Low-latency implementation of the GIFT cipher on RISC-V architectures

Gheorghe Pojoga[1] and Kostas Papagiannopoulos[2]

[1] University of Amsterdam, The Netherlands, gheorghe.pojoga@os3.nl
[2] University of Amsterdam, The Netherlands, k.papagiannopoulos@uva.nl

Abstract. Lightweight cryptography is a viable solution for constrained computational environments that require a secure communication channel. To standardize lightweight primitives, NIST has published a call for algorithms that address needs like compactness, low-latency, low-power/energy, etc. Among the candidates, the GIFT family of block ciphers was utilized in various NIST candidates due to its high-security margin and small gate footprint. As a result of their hardware-oriented design, software implementations of GIFT require additional optimization techniques such as bitslicing and fixslicing to achieve optimal performance. Even though the performance of these methods has been assessed for several ISA families such as x86 and ARM, there is currently a lack of data with regards to their acceleration capabilities for RISC-V. Since this ISA is an important element of the growing open-hardware movement, our goal is to address this knowledge gap. Therefore, we have developed several assembly implementations for both GIFT-64 and GIFT-128, using the RV32I ISA, and performed a quantitative assessment of their performance using a physical board i.e., Hifive1 Rev B. Our study has shown that by using bitslicing the number of clock cycles can be reduced by 69.33% for GIFT-64 and 71.38% for GIFT-128, compared to a naive assembly implementation, while fixslicing decreases the number of clock cycles by 85.7% (GIFT-64) and 81.28% (GIFT-128). Nonetheless, the preferred technique is fixslicing with key pre-computation, which can achieve a reduction of 88.69% (GIFT-64) and 95.05% (GIFT-128), while maintaining relatively low memory requirements of 938 bytes (GIFT-64) and 1388 bytes (GIFT-128), respectively.

Keywords: GIFT, RISC-V, implementation, bitslicing, fixslicing

1 Introduction

Conventional cryptographic algorithms, such as AES-128, have satisfied most of the security and privacy requirements of our society. However, due to multiple emerging areas which employ constrained computational environments e.g., the automotive industry, internet of things, sensor networks, healthcare systems, RFID tags, etc., more efficient and custom cryptographic algorithms are needed. Such environments have multiple requirements in addition to the ones for conventional cryptography e.g., low energy consumption, small code size, and low chip area. For this reason, the National Institute of Standards and Technology (NIST) has directed significant efforts toward standardizing lightweight cryptography [9, 10, 16]. In this sense, it has issued a call for submissions in 2018 for a lightweight AEAD (authenticated encryption with associated data) algorithm i.e., an algorithm that is viable for low chip area, will require low RAM and ROM usage, as well as, provide support for low energy, low power, and low latency implementations [11]. Multiple NIST applicants are based on/inspired by the GIFT family of block ciphers i.e., ESTATE, Fountain, GIFT-COFB, HyENA, LOTUS-AEAD, and LOCUS-AEAD, Simple64/Simple128, SUNDAE-GIFT, TGIF and TRIFLE [12].

GIFT [2] is a family of block ciphers that consists of GIFT-64 and GIFT-128. It is inspired by PRESENT [4], however, it is significantly smaller and faster, as well as, it is resistant against linear hulls [8] i.e., the weak point of PRESENT. GIFT has been the subject of multiple security assessments [2, 18, 13] while preserving a high-security margin. Moreover, its low computational requirements make it a promising block cipher in the context of constrained environments. However, due to its hardware-oriented design, which involves a bit-oriented permutation layer, the software implementations require non-trivial acceleration techniques to achieve viable performance. Depending on the use case, such techniques can be aimed at optimizing the encryption latency i.e., the number of clock cycles required for the encryption of a single block, or at increasing the encryption throughput i.e., the overall number of encrypted bits per clock cycle, which can be achieved by using a highly-parallelized implementation. Therefore, the design decisions for the implementation depend on the metric that is meant to be improved. The goal of this project is to optimize the encryption latency. Hence, we have used bitslicing [2] and fixslicing [1], as acceleration techniques. The performance of these techniques has been previously evaluated using ARM [1] and x86 [2] instruction set architectures (ISA), however, there is a lack of data with regards to their performance on RISC-V. Considering that the support for this ISA family is currently growing, as it is also regarded as the "Linux of the open-hardware movement", a quantitative assessment of the available acceleration techniques is required.

Contribution. This paper describes low-latency implementations of the GIFT cipher on RISC-V (RV32I), using bitslicing and fixslicing as optimization techniques. The reason we have used this ISA is that RV32I is the least complex RISC-V instruction set, except for RV32E. Hence, our implementation can be easily adapted to any other RISC-V ISA, and the results represent a lower bound for the possible optimization capabilities. Additionally, we put forward an alternative description for fixslicing, which can be directly mapped to the RISC-V architecture, as well as, an optimized matrix transposition, presented in appendix 8.1. Moreover, assembly implementations of the optimization techniques, in addition to a naive (Baseline) implementation of cipher, have been developed. Our performance assessment has shown that the preferred implementation is fixslicing in combination with key pre-computation i.e., the number of clock cycles has been reduced by 88.69% and 95.05% for GIFT-64 and GIFT-128, compared to the baseline implementation. The source code of the implementation can be found at `https://github.com/gra2p/gift_risc-v`.

2 GIFT CIPHER

GIFT is a family of block ciphers comprised of GIFT-64 and GIFT-128. Both ciphers require a key of 128 bits and block sizes of 64 and 128 bits, respectively. They are based on a substitution-permutation network (SPN) with 28 rounds for GIFT-64 and 40 rounds for GIFT-128. Each round consists of four layers: `SubCells`, `PermBits`, `AddRoundKey`, `KeySchedule`.

