electronics and Design, ISLPED, 2023, pp. 1–6.

[38] J. Park, K. Han, E. Choi, J.-J. Lee, K. Lee, W. Lee, M. Pedram, Designing low-power RISC-V multicore processors with a shared lightweight floating point unit for IoT endnodes, IEEE Trans. Circuits Syst. I. Regul. Pap. 71 (9) (2024) 4106–4119.

[39] A.K. Panigrahy, S. Hanumanthakari, S.B. Devamane, S.B. Choubey, M. Prasad, D. Somasundaram, N. Kumareshan, N.A. Vignesh, G. Subramaniam, D.P. M, R. Swain, Analysis of GAA junction less NS FET towards analog and RF applications at 30 nm regime, IEEE Open J. Nanotechnol. 5 (2024) 1–8.

[40] V.B. Sreenivasulu, A.K. Neelam, A.K. Panigrahy, L. Vakkalakula, J. Singh, S.G. Singh, Benchmarking of multi-bridge-channel FETs toward analog and mixed-mode circuit applications, IEEE Access 12 (2024) 7531–7539.

[41] U. Gowthami, A. Kumar Panigrahy, D. Shobha Rani, M. Nayak Bhukya, V. Bharath Sreenivasulu, M. Durga Prakash, Performance improvement of spacer-engineered N-type tree shaped NSFET toward advanced technology nodes, IEEE Access 12 (2024) 59716–59725.

[42] A.K. Panigrahy, V.V.S. Amudalapalli, D.S. Rani, M.N. Bhukya, H.B. Valiveti, V.B. Sreenivasulu, R. Swain, Spacer dielectric analysis of multi-channel nanosheet FET for nanoscale applications, IEEE Access 12 (2024) 73160–73168.

[43] A.C. M. Amani, et al., Design and comparative analysis of FD-SOI FinFET with dual-dielectric spacers for high speed switching applications, Silicon 16 (2024) 1525–1534.

[44] S. Majji, A.K. Panigrahy, D.S. Rani, M.N. Bhukya, C.S. Dash, Design and performance analysis of ISFET using various oxide materials for biosensing applications, IEEE Open J. Nanotechnol. 5 (2024) 23–29.