Proceedings of CONF-SEML 2026 Symposium: Computational Analysis and Modeling in Complex Intelligent Systems
DOI: 10.54254/2755-2721/2026.GU35297

System-Level Optimization and Co-Design of the SM4 Algorithm for RISC-V

Dawei Zhang[1], Xueyong Liu[2*]

[1]James Cook University, Townsville, Australia
[2]Institute of Microelectronics, Chinese Academy of Sciences, Beijing, China
**Corresponding Author. Email: liuxueyong@ime.ac.cn*

Abstract. In response to issues such as high computational latency and excessive energy consumption encountered when implementing the SM4 algorithm on general-purpose RISC-V processors, this paper proposes an assembly-level system optimization and hardware-software co-design scheme tailored for the RISC-V platform. Optimization is conducted at both the instruction level and the architectural level. First, memory access operations and inter-instruction dependencies are reduced through techniques including S-box lookup table reuse, shift-XOR instruction reordering, and lightweight pipelining. Next, modular assembly implementations are developed for the T-transformation, T′-transformation, and key expansion, with unified register conventions and calling interfaces established. Then, register rotation and loop unrolling mechanisms are introduced to enhance data reuse and lower control overhead. Finally, a hardware-friendly instruction sequence and interface specification are designed to facilitate smooth migration to custom instructions or coprocessor extensions in the future. Experimental results show that, compared to the C implementation, the optimized assembly version reduces the total instruction count by 37.5% and the cycle count by 54.2%, yielding an overall performance gain of 2.2-fold. Notably, the efficiency of the T′-transformation improves by nearly 3.9-fold. Without additional hardware cost, the proposed approach significantly boosts the execution efficiency of SM4 on RISC-V. It thus provides a systematic optimization path for embedded cryptographic implementations that ensures performance, scalability, and portability.

Keywords: SM4, RISC-V, Assembly-level Optimization, Hardware-Software Co-design, System-level Optimization

1. Introduction

With the rapid development of the Internet and Internet of Things technologies, the security of data transmission and storage has become increasingly prominent. Symmetric encryption algorithms, due to their high computational efficiency and simple implementation, have become one of the key technologies for ensuring information security [1]. Owing to its open-source, modular, and extensible characteristics, RISC-V provides an ideal platform for the efficient implementation of symmetric cryptographic algorithms [2]. However, in resource-constrained embedded systems, the

DOI: 10.54254/2755-2721/2026.GU35297

software implementation of the SM4 algorithm often faces problems such as low execution efficiency and high overhead. Therefore, how to improve its execution efficiency while maintaining security has become a research hotspot in both academia and industry.Studies have shown that hardware acceleration through instruction set extension or dedicated coprocessors can achieve a better balance between power consumption and code size [3], significantly improving processing throughput, especially in resource-constrained but security-sensitive application scenarios such as 5G and the Internet of Things [4].

Since the release of the RISC-V instruction set, the implementation and acceleration of cryptographic algorithms on this architecture have become a research hotspot. There are two typical technical routes. In terms of software-first optimization, Kwon et al. [5] proposed an optimized implementation model of SM4 based on instruction-level scheduling. Its core idea is instruction reorganization for S-box lookup and round function logic. By reducing memory accesses and logic instruction dependencies, the cycles per byte (cpb) are significantly reduced, proving that portable performance improvement can still be achieved without modifying the ISA. However, their method focuses more on local operator optimization and has not yet formed a systematic assembly-level optimization framework. There is still room for improvement in register reuse and instruction pipelining.In addition, Nişancı et al. [6] proposed a performance evaluation framework for symmetric cryptographic algorithms on the RISC-V platform. Its core modules include instruction-level performance modeling and optimization analysis for algorithms such as AES and SM4. Through instruction statistics and pipeline simulation, it quantitatively evaluates algorithm efficiency and provides a basis for software optimization. However, this method lacks in-depth discussion of systematic assembly-level optimization strategies and does not form a reusable and extensible optimized code framework.In terms of hardware coprocessors and custom instructions, Wang Jinglun et al. [7] proposed an SM4 coprocessor design scheme based on the NICE interface. Its core module is a dedicated cryptographic processing unit integrated into the Hummingbird E203, which, together with a small number of custom instructions, achieves significant encryption and decryption acceleration under resource and power constraints, thereby verifying the high-performance potential of the hardware path. However, this design depends on a specific SoC platform and extension environment, and its portability and plug-and-play capability on general RISC-V cores are relatively weak.At the same time, Saarinen et al. [8] proposed a lightweight instruction set extension method. Its core module is a dedicated cryptographic instruction set for AES and SM4 round functions, which improves algorithm execution efficiency through instruction-level extensions. However, the scheme still requires processor vendors to support the instruction extension, and the standardization level of the hardware-software interface is insufficient, which limits its wide deployment.