Data Representation. Each round i receives as input the cipher state S^{i-1}(64,128-bits), the key state K^i (128-bits) and the round constant C^i (6-bits), and produces S^i, K^{i+1} and C^{i+1}. $S^0 = b_{n-1}b_{n-2}b_{n-3}\ ...b_0$ is initialized with the plaintext, where $n = 64, 128$ and b_0 is the least significant bit of the plaintext. Additionally, S^i can be represented as $S^i = w_{n/4-1}w_{n/4-2}w_{n/4-3}...w_0$, where $w_i = b_{4i+3}b_{4i+2}b_{4i+1}b_{4i}$. Similarly, $K^1 = k_{127}k_{126}k_{125}...k_0$, where k_0 is the least significant bit of the encryption key. Alternatively, the key state can be represented as $K^1 = k_7k_6k_5...k_0$, where k_j is a 16-bit block. Moreover, the round constant for round i is defined as $C^i = c_5c_4c_3c_2c_1c_0$, where c_0 is the least significant bit. The constant for round 1 is initialized to 1 i.e., $C^1 = 1$.

SubCells. The substitution layer is the same for both GIFT-64 and GIFT-128, and is based on an invertible 4-bit SBox, GS, presented in Table 1, i.e.,

$$\forall i \in [0, n/4] : w_i \leftarrow GS(w_i), \text{ where } n = 64, 128$$

Table 1: Specification of the GIFT SBox in hexadecimal notation.

w	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
GS(w)	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

PermBits. The permutation layer maps the bits from position i to position $P(i)$ i.e.,

$$\forall i \in [0, n) : b_{P(i)} \leftarrow b_i, \text{ where } n = 64, 128.$$

This layer is different for GIFT-64 and GIFT-128 i.e.,

$$P_{64}(i) = 4\lfloor \tfrac{i}{16} \rfloor + 16((3\lfloor \tfrac{1 \bmod 16}{4} \rfloor + (i \bmod 4)) \bmod 4) + (i \bmod 4)$$

$$P_{128}(i) = 4\lfloor \tfrac{i}{16} \rfloor + 32((3\lfloor \tfrac{1 \bmod 16}{4} \rfloor + (i \bmod 4)) \bmod 4) + (i \bmod 4)$$

AddRoundKey. This layer adds the round key K^i and the round constant C^i to the cipher state S^i. For GIFT-64,

$$\forall i \in [0, 16) : b_{4i+1} \leftarrow b_{4i+1} \oplus u_i,\ b_{4i} \leftarrow b_{4i} \oplus v_i,$$
$$\text{where } u \leftarrow k_1,\ v \leftarrow k_0$$

For GIFT-128 the round key is added as follows,

$$\forall i \in [0, 32) : b_{4i+2} \leftarrow b_{4i+2} \oplus u_i,\ b_{4i+1} \leftarrow b_{4i+1} \oplus v_i,$$
$$\text{where } u \leftarrow k_5 || k_4,\ v \leftarrow k_1 || k_0$$

The round constant is the same for both GIFT-64 and GIFT-128, and it is applied as follows,

$$\forall i \in [0, 5] : b_{4i+3} \leftarrow b_{4i+3} \oplus c_i,$$
$$b_{n-1} \leftarrow b_{n-1} \oplus 1, \text{ where } n = 64, 128$$

KeySchedule. This layer is responsible for updating the round key and the round constant and it is the same for both GIFT-64 and GIFT-128. The key state is updated as follows,

$$k_7 || k_6 || ... || k_1 || k_0 \leftarrow k_1 \ggg 2 || k_0 \ggg 12 || ... || k_3 || k_2$$

The new round constant is computed using the following mapping:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1)$$

3 Optimization techniques

3.1 Bitslicing

Bitslicing is an alternative representation for GIFT-64 and GIFT-128, which uses the Single Instruction/Multiple Data (SIMD) paradigm to achieve improved performance in software implementations [3]. This approach preserves the same layered structure of the, 28 or 40, rounds. The steps `SubCells` and `AddRoundKey` can leverage this representation, in order to reduce the number of the operations. Conversely, the steps `PermBits` and `KeySchedule` remain unchanged. We have used the same approach to bitslicing, as presented by Banik et al. [2].

Data Representation. Similarly to the classical representation, each round i receives as input the cipher state S^{i-1} (64,128-bits), the key state K^i (128-bits) and round constant C^i (6-bits). K^i and C^i preserve the same representation as in the classical approach, while S is defined as follows,

$$S = \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} b_{n-4} & \dots & b_8 & b_4 & b_0 \\ b_{n-3} & \dots & b_9 & b_5 & b_1 \\ b_{n-2} & \dots & b_{10} & b_6 & b_2 \\ b_{n-1} & \dots & b_{11} & b_7 & b_3 \end{bmatrix}$$

where $n = 64, 128$.

SubCells. This representation of the cipher state allows the definition of the substitution layer through a series of logical operations with multiplicative complexity of 4:

$$\begin{aligned} S_1 &\leftarrow S_1 \oplus (S_0 \wedge S_2) \\ T &\leftarrow S_0 \oplus (S_1 \wedge S_3) \\ S_2 &\leftarrow S_2 \oplus (T \vee S_1) \\ S_0 &\leftarrow S_3 \oplus S_2 \\ S_1 &\leftarrow S_1 \oplus S_0 \\ S_0 &\leftarrow \neg S_0 \\ S_2 &\leftarrow S_2 \oplus (T \wedge S_1) \\ S_3 &\leftarrow T \end{aligned}$$

The benefit of this definition is that `SubCells` is applied to all nibbles in parallel i.e., it is not necessary to sequentially match and replace each nibble.

AddRoundKey. The bitsliced representation prevents the necessity of performing per-bit application of the round key and round constant. For GIFT-64 the round key is added as follows,

$$\begin{gathered} S_1 \leftarrow S_1 \oplus u \\ S_0 \leftarrow S_0 \oplus v, \\ \text{where } u \leftarrow k_1,\ v \leftarrow k_0 \end{gathered}$$

Similarly, in the case of GIFT-128 the round key is applied as,

$$\begin{gathered} S_2 \leftarrow S_2 \oplus u \\ S_1 \leftarrow S_1 \oplus v, \\ \text{where } u \leftarrow k_5 \| k_4,\ v \leftarrow k_1 \| k_0 \end{gathered}$$

The round constant is applied to the cipher state of both GIFT-64 and GIFT-128 as follows.

$$\begin{gathered} S_3 \leftarrow S_3 \oplus T, \\ \text{where } T = (1 \ll n) \oplus C,\ \text{n=15,31 (GIFT-64, GIFT-128)} \end{gathered}$$

3.2 Fixslicing

The bitsliced representation significantly decreases the number of operations required for applying `SubCells` and `AddRoundKey`, however, the effect on `PermBits` is insignificant, even though this layer involves multiple per-bit operations, which result in performance degradation for software implementations. Hence, a new acceleration technique, called Fixslicing [1], has been proposed. Fixslicing preserves the same definition for `SubCells` as in the case of bitslicing, however, the layers `PermBits`, `KeySchedule`, and `AddRoundKey` for GIFT-128, require modifications. In order to accommodate this technique to a RISC-V context, we have deviated from the original specification of Fixslicing [1]. Below we present our modified representation of the slices, as well as, the adapted layers `PermBits`, `KeySchedule` and `AddRoundKey`.

Data Representation. Similarly to bitslicing, the cipher state S is divided into four slices. However, each slice (S_i) is represented as a 4×4 matrix (in the case of GIFT-64) or a

4×8 matrix (in the case of GIFT-128). In contrast to the original Fixsliced representation [1], we have used the Big-Endian notation to represent each slice i.e.,

$$S_i = \begin{bmatrix} b_{i+n-4+3n} & b_{i+n-8+3n} & \dots & b_{i+4+2n} & b_{i+2n} \\ b_{i+n-4+2n} & b_{i+n-8+2n} & \dots & b_{i+4+2n} & b_{i+2n} \\ b_{i+n-4+n} & b_{i+n-8+n} & \dots & b_{i+4+n} & b_{i+n} \\ b_{i+n-4} & b_{i+n-8} & \dots & b_{i+4} & b_i \end{bmatrix}$$
$$\text{where } i \in \{0,1,2,3\}, \text{ and } n \in \{16,32\}$$

This alternative representation requires the modification of the `AlignBits` and `KeySchedule` layers. Nonetheless, the benefit is that it offers a more natural mapping between the plaintext bitstring and the internal representation of the slices. Additionally, each GIFT-128 slice is represented as a 4×8 matrix, instead of two 4×4 matrices in the original definition. The reason is that RV32I does not have an inline barrel shifter instruction, which has been used in the ARM implementation [1] to increase performance. Therefore, in our context, splitting a GIFT-128 slice into two sub-slices adds additional complexity without any performance gains. The new representation requires modification of several layers, as presented in the next subsections i.e., **GIFT-64** and **GIFT-128**.

GIFT-64. Fixslicing is based on the observation that after several repeated applications of the permutation layer, the bits return to their initial position. In the case of GIFT-64, this occurs, for all slices, every four rounds. Therefore, instead of performing the bit permutations, one of the slices is fixed to the same configuration, while the other slices are modified such that the `SubCells` mapping can be applied. The new layer replaces `PermBits`, and in order to avoid confusion we will refer to it as `AlignBits`.

AlignBits. Since in the case of GIFT-64 all slices return to the initial position after four permutations, any slice can be chosen as the fixed slice. We have decided to fix S_0. Moreover, for each of the four sub-rounds, the definition of `AlignBits` is different. Therefore, the fixsliced implementation of GIFT-64 consists of 7 rounds, each containing 4 sub-rounds i.e.,

`round 1:` S_i is rotated by i columns to the right.
`round 2:` S_i is rotated by i rows downwards.
`round 3:` S_i is rotated by i columns to the left.
`round 4:` S_i is rotated by i rows upwards.

KeySchedule. Due to the division of each round into four sub-rounds with different orders of the cipher state bits, we split the 16-bit blocks of the key state into pairs, and their bits are permuted to match the cipher state, i.e.,

$$k_0, k_1 = \begin{bmatrix} b_7 & b_{11} & b_{15} & b_3 \\ b_6 & b_{10} & b_{14} & b_2 \\ b_5 & b_9 & b_{13} & b_1 \\ b_4 & b_8 & b_{12} & b_0 \end{bmatrix} \qquad k_2, k_3 = \begin{bmatrix} b_5 & b_6 & b_7 & b_4 \\ b_9 & b_{10} & b_{11} & b_8 \\ b_{13} & b_{14} & b_{15} & b_{12} \\ b_1 & b_2 & b_3 & b_0 \end{bmatrix}$$

$$k_4, k_5 = \begin{bmatrix} b_{13} & b_9 & b_5 & b_1 \\ b_{14} & b_{10} & b_6 & b_2 \\ b_{15} & b_{11} & b_7 & b_3 \\ b_{12} & b_8 & b_4 & b_0 \end{bmatrix} \qquad k_6, k_7 = \begin{bmatrix} b_{15} & b_{14} & b_{13} & b_{12} \\ b_{11} & b_{10} & b_9 & b_8 \\ b_7 & b_6 & b_5 & b_4 \\ b_3 & b_2 & b_1 & b_0 \end{bmatrix}$$

Similarly to `AlignBits`, the `KeySchedule` must be different for each sub-round in order to update each pair of key state block i.e., a sub-round i uses the mapping $\texttt{KeySchedule}^i$. Moreover, each pair is used every 4 sub-rounds. Hence, a given round i will use the pair k_{2i-2}, k_{2i-1}.