Although the above methods have achieved significant results in software optimization and hardware acceleration, they still have shortcomings in the construction of systematic assembly optimization frameworks, the standardization of hardware-software interfaces, and cross-platform portability. It is therefore difficult to realize an efficient, flexible, and smoothly migratable SM4 acceleration solution on a general RISC-V platform.To address these problems, this paper proposes a systematic assembly-level optimization method for SM4 on RISC-V without modifying the ISA. Around S-box lookup, shift-XOR reordering, lightweight pipelining, register rotation, and loop unrolling, a reusable implementation framework is formed, and an instruction sequence specification that can be smoothly migrated to a coprocessor is designed, thereby addressing the issues of efficiency, flexibility, and smooth migration.

The primary innovations and contributions of this work are as follows:

DOI: 10.54254/2755-2721/2026.GU35297

• This paper proposes a systematic assembly-level optimization framework for SM4 on RISC-V. Without ISA modifications, it integrates techniques such as S-box lookup table reuse, shift-XOR instruction reordering, and lightweight pipelining to reduce memory accesses and inter-instruction dependencies, thereby significantly improving software execution efficiency.

• We propose a modular and reusable architecture that encapsulates the T-transformation, T′-transformation, and key expansion into independent assembly modules with standardized registers and interfaces. This design enhances code reusability and maintainability while laying a clear foundation for future hardware-software co-optimization.

• This work ensures forward-looking extensibility by designing a hardware-friendly instruction sequence and a scalable interface. This allows the optimized software to seamlessly transition to future custom instructions or coprocessors, creating a structured pathway for sustained performance gains.

2. Related work

2.1. RISC-V instruction set architecture

RISC-V is a new open-source instruction set architecture based on the Reduced Instruction Set Computer (RISC) philosophy. It was proposed by the University of California, Berkeley, and is characterized by openness, modularity, and scalability [9]. RISC-V can also be extended with custom instructions to match specific application requirements [10]. For example, vector extensions can significantly reduce the number of instructions in computer vision workloads [11]. The instruction formats of RISC-V mainly include six types: R, I, S, U, B, and J, which makes encoding flexible in different scenarios. Compared with traditional ARM and x86 architectures, RISC-V provides custom instruction spaces (custom-0 to custom-3), enabling developers to extend specific functions according to application needs and providing strong support for hardware acceleration of encryption algorithms such as SM4 [12].

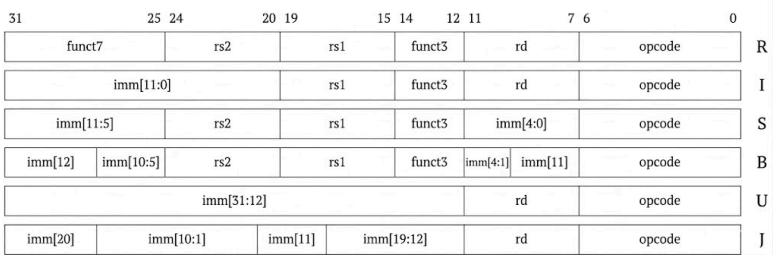

Figure 1. Basic RISC-V instruction formats

2.2. Principles of the SM4 algorithm

The SM4 algorithm is a block symmetric encryption algorithm with a block length of 128 bits and a key length of 128 bits, using a 32-round iterative structure. The algorithm represents a 128-bit plaintext as four 32-bit words. In each round, a new word is updated according to the round key and the state window is advanced. Its encryption round function can be expressed as:

$$X_{i+4} = X_i \oplus T\left(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i\right), i = 0, 1, \ldots, 31 \tag{1}$$

where $T(\cdot)$ is a composite transformation consisting of nonlinear substitution $\tau(\cdot)$ and linear transformation $L(\cdot)$:

DOI: 10.54254/2755-2721/2026.GU35297

$$T(x) = L(\tau(x)) \tag{2}$$

The nonlinear substitution $\tau(\cdot)$ is usually implemented by S-box lookup: the 32-bit input is split byte by byte, each byte is mapped through the S-box, and then recombined into a 32-bit output, thereby providing cryptographic nonlinearity and resistance against differential and linear attacks [13]. In the linear transformation part, SM4 uses cyclic shifts and XOR operations to achieve diffusion, in the form of:

$$L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24) \tag{3}$$

It can therefore be seen that the key time-consuming parts of a single round are mainly concentrated in: (1) byte-level splitting and S-box memory access; and (2) the long dependency chain formed by multiple cyclic shifts and XOR operations. This also explains why, when implementing SM4 on embedded or general-purpose processors, the number of memory accesses, register pressure, and instruction scheduling significantly affect the overall performance.