`round 1:` For this sub-round the pair k_0, k_1 is used. k_1 is updated by rotating the entire matrix downwards by 2 rows, and the first 2 rows of the resulting matrix to the left by 1 column. k_0 is updated by rotating the entire matrix to the right by 1 column.
`round 2:` For sub-round 2 the pair k_3, k_2 is used. k_3 is updated by rotating the entire matrix to the right by 2 columns, and the inner columns of the resulting matrix by 1 row upwards. k_2 is updated by rotating the entire matrix downwards by 1 row.
`round 3:` For sub-round 3 the pair k_5, k_4 is used. k_5 is updated by rotating the entire matrix downwards by 2 rows, and the 2 inner rows of the resulting matrix by 1 column to the right. k_4 is updated by rotating the entire matrix to the left by 1 column.
`round 4:` For sub-round 4 the pair k_7, k_6 is used. k_7 is updated by rotating the entire matrix to the right by 2 columns, and the first 2 columns of the resulting matrix downwards by 1 row. k_6 is updated by rotating the entire matrix upwards by 1 row.

GIFT-128. In contrast to GIFT-64, slices do not return to the initial position after 4 applications of `PermBits` in the case of GIFT-128 i.e., Slice 0 takes 31 rounds, Slice 1 takes 10 rounds, Slice 2 takes 31 rounds and Slice 3 takes 5 rounds. Therefore, the Slice 3 is fixed, to minimize the number of sub-rounds required. Hence, the fixsliced implementation of GIFT-128 consists of 8 rounds, which contain 5 sub-rounds. Similarly to fixslicing for GIFT-64, the bitsliced definition of `SubCells` is reused, as well as, `PermBits` is substituted with `AlignBits`. Conversely, the definition of `KeySchedule` is not modified i.e., the bitsliced version is used, therefore, an additional sublayer is added to `AddRoundKey` to map the bitsliced key state to the fixsliced representation. This difference between fixslicing for GIFT-64 and GIFT-128 is due to a significant performance overhead of a potential modification of the `KeySchedule` since a trivial alignment of the key state to the cipher state does not exist. Hence, the usage of an `AddRoundKey` sublayer is preferred due to a lower performance penalty.

AlignBits. The Slice 3 is fixed, since it requires the smallest amount of rounds to return to the initial position i.e., 5. Similarly to fixslicing for GIFT-64, each of the 5 sub-rounds uses a different definition of `AlignBits` i.e.,

`round 1:` Each 4×4 half of S_i is rotated independently by i columns to the right.
`round 2:` The slices S_0, S_1 and S_2 must be modified as follows:

S_0: The slice must be rotated by 4 columns to the right. The rows $0 \leftrightarrow 1$, as well as, $2 \leftrightarrow 3$ of the resulting first half must be swapped.
S_1: The rows $0 \leftrightarrow 1$, as well as, $2 \leftrightarrow 3$ of the slice must be swapped.
S_2: The slice must be rotated by 4 columns to the right. The rows $0 \leftrightarrow 1$, as well as, $2 \leftrightarrow 3$ of the resulting second half must be swapped.

`round 3:` The slices S_0, S_1 and S_2 must be modified as follows :

S_0: The slice must be rotated 2 rows downwards. The columns of the first two rows must be swapped as follows : $0 \leftrightarrow 1$, $2 \leftrightarrow 3$, $4 \leftrightarrow 5$, $6 \leftrightarrow 7$
S_1: The following columns must be swapped : $0 \leftrightarrow 1$, $2 \leftrightarrow 3$, $4 \leftrightarrow 5$, $6 \leftrightarrow 7$.
S_2: The slice must be rotated 2 rows downwards. The following columns of the resulting first two rows must be swapped : $0 \leftrightarrow 1$, $2 \leftrightarrow 3$, $4 \leftrightarrow 5$, $6 \leftrightarrow 7$.

`round 4:` The slices S_0, S_1 and S_2 must be rotated 2, 4, and 6 columns to the left.
`round 5:` The slices S_0, S_1 and S_2 must be rotated 1, 2 and 3 rows upwards.

AddRoundKey. This layer is implemented the same way as in the case of bitslicing since the same key scheduling algorithm is used. The only difference is that an additional sublayer (`MapKey`) is introduced in order to map the bitsliced key state to the fixsliced counterpart i.e., for each sub-round i the output of `MapKey` for $k_1 \| k_0$ and $k_5 \| k_4$ must be aligned with the resulting S_1 and S_2 of respective `AlignBits` sub-round.

4 Implementation

In addition to the optimized GIFT implementations, we have developed assembly-based baseline implementations for both GIFT-64 and GIFT-128 to assess the efficiency of these techniques in the case of RISC-V. The programs have been executed using a development board i.e., Hifive1 Rev B [15].

4.1 Integrated Memory

In order to improve the performance, we have leveraged the features provided by the development board [14] i.e., the instructions have been stored in the Instruction Tightly-Integrated Memory (ITIM), while the data has been stored in the Data Tightly-Integrated Memory (DTIM). ITIM is a volatile memory that is used for high-performance and predictable instruction delivery. Fetching an instruction from ITIM is equivalent to an instruction-cache hit. The disadvantage of ITIM is its modest size of only 8 KiB which may limit the code size. Data Tightly Integrated Memory (DTIM) is also a volatile memory, however, it is used for storing data, instead of instructions. Even though ITIM can also hold data besides instructions, the loads and stores of a core to its ITIM are less performant than the loads and stores to its DTIM. Similar to ITIM, the main disadvantage of DTIM is its size i.e., 16 KiB.

Additionally, QSPI Flash is the non-volatile memory of the chip, and it is also the default location where the programs are loaded on the chip. The disadvantage of using the QSPI Flash for storing the instructions of a running program is the unpredictable and slow instruction delivery, which results in slower execution compared to programs located entirely in ITIM. Nonetheless, an advantage of QSPI is its size i.e., 4 MB.

Therefore, in order to write high-performance programs for Hifive1 Rev B, it is important to efficiently distribute program sections across QSPI, ITIM, and DTIM. The code size of the GIFT cipher is fairly small thus all the implementations we developed as part of this project were entirely loaded onto ITIM and DTIM. To ensure this, we have not used loop unrolling as an optimization technique, since the resulting program would not fit entirely in the ITIM and lead to significant performance degradation.

4.2 Baseline

The design of the program is as close as possible to the original definition of GIFT. The cipher state is stored in 2 32-bit registers in the case of GIFT-64 and 4 registers in the case of GIFT-128. The key state for GIFT-64 is stored in 8 registers i.e., as 16-bit block, while for GIFT-128 it stored in 4 registers as 16-bit block pairs i.e., $k_7||k_6$, $k_5||k_4$, $k_3||k_2$, $k_1||k_0$. The `SubCells` layer is implemented via a lookup table, which is stored in DTIM. The permutation layer (`PermBits`) is implemented by isolating each bit, through a mask, and shifting it to the new position, as well as, storing it in the correct register. Furthermore, in order to apply the round key, k_1 and k_0 (GIFT-64), or $k_5||k_4$ and $k_1||k_0$ (GIFT-128), must be expanded to 64, or 128 bits. The key expansion is applied twice i.e., for k_1 and k_0, or $k_5||k_4$ and $k_1||k_0$.

In order to store the key state we have used 8 registers for GIFT-64 i.e., each 16-bit block is stored in a separate register, and 4 registers for GIFT-128 i.e., the key state is stored in pairs of 16-bit blocks ($k_7||k_6$, $k_5||k_4$, $k_3||k_2$ and $k_1||k_0$). The generation of the next round key (`KeySchedule`), for GIFT-64, is done by rotating the least-significant 16-bits of the respective registers, as well as, circularly moving the data among the registers holding the key state. In the case of GIFT-128, the most significant 16 bits, as well as, the least significant 16-bits of the register holding $k_1||k_0$ must be rotated independently by 2 and 12 bits respectively. Afterwards, the 32-bit key state blocks are moved circularly between the key state registers.

4.3 Bitslicing

In contrast with the data representation of the baseline, in the case of bitslicing the cipher state is stored in 4 32-bit registers, for both GIFT-64 and GIFT-128 i.e., one register per slice (S_i). The key states are stored similarly to the baseline implementations. The `SubCells` layer has been implemented using the Boolean function from the previous section, hence, avoiding the usage of a lookup table.

PermBits. Considering the definition of the permutation layer, in a bitsliced representation the bits will only be moved within the same slice. Therefore, we can apply this mapping for each slice independently. A naive implementation would use a series of bit masks and bit shifts. In the case of S_0 for GIFT-64, such an approach will require 54 instructions i.e., the bits b_0 and b_{40} do not change their position, which results in 1 operation (mask & store in the resulting register), the bits b_{44} and b_4 can be moved together, which requires 4 instructions, the remaining 12 bits have to be moved individually, and additionally we need 1 operation to store the result in s0, therefore, $1+4+12\times4+1=54$.

We improve this approach by considering the 16-bits of S_0 as a 4×4 matrix:

$$b_{60}b_{56}b_{52}b_{48}b_{44}b_{40}b_{36}b_{32}b_{28}b_{24}b_{20}b_{16}b_{12}b_8b_4b_0 \equiv \begin{bmatrix} b_{60} & b_{56} & b_{52} & b_{48} \\ b_{44} & b_{40} & b_{36} & b_{32} \\ b_{28} & b_{24} & b_{20} & b_{16} \\ b_{12} & b_8 & b_4 & b_0 \end{bmatrix}$$

Considering the matrix representation, we can compute the transpose of S_0, with 15 instructions. Afterwards, the columns 0 and 2 must be swapped to obtain the required result, which can be done in 9 instructions. Therefore, with this method, we can achieve the same result with only 24 instruction instead of 54. The same technique can be applied, with slight changes, for the other slices, as well as, for GIFT-128, to optimize the permutation layer. The full details are presented in Appendix 8.1.

AddRoundKey. An additional benefit of the bitsliced representation is the simplicity of the `AddRoundKey`, which requires only 6 instructions for both GIFT-64 and GIFT-128 i.e., two `xor` instructions for the round key, and 4 instructions for the round constant, as presented in Algorithms 1 and 2. Nonetheless, 8 cycles are consumed, because the load byte operation (`lb`) requires 3 cycles, which results in $224 = 8\times28$ cycles for GIFT-64 and $320 = 8\times40$ cycles for GIFT-128, as presented in Section 5.

Algorithm 1 `AddRoundKey (round key)`

```
1: xor s0, s0, a0        ▷ apply V (k_0 or k_1||k_0)
2: xor s1, s1, a1        ▷ apply U (k_1 or k_5||k_4)
```

The algorithm 2, presents the implementation for GIFT-64. In the case of GIFT-128, the only difference is that for the operation 3, we have used the mask `0x80000000` ($1 \ll 31$), as defined in section 3.1.

Algorithm 2 `AddRoundKey (round constant GIFT-64)`

```
1: lb t0, 0(s4)          ▷ load round constant | s4-r.c. address
2: xor s3, s3, t0        ▷ apply round constant
3: li t0, 0x8000
4: xor s3, s3, t0        ▷ S_3 ← S_3 ⊕ (1 ≪ 15)
```

Furthermore, the key scheduling is the same as in the case of the baseline implementation and does not require any modifications.