The key expansion process is similar to the encryption round structure: the initial key is divided into four 32-bit words, and 32 round keys rk_i are generated by combining system parameters FK and round constants CK . The composite transformation used in key expansion is denoted as $T'(\cdot)$, which also consists of S-box substitution and linear transformation, but the linear diffusion form is different:

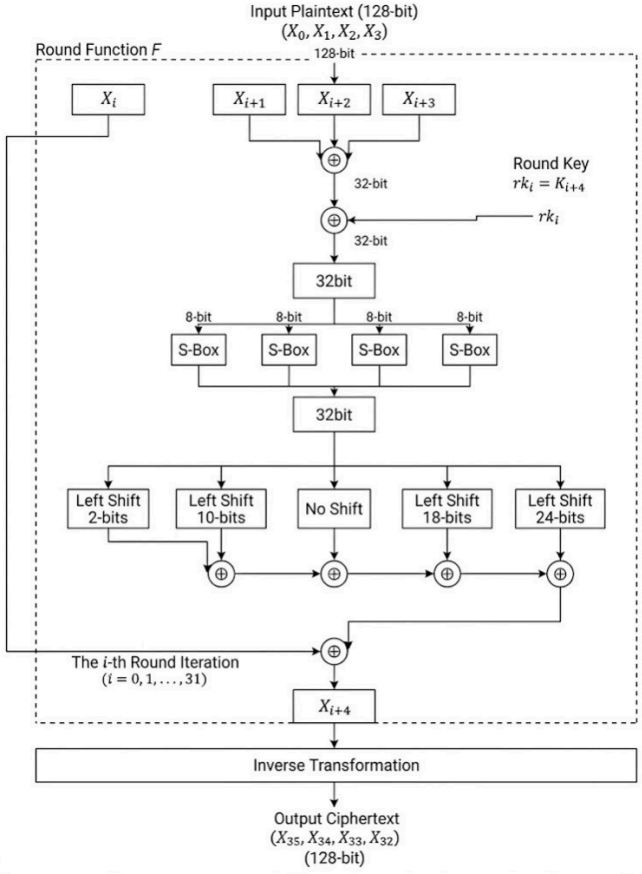

Figure 2. Encryption flow of the SM4 block cipher algorithm

DOI: 10.54254/2755-2721/2026.GU35297

Therefore, at the implementation level, $T(\cdot)$ and $T'(\cdot)$ can be abstracted as two core submodules: they share the S-box lookup process, but differ in the shift items and instruction sequence of the linear transformation, which provides a basis for modular implementation and code reuse.

In practical applications, SM4 can operate in multiple block cipher modes such as ECB and CBC to satisfy confidentiality requirements in different scenarios. At the same time, in hardware and embedded implementations, side-channel attack protection, such as power analysis resistance, must also be considered [14]. In addition, the public security evaluation of SM4 is still ongoing. In recent years, studies on reduced-round versions, such as 25-round linear analysis, have further defined its security boundary [15].

3. Design and implementation of the SM4 coprocessor

3.1. Overall process

Based on the overall process shown in Figure 3, this paper uses RARS under the Windows environment to build an RV32 assembly development and debugging platform for implementing and optimizing SM4. First, the system input/output and data layout are determined: the 128-bit plaintext/ciphertext is represented as 4×32-bit state words, and the register mapping and access conventions for the round key array, S-box base address, and temporary variables are clearly defined. Second, the core functional modules are completed: the T transformation (S-box substitution + linear transformation L), the T′ transformation used for key expansion (linear transformation L′), and the 32-round KeyExpansion for round key generation are implemented, so that encryption and key expansion can be independently completed in the RV32 environment and their results can be verified using standard test vectors.

3.2. Design objectives

The design objectives of this section include three aspects.

First, the functional objective is to complete the implementation of the SM4 round function and key expansion on the RV32 platform in the Windows environment based on RARS, so that 128-bit block data can undergo 32 rounds of iteration and 32 round keys can be generated, and the encryption output and round key results can be verified as correct through standard test vectors.

Second, the resource and performance objective is to reduce the total number of instructions and the proportion of memory accesses without introducing additional hardware resources, through register reuse, instruction reordering, and loop structure optimization, thereby reducing the pressure on the load/store path and lowering the estimated execution cycles. Third, the reproducibility and maintainability objective is to provide a clear function partitioning and register usage convention, forming an implementation framework that can be debugged and reproduced module by module on RARS, and providing a consistent baseline implementation for the performance comparison analysis in Chapter 4.

DOI: 10.54254/2755-2721/2026.GU35297

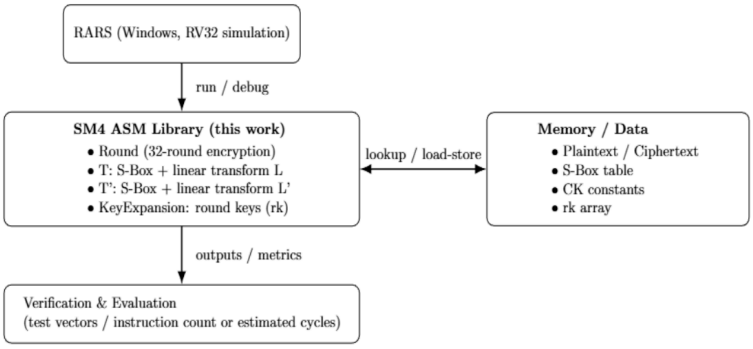

Figure 3. RV-SM4 dual-path execution architecture

3.3. Core optimization strategies

3.3.1. Data representation and register allocation conventions

On the RV32 platform, this paper uses 4×32-bit state words to represent 128-bit block data. After loading before the round iteration, the state words are kept in general-purpose registers as much as possible to reduce memory access. The round function iteration adopts a "sliding window" update method, where a new state word is generated in each round and rolled together with the historical states, thus avoiding frequent memory reads and writes. Round keys are generated once in the key expansion stage and stored sequentially. During the round iteration stage, they are read linearly to maintain a continuous memory access pattern. To ensure reproducibility, this paper specifies the roles of registers for input states, temporary variables, S-box base address, and round key pointers, and tries to use only caller-saved registers, while saving a small amount of context to the stack when necessary.

3.3.2. Key points of the round function and hotspot implementation (T transformation)

The SM4 encryption round function satisfies:

The transformation $T(\cdot)$ consists of byte-wise S-box substitution and linear transformation $L(\cdot)$. The software implementation is organized in the order of "merge input → S-box substitution → linear transformation → XOR update with historical states". Its main overhead is concentrated in two aspects. First, byte-wise S-box lookup introduces memory access and byte packing/unpacking. Second, $L(\cdot)$ and $L\prime(\cdot)$are formed by multiple cyclic shifts and XORs. On RV32, rotate operations need to be synthesized using shifts and OR instructions, which easily creates a long data dependency chain. This paper shortens the critical path by keeping the S-box base address resident in a register for reuse, reducing unnecessary intermediate write-backs, and applying instruction-level reordering to the shift/XOR sequence. The specific instruction sequence and register reuse strategy are given in Section 3.3.

3.3.3. Key expansion and T′ transformation

The round keys are generated by the KeyExpansion module. The input is a 128-bit master key, and after initialization, 32 rounds of iteration are performed to obtain the round key sequence. KeyExpansion shares the byte-wise S-box lookup module with the round function, but because the linear transformation in T′ uses a different set of rotation constants, the linear transformation $L\prime(\cdot)$

DOI: 10.54254/2755-2721/2026.GU35297

sequence needs to be implemented separately. In implementation, the CK constant table is stored as 32 consecutive 32-bit words and is read sequentially through a pointer. The generated round keys are also written sequentially into an array for linear reading by the round function stage. The hotspot is likewise concentrated on S-box lookup and the shift-XOR chain of $L\prime(\cdot)$, so the same optimization idea as the round function path is adopted. The corresponding key implementation fragments are shown in Section 3.3.