4.4 Fixslicing

The fixsliced implementation uses the same initial data representation as in bitslicing, as well as, the same `SubCells` definition. Even though both GIFT-64 and GIFT-128 share the same idea of replacing `PermBits` with `AlignBits`, and therefore significantly decreasing the complexity of the permutation layer, the way in which the round key is adapted to the new representation is different. In the case of GIFT-64 `AddRoundKey` remains unchanged i.e., the same as for bitslicing, while `KeySchedule` is expected to adjust the key state such that it matches the cipher state for each sub-round. Conversely, for GIFT-128 the `KeySchedule` for bitslicing is reused, while `AddRoundKey` is enhanced with an additional sublayer (`MapKey`), which is meant to map the bitsliced key state to a fixsliced representation, such that it can be applied to the cipher state.

In the case of GIFT-64 the implementations of `AlignBits` and `KeySchedule` follow the definitions from the section 3.2. Conversely, for GIFT-128 the `MapKey` is different for each sub-round, as well as, it is absent for the sub-round 5, since in this sub-round the bits return to the initial, bitsliced, position i.e., the round key can be applied without modifications. The mapping, for the sub-rounds 1-4, is implemented through bit permutations, which leads to a shift of the computational complexity from the `PermBits` to the `KeySchedule` layer. Our definition of the `MapKey` sub-layer is presented in Appendix 8.2.

4.5 Fixslicing with Key Precomputation

The goal of fixslicing is to minimize the complexity of the permutation layer, by performing pseudo-permutations i.e., `AlignBits`. This has been achieved for both GIFT-64 and GIFT-128, as it can be observed in Section 5. However, in the case of GIFT-128, the cost of the lower complexity of `PermBits` is a significant performance penalty for `AddRoundKey`, due to the additional sublayer i.e., `MapKey`. Nonetheless, the main source of performance degradation is now related to the representation of the key, which can be addressed by precomputing the key for all rounds and their sub-rounds. The downside of this method is that additional memory is required in order to store the precomputed keys. In the case of GIFT-64, 896 bits are required, in addition to the input block and the precomputed round constants i.e., 896 = (nr rounds) × (nr sub-rounds) × ((size of k_0) + (size of k_1)) = $7 \times 4 \times (16 + 16)$. Similarly, in the case of GIFT-128, 2560 bits are required i.e., 2560 = (nr rounds) × (nr sub-rounds) × ((size of $k_5||k_4$) + (size of $k_1||k_0$)) = $8 \times 5 \times (32 + 32)$. However, as presented in Section 5 the increased data size, does not lead to higher memory requirements, since Key Scheduling is omitted at runtime, which leads to a significantly lower code size.

5 Results

The performance of our implementations has been measured, and is presented in Tables 2 and 3, together with the respective memory requirements in Tables 4 and 5. Each implementation has been divided into 6 sections. The sections `SubCells`, `PermBits`, `AddRoundKey` and `KeySchedule` represent the respective definitions presented in Section 2. The section `Initialization` represents the initialization of the data necessary for the round function e.g. loading the cipher state and the key state. The Section `Other` includes additional operations such as modifying the loop guard and performing branching for each loop iteration. Additionally, we have measured the performance of the reference C implementation [6], by using the `gcc` compiler version 11.1.0, provided by the RISC-V GNU Compiler Toolchain [7], with the optimization flag `-O3`.

Table 2: GIFT-64 implementations performance measurements.

Implementation	Speed (clock cycles)						
	Initialization	SubCells	PermBits	AddRoundKey	KeySchedule	Other	Total
C reference (-O3)	32090	1904	37548	44100	12068	4055	131765
Baseline	18	4088	5432	3808	448	147	13941
Bitslicing	64	336	3052	224	532	68	4276
Fixslicing	56	336	609	287	672	34	1994
Fixslicing & KP	24	336	609	518	56	34	1577

Table 3: GIFT-128 implementations performance measurements.

Implementation	Speed (clock cycles)						
	Initialization	SubCells	PermBits	AddRoundKey	KeySchedule	Other	Total
C reference (-O3)	34200	5360	97480	91800	17080	4955	250875
Baseline	16	11680	18400	11120	760	179	42155
Bitslicing	15	480	10320	320	840	90	12065
Fixslicing	14	480	1120	5352	840	87	7893
Fixslicing & KP	10	480	1120	360	80	35	2085

Table 4: GIFT-64 implementations, memory requirements.

Memory (bytes)	Implementation				
	C reference (-O3)	Baseline	Bitslicing	Fixslicing	Fixslicing & KP
Code size	5890	1692	688	1090	762
Data size	388	168	52	80	176
Total	6278	1860	740	1170	938

Table 5: GIFT-128 implementations memory requirements.

Memory (bytes)	Implementation				
	C ref. (-O3)	Baseline	Bitslicing	Fixslicing	Fixslicing & KP
Code size	1046	3642	1180	3618	892
Data size	432	240	72	192	496
Total	1478	3882	1252	3810	1388

6 Discussion

Mappings that involve multiple bit-level operations, such as permutations, are the main source of performance degradation. As it can be observed in the Tables 2 and 3, the bitsliced implementation has significantly reduced the number of clock cycles required for `SubCells` and `AddRoundKey`. In the baseline implementation of `SubCells`, in order to use the lookup table, each nibble was isolated and shifted to the least-significant bits of a register, as well as, the replacement value was shifted to the correct position. This sequence of operations results in significant performance penalties. The bitsliced implementation of `SubCells` eliminates the need of isolating and shifting nibbles, hence, resulting in more efficient computation of the substitution layer. In the case of `AddRoundKey`, the acceleration is explained by the fact that key expansion, which involves multiple bit-level operations, is not required since the round key can be directly applied to the required slices. Moreover, it can be observed that the performance of the permutation layer has also been improved. This is due to the technique described in Section 3, which reduces the number of bit-level operations. Overall, the bitsliced implementations of GIFT-64 and GIFT-128 have reduced the number of required clock cycles by 69.33% and 71.38%, respectively.