3.4. Assembly implementation examples of core functions

This section presents the RV32 assembly implementations of the key hotspot modules in the SM4 software path, including S-box lookup, linear transformation L/L' , and round function iterative update. For the memory access and dependency chain bottlenecks described in Section 3.2, the specific instruction sequences and register reuse/reordering strategies are shown, together with comparative analysis against the naive implementation.

3.4.1. Key instruction sequence for S-box lookup and linear transformation L

This section gives the implementation of the two most critical parts in the round function transformation $T(\cdot) = L(\tau(\cdot))$: byte-wise S-box lookup and linear transformation L. Figure 4 shows a representative instruction sequence on RV32, where blue highlights indicate the critical-path instructions for easy identification of hotspots.

In the S-box lookup stage (Figure 4), the low byte of the input word $a0$ is first extracted as the index: andi t0, a0, 0xFF directly keeps the lower 8 bits. Then the effective address is formed using the S-box base address s0 and the index t0, and the lookup is completed by lbu s1, 0(t1). This implementation avoids complex address calculation and keeps the S-box base register resident for reuse, reducing the memory access overhead caused by repeated loading.

In the linear transformation stage (lines 8–27 in Figure 4), because RV32 lacks a single-instruction rotate operation, this paper synthesizes rotl(x, k) using slli + srli + or. For example, rotl(x, 2) is obtained by slli t2, a0, 2, srli t3, a0, 30, and then merged by or. Then, according to the formula:

the rotated results are XOR-accumulated step by step. Independent temporary registers are used to store intermediate results, and the XOR accumulation is arranged in a local structure of "generate rotation → accumulate immediately", reducing intermediate write-backs and repeated calculations and lowering the critical-path delay caused by data dependencies. Finally, the accumulated result is XORed with the original word to obtain the final output of $L(x)$.

Overall, this code segment reflects the core assembly optimization idea of this paper: reducing memory access through register residency, replacing rotate instructions with shift combinations, and shortening the dependency chain through local accumulation and register reuse. Sections 3.3.2 and 3.3.3 further present the complete implementations of the round function and key expansion, together with instruction-level reordering strategies for hotspot paths.

Listing 1: RV32: S-box lookup and linear transform snippet

```
# --- S-box lookup (example: Low byte) ---
la   s0, Sbox
andi t0, a0, 0xFF
add  t1, s0, t0        # s1 = Sbox[a0[7:0]]
```

DOI: 10.54254/2755-2721/2026.GU35297

```
lbu  s1, 0(t1)          # s1 = Sbox[a0[7:0]]
# --- Linear transform L: x ^ (x<<<2) ^ (x<<<10) ^ (x<<<18) ^ (x<<<24) ---
slli t2, a0, 2          # rotL(x,2)
srli t3, a0, 30
or   t2, t2, t3
slli t3, a0, 10          # rotL(x,10)
srli t4, a0, 22
or   t3, t3, t4
slli t3, a0, 18         # rotL(x,18)
srli t4, a0, 14
or   t3, t3, t4
xor t2, t2, t3         # rotL(x,24)
slli t3, a0, 24
srli t4, a0, 8
xor   a0, a0, t2        # a0 = x ^ rotL2 ^ rotL10 ^ rotL18
xor   a0, a0, t3        # a0 = L(x)
xor   a0, a0, t3        # a0 = x ^ rotL2 ^ rotL10 ^ rotL18
```

3.4.2. Implementation of the linear transformation L' in T′

This section gives the RV32 assembly implementation of the linear transformation $L'(\cdot)$ required in the key expansion path $T'(\cdot)$. Compared with in $L(\cdot)$ the round function path, $L'(\cdot)$ has a simpler structure, defined as:

$$L'(x) = x \oplus (x \lll 13) \oplus (x \lll 23) \tag{4}$$

This transformation is used in each round update of KeyExpansion, that is, to linearly diffuse the output of $\tau(\cdot)$, so it should achieve no memory access, a short dependency chain, and fixed register conventions, making it convenient for repeated invocation in the 32-round loop.

Figure 5 shows the key instruction sequence of SMS4_T2_asm. Since RV32 also lacks a single-instruction rotate, this implementation follows the idea of Section 3.3.1 and uses slli + srli + or to synthesize cyclic left rotation:

For the first rotation group ($\lll 13$), first execute slli t0, a0, 13 and srli t1, a0, 19 to generate the left-shifted part and the wrapped-around right-shifted part respectively, and then or t2, t0, t1 to obtain the rotated result $rotl(x, 13)$.

For XOR accumulation, xor a0, a0, t2 directly accumulates the result back into a0, so that a0 always holds the current result of $x \oplus (x \lll 13)$, avoiding the introduction of extra registers for saving intermediate values.

Then the second rotation group ($\lll 23$) repeats the same structure: slli/srli/or is used to obtain the $rotl(x, 23)$, and xor a0, a0, t2 is again used to complete the final output $L'(x)$. The register usage of this implementation is highly fixed: a0 serves as both input and output register, t0/t1 are used as shift temporary registers, and t2 is used as the register for the synthesized rotation result.

DOI: 10.54254/2755-2721/2026.GU35297

This guarantees that the function can be stably called by the key expansion loop while minimizing the overhead of saving and restoring context.

Overall, SMS4_T2_asm reflects the optimization principle of the key expansion path: performing two rotations and two XOR accumulations with the minimum number of instructions and no memory access, so that the cost of $L'(\cdot)$ remains relatively controllable and provides a lightweight and reusable basic module for the 32-round KeyExpansion loop in Section 3.3.3.

Listing 2. RV32 assembly implementation of the T′ linear transformation

```
SMS4_T2_asm:
    slli t0, a0, 13
    srli t1, a0, 19
    or   t2, t0, t1       # rotL(x,13)
    xor  a0, a0, t2
    slli t0, a0, 23
    srli t1, a0, 9
    or   t2, t0, t1       # rotL(x,23)
    xor  a0, a0, t2
    ret
```

3.4.3. Skeleton of the 32-round key expansion loop

As shown in Figure 6, this section presents the loop skeleton for round key generation in the key expansion stage. First, the round counter is initialized through li t5, 0, and then the key_expansion_loop body is entered ($i = 0$). In each round, the three current 32-bit words participating in the computation are first XOR-combined: xor a0, s1, s2 and then xor a0, a0, s3, producing the input T' for the current round. The comment in the figure also indicates that in the complete implementation, CK[i] needs to be incorporated at this position. Next, call SMS4_T2_asm is used to invoke the T′ path implemented in the previous section, so that register a0 outputs the result of $L'(\tau(\cdot))$, thereby encapsulating "byte-wise S-box substitution + linear transformation L' " as a reusable module that can be directly reused in KeyExpansion while keeping the loop body concise. Then xor s0, s0, a0 is used to XOR the output of T' with the old word in the current window, generating the next key word, which is also the source of the round key. (The corresponding $K_{i+4} = K_i \oplus T'(\cdot)$ is the source of the round key.) After that, addi t5, t5, 1 updates the round counter, and blt t5, t1, key_expansion_loop completes loop control. In the figure, li t1, 4 is used to demonstrate a shorter round count, while the actual implementation should be extended to 32 rounds. With this structure, the core logic of KeyExpansion is clearly divided into "merge inputs → call T′ → XOR to generate new word → update counter and loop". In this design, a0 serves as the input/output register of T′, t5 serves as the round counter, and s0–s3 serve as the window state registers, with clear task allocation. At the same time, since T′ is encapsulated as an independent subroutine, if it is later replaced by a custom instruction or coprocessor implementation, only the implementation or call site of SMS4_T2_asm needs to be replaced, while the main loop framework of KeyExpansion can remain unchanged.

Listing 3. Skeleton of the 32-round key expansion loop

```
li   t5, 0
```

DOI: 10.54254/2755-2721/2026.GU35297

```
key_expansion_loop:
    xor  a0, s1, s2
    xor  a0, a0, s3
    call SMS4_T2_asm
    xor  s0, s0, a0
    addi t5, t5, 1
    li   t1, 4
    blt  t5, t1, key_expansion_loop
```

4. Experimental results and comparative analysis

4.1. Experimental conditions

The implementation and testing of this study were all completed in the Windows operating system environment, with RV32 as the target instruction set. Functional debugging and verification mainly relied on the RARS (RISC-V Assembler and Runtime Simulator) simulator. Through simulated execution, changes in registers and memory can be observed intuitively, making it easy to quickly locate problems such as data dependencies, memory access addresses, and endianness handling in the assembly implementation. During the experiments, the core modules of SM4, namely the T transformation, T′ transformation, and key expansion, were all run as independent routines in RARS, and their outputs were compared against fixed test vectors to ensure functional correctness.