Despite the significant acceleration obtained through bitslicing, `PermBits` has remained the principal consumer of clock cycles. The fixsliced implementation of GIFT-64 has resulted in a more uniform distribution of complexity across the layers, and it has reduced the overall number of clock cycles by 85.7%, compared to the baseline. Even though the fixsliced implementation of GIFT-128 has also reduced the total number of clock cycles i.e., by 81.28%, instead of `PermBits` complexity being eliminated, it has been moved to

`AddRoundKey`, as it can be observed in Table 3, due to the additional sublayer i.e., `MapKey`. Nonetheless, this complexity shift was beneficial, because it can be eliminated through key pre-computation, as described in Section 3. By using fixslicing in combination with key pre-computation, the number of clock cycles has been reduced by 88.69% for GIFT-64 and by 95.05% for GIFT-128. Therefore, this is the technique that is capable of providing the highest acceleration for GIFT on RV32I. Moreover, fixslicing with key pre-computation also has lower memory requirements than its counter-part without key pre-computation i.e., 938 bytes (GIFT-64) and 1388 bytes (GIFT-128). This is explained by a significant decrease in the code size, which compensates for the increase in the data size. The decrease is due to the elimination of the Key Scheduling section since the pre-computed round keys are stored in memory in a hard-coded fashion. Nonetheless, the implementation with the lowest memory requirements is bitslicing i.e., 740 bytes (GIFT-64) and 1252 bytes (GIFT-128).

7 Conclusion and Future Work

We have identified three optimization techniques which are applicable for GIFT on RV32I i.e., Bitslicing, Fixslicing and Fixslicing with key pre-computation. Each technique has been implemented in assembly, and a quantitative assessment of their performance has been executed. All techniques have significantly accelerated the encryption latency. Fixslicing with key pre-computation has been identified as the most efficient technique, with a clock cycle reduction of 88.69% for GIFT-64 and 95.05% for GIFT-128. This technique is also applicable in memory-constrained environments. Alternatively, bitslicing can slightly decrease the memory requirements, at the cost of a significantly higher latency. Regarding future work on GIFT, the natural extension of the current approach is towards recent side-channel resistant implementations such as code-based masking [17] and glitch-robust [5] schemes, while employing slicing techniques to accelerate performance on the RISC-V platform.

References

[1] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A new gift representation: Fast constant-time implementations of gift and gift-cofb on arm cortex-m. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):402–427, Jun. 2020.

[2] Subhadeep Banik, Sumit Pandey, Thomas Peyrin, Siang Meng Sim, and Yosuke Todo. Gift: A small present. pages 321–345, 08 2017.

[3] Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *Fast Software Encryption*, pages 260–272, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[4] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021, Issue 2:136–158, 2021.

[6] Gift. Reference implementation. https://github.com/giftcipher/gift, 2022.

[7] RISC-V GNU. Compiler toolchain. https://github.com/riscv-collab/riscv-gnu-toolchain, 2022.

[8] Gregor Leander. On linear hulls, statistical saturation attacks, present and a cryptanalysis of puffin. volume 6632, pages 303–322, 05 2011.

[9] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. *NISTIR 8114*, 2017.

[10] Kerry A. McKay, Larry Bassham, Meltem Sönmez Turan, Çağdaş Çalık, and Donghoon Chang. Status report on the first round of the nist lightweight cryptography standardization process. *NISTIR 8268*, 2019.

[11] NIST. Submission requirements and evaluation criteria for the lightweight cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf, 2018.

[12] NIST. Lightweight cryptography, round 1 candidates. https://csrc.nist.gov/Projects/lightweight-cryptography/round-1-candidates, 2022.

[13] Yu Sasaki. Integer linear programming for three-subset meet-in-the-middle attacks: Application to gift. In *IWSEC*, 2018.

[14] SiFive. Fe310-g002 manual. https://www.sifive.com/documentation, 2022.

[15] SiFive. Hifive1 rev b. https://www.sifive.com/boards/hifive1-rev-b, 2022.

[16] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Çağdaş Çalık, Lwrence Bassham, Jinkeon Kang, and John Kelsey. Status report on the second round of the nist lightweight cryptography standardization process. *NISTIR 8369*, 2021.

[17] Weijia Wang, Pierrick Méaux, Gaëtan Cassiers, and François-Xavier Standaert. Efficient and private computations with code-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):128–171, Mar. 2020.

[18] Baoyu Zhu, Xiaoyang Dong, and Hongbo Yu. Milp-based differential attack on round-reduced gift. Cryptology ePrint Archive, Report 2018/390, 2018. https://ia.cr/2018/390.

8 Appendix

8.1 Efficient matrix transposition

Let M be a 4×4 matrix defined as,

$$M = \begin{bmatrix} 15 & 14 & 13 & 12 \\ 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

Let M^v be the row vector representation of M i.e. $M^v \equiv M$, where

$$M^v = \begin{bmatrix} 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}$$

The row vector representation of the transpose of M (M^T) is defined as,

$$M^{Tv} = \begin{bmatrix} 15 & 11 & 7 & 3 & 14 & 10 & 6 & 2 & 13 & 9 & 5 & 1 & 12 & 8 & 4 & 0 \end{bmatrix}$$

A naive approach of mapping M^v to M^{Tv}, would be to group cells based on the shift direction and amplitude i.e.,

$$\begin{aligned} (15, 10, 5, 0) &: \text{no shift} \\ (14, 9, 4) &: \text{shift 3 cells to the right} \\ (13, 8) &: \text{shift 6 cells to the right} \\ (12) &: \text{shift 9 cells to the right} \\ (3) &: \text{shift 9 cells to the left} \\ (7, 2) &: \text{shift 6 cells to the left} \\ (11, 6, 1) &: \text{shift 3 cells to the left} \end{aligned}$$

A better approach is to decompose the problem i.e., a 4×4 matrix can be treated as 2×2 matrix which contains in each cell a 2×2 matrix. Therefore we can first transpose each inner 2×2 matrix, and as a last step the outer matrix i.e.,

$$M = \begin{bmatrix} 15 & 14 & 13 & 12 \\ 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 15 & 11 & 13 & 9 \\ 14 & 10 & 12 & 8 \\ 7 & 3 & 5 & 1 \\ 6 & 2 & 4 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 15 & 11 & 7 & 3 \\ 14 & 10 & 6 & 2 \\ 13 & 9 & 5 & 1 \\ 12 & 8 & 4 & 0 \end{bmatrix} = M^T$$

M^v can be mapped to the row vector representation of the intermediary matrix as follows,

$$\begin{aligned} (15, 13, 10, 8, 7, 5, 2, 0) &: \text{no shift} \\ (14, 12, 6, 4) &: \text{shift 3 cells to the right} \\ (11, 9, 3, 1) &: \text{shift 3 cells to the left} \end{aligned}$$

Similarly, the row representation of the intermediary matrix is mapped to $M^T v$ as follows,

$$\begin{aligned} (15, 11, 14, 10, 5, 1, 4, 0) &: \text{no shift} \\ (13, 9, 12, 8) &: \text{shift 6 cells to the right} \\ (7, 3, 6, 2) &: \text{shift 6 cells to the left} \end{aligned}$$

Hence, with this approach, we can avoid the movement of one group of cells. Moreover, the naive approach uses 6 shifts and 1 no-shift. One shift translates to 4 instructions on RV32I, while 1 no-shift is implemented with 2 instructions. Hence, the naive approach would use 26 assembly instructions. Conversely, the other approach uses 4 shifts and 2 no shifts i.e., 20 assembly instructions. Therefore, the second method can save 6 instructions for the transpose of a 4×4 matrix.

The same approach can be applied for computing the transpose of other types of matrices. For instance, in the case of a 4×8 matrix B the transpose, B^T can be computed as follows,

$$
\begin{aligned}
B = &\begin{bmatrix} 31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\ 23 & 22 & 21 & 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix} \\
\rightarrow &\begin{bmatrix} 31 & 23 & 29 & 21 & 27 & 19 & 25 & 17 \\ 30 & 22 & 28 & 20 & 26 & 18 & 24 & 16 \\ 15 & 7 & 13 & 5 & 11 & 3 & 9 & 1 \\ 14 & 6 & 12 & 4 & 10 & 2 & 8 & 0 \end{bmatrix} \\
\rightarrow &\begin{bmatrix} 31 & 23 & 15 & 7 & 27 & 19 & 11 & 3 \\ 30 & 22 & 14 & 6 & 26 & 18 & 10 & 2 \\ 29 & 21 & 13 & 5 & 25 & 17 & 9 & 1 \\ 28 & 20 & 12 & 4 & 24 & 16 & 8 & 0 \end{bmatrix} \\
\rightarrow &\begin{bmatrix} 31 & 23 & 15 & 7 \\ 30 & 22 & 14 & 6 \\ 29 & 21 & 13 & 5 \\ 28 & 20 & 12 & 4 \\ 27 & 19 & 11 & 3 \\ 26 & 18 & 10 & 2 \\ 25 & 17 & 9 & 1 \\ 24 & 16 & 8 & 0 \end{bmatrix} = B^T
\end{aligned}
$$

In the case of a 4×8 matrix, a naive computation of the transpose will require 31 groups, i.e., 1 no-shift and 30 shift groups. Hence, the RV32I implementation would result in 122 instructions. Conversely, the new method requires only 10 shift groups and 3 no-shift groups, which results in only 46 instructions.

8.2 MapKey definition

The `MapKey` sublayer is only present for GIFT-128. It is different for each sub-round, as well as, it is absent for the sub-round 5 since in this sub-round the bits return to the initial, bitsliced, position i.e., the round key can be applied without modifications. For each sub-round $i \in [1, 4]$, $\texttt{MapKey}^i$ is applied independently to $k_1 \| k_0$ and $k_5 \| k_4$.

Before we proceed with the iterations of `MapKey`, we define the `LeftShift` and `RightShift` operations, which represent the bit shifting in the respective direction of a masked sequence of bits. The definition of `RightShift` is presented in Algorithm 3, while `LeftShift` is implemented similarly, however, instead of `srl`, `sll` (logical left shift) is used.

Algorithm 3 Right Shift

Require:
dst - destination register;
src - source register;
mask - used to isolate the required bits;
shift - the amount of right shift (in bits);

```
1: li t0, mask                    ▷ Load the mask in register t0
2: and t0, src, t0                ▷ Apply the mask and store result in t0
3: srl t0, t0, shift              ▷ Shift isolated bits by <shift> cells to the right
4: or dst, dst, t0                ▷ Store the result in register <dst>
```

MapKey^1 **.** Firstly, we consider the key state as a 4×8 matrix i.e.,

$$b_{31}b_{30}...b_1b_0 \equiv \begin{bmatrix} b_{31} & b_{30} & b_{29} & b_{28} & b_{27} & b_{26} & b_{25} & b_{24} \\ b_{23} & b_{22} & b_{21} & b_{20} & b_{19} & b_{18} & b_{17} & b_{16} \\ b_{15} & b_{14} & b_{13} & b_{12} & b_{11} & b_{10} & b_9 & b_8 \\ b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \end{bmatrix}$$

The mapping between bitstring and matrix representations does not require any modification, since this is only a way of viewing the key state i.e., the key state is stored in a register as a bitstring at all times. Firstly, we compute the transpose of this matrix, as described in appendix 8.1, and return to the bitstring representation, which is then updated, by using the algorithm 4. The result is the required fixsliced representation of the key.

Algorithm 4 MapKey[1] Swap

Require:
a4 - src. bitstring (after transpose);
a5 - the destination register;

```
1: RightShift a5, a4, 0x44444444, 1
2: RightShift a5, a4, 0