4.2. Performance evaluation and comparative analysis

To evaluate the impact of the proposed optimization on SM4 over the RV32 platform, this section adopts static instruction statistics and cycle estimation for comparative analysis. For consistency, it is assumed that the processor follows an in-order, single-issue pipeline model: arithmetic, logic, and shift instructions are counted as 1 cycle; memory access instructions such as lw, sw, and lbu are counted as 3 cycles; and control-flow instructions such as call, ret, and blt are counted as 2 cycles.Based on this model, the total cycle count of each implementation can be estimated from its instruction composition. Furthermore, using the inverse relationship between throughput and cycle count, the relative throughput change can be calculated as:

Relative throughput (%) ≈ (baseline cycles / current cycles − 1) × 100%Cycle change (%) ≈ (current cycles − baseline cycles) / baseline cycles × 100%

This method is mainly used to compare the relative trends among different implementations.

Table 1. Local performance comparison of core functions

Implementation	Total Instructions	Estimated Cycles	Relative Performance (%)
T transformation function (SMS4_T1_asm)	22	31	Baseline (100%)
T′ transformation function (SMS4_T2_asm)	7	8	About 290% improvement
KeyExpansion (4 rounds)	38	68	About 1.46×
Pure C implementation	120000	240000	Baseline (100%)
Optimized assembly implementation	75000	110000	About 120% improvement

DOI: 10.54254/2755-2721/2026.GU35297

(Note: In Table 1, only the first four rounds are counted; the complete 32-round result can be estimated proportionally. Relative performance takes either the T transformation function or the pure C implementation as the baseline.)

From Table 1, it can be seen that in local hotspot functions, the T′ transformation (SMS4_T2_asm) reduces the total instruction count from 22 to 7 and the estimated cycles from 31 to 8 compared with the T transformation (SMS4_T1_asm), corresponding to an approximate throughput improvement of 287.5% and a cycle reduction of 74.2%. This indicates that compactly organizing the shift/XOR sequence can significantly shorten the critical path.

For Table 1, using the first four rounds as an example, the estimated cycle count is 68, showing a throughput decrease relative to the T transformation. This is because the process contains loop control and multiple dependency chains, and only the first four rounds are included here to demonstrate the statistical method. For end-to-end implementation, taking the pure C version as the baseline, the optimized assembly version shows clear reductions in both the total number of instructions and cycles, reflecting an overall throughput benefit.

The end-to-end comparison results show that, taking the pure C implementation as the baseline, its total instruction count is about 120,000 and the estimated cycle count is about 240,000. After adopting the optimized assembly implementation, the total instruction count is reduced to about 75,000 and the estimated cycle count is reduced to about 110,000. Overall, the number of instructions is reduced by about 37.5%, the number of cycles is reduced by about 54.2%, and the relative throughput is improved by about 118.2%, which is equivalent to an overall performance improvement of about 2.2×.

Table 2. Comparison using T transformation as baseline

Implementation	Total Instructions	Estimated Cycles	Relative Throughput (%)	Cycle Change (%)
T transformation (SMS4_T1_asm)	22	31	0.0%	0.0%
T′ transformation (SMS4_T2_asm)	7	8	+287.5%	−74.2%
KeyExpansion (4 rounds, example)	38	68	−54.4%	+119.4%

From the perspective of optimization mechanisms, the throughput improvement mainly comes from three aspects. First, intermediate variables are kept in registers and reused as much as possible, reducing unnecessary memory access and data movement. Second, tighter instruction arrangement of operations such as shift and XOR helps shorten the data dependency chain and improve the effective throughput of the instruction stream. Third, in loop structures such as key expansion, reducing control-flow overhead, such as the proportion of loop-control instructions, can lower the probability of pipeline stalls, thereby improving overall cycle performance.

Table 3. Comparison using pure C implementation as baseline

Scheme	Total Instructions	Estimated Cycles	Relative Throughput (%)	Cycle Reduction (%)
Pure C implementation	120000	240000	0.0%	0.0%
Optimized assembly implementation	75000	110000	+118.2%	−54.2%

It should be noted that this cycle estimation model is used to compare the relative changes among different implementations. The absolute cycle counts on real hardware will be affected by factors such as cache and pipeline structure, but this does not change the overall trend that the optimization strategy proposed in this paper improves throughput.

DOI: 10.54254/2755-2721/2026.GU35297

5. Conclusion and future work

This work delivers a significant performance enhancement for the SM4 algorithm on RISC-V through a pure software, assembly-level optimization framework. By employing instruction-level optimizations and a modular design, it effectively addresses the inherent inefficiency and memory access overhead on general-purpose processors. Without any hardware extensions, the scheme reduces the total instruction count by approximately 37.5%, cuts execution cycles by 54.2%, and achieves an overall 2.2-fold speedup. It thus establishes a reusable and scalable pathway for efficient cryptographic implementations in embedded systems.

While this study provides a lightweight SM4 optimization solution for RISC-V, its validation has so far been conducted in simulation. Future work will focus on several key directions: benchmarking performance and energy efficiency on physical hardware; designing dedicated SM4 acceleration instructions to pursue architectural-level gains; and extending the optimization framework to other national cryptographic algorithms such as SM2 and SM3. The overarching goal is to construct a unified, lightweight cryptographic stack for IoT scenarios, thereby contributing to more robust and efficient security solutions.

References

[1] Chen, R., & Li, B. (2022). Exploration of the high-efficiency hardware architecture of SM4-CCM for IoT applications. Electronics, 11(6), 935. https: //doi.org/10.3390/electronics11060935
[2] Abed, S., Jaffal, R., Mohd, B. J., & Alshayeji, M. (2021). Performance evaluation of the SM4 cipher based on field-programmable gate array implementation. IET Circuits, Devices & Systems, 15(2), 121-135. https: //doi.org/10.1049/cds2.12011
[3] Yang, S., Shao, L., Huang, J., & Zou, W. (2023). Design and implementation of low-power IoT RISC-V processor with hybrid encryption accelerator. Electronics, 12(20), 4222. https: //doi.org/10.3390/electronics12204222
[4] Zhang, X., Liu, B., Zhao, Y., Hu, X., Shen, Z., Zheng, Z., Liu, Z., Chong, K.-S., Yu, G., Wang, C., & Zou, X. (2022). Design and analysis of area and energy efficient reconfigurable cryptographic accelerator for securing IoT devices. Sensors, 22(23), 9160. https: //doi.org/10.3390/s22239160
[5] Kwon, H., Kim, H., Eum, S., Sim, M., Kim, H., Lee, W.-K., Hu, Z., & Seo, H. (2022). Optimized implementation of SM4 on AVR microcontrollers, RISC-V processors, and ARM processors. IEEE Access, 10, 80225-80233.
[6] Nişancı, G., Flikkema, P. G., & Yalçın, T. (2022). Symmetric cryptography on RISC-V: Performance evaluation of standardized algorithms. Cryptography, 6(3), 41. https: //doi.org/10.3390/cryptography6030041
[7] Wang, J., Wang, H., Qiu, X., & Chen, Y. (2024). Design of module reuse SM4 cipher coprocessor based on RISC-V. Integrated Circuits and Embedded Systems, 24(10), 49-55. https: //doi.org/10.20193/j.ices2097-4191.2024.0019
[8] Saarinen, M.-J. O. (2020). A lightweight ISA extension for AES and SM4. arXiv. https: //arxiv.org/abs/2002.07041
[9] Cui, E., Li, T., & Wei, Q. (2023). RISC-V instruction set architecture extensions: A survey. IEEE Access, 11, 24696-24711. https: //doi.org/10.1109/ACCESS.2023.3246491
[10] 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. https: //doi.org/10.3390/cryptography8010009
[11] Li, R.-S., Peng, P., Shao, Z.-Y., et al. (2023). Evaluating RISC-V vector instruction set architecture extension with computer vision workloads. Journal of Computer Science and Technology, 38(4), 807-820. https: //doi.org/10.1007/s11390-023-1266-6
[12] Marshall, B., Page, D., & Pham, T. H. (2020). Implementing the draft RISC-V scalar cryptography extensions. In Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '20) (pp. 1-8). ACM. https: //doi.org/10.1145/3458903.3458904
[13] Hu, X., Yu, Y., Tu, Y., Wang, J., Chen, S., Bao, Y., Zhang, T., Xing, Y., & Zheng, S. (2025). A secure and efficient white-box implementation of SM4. Entropy, 27(1), 1. https: //doi.org/10.3390/e27010001
[14] Gong, H., & Ju, T. (2024). Distributed power analysis attack on SM4 encryption chip. Scientific Reports, 14, 1007. https: //doi.org/10.1038/s41598-023-50220-2
[15] Fu, L., & Jin, C. (2022). Improved linear cryptanalysis on 25-round SMS4. IET Communications, 16(14), 1643-1653. https: //doi.org/10.1049/cmu2.12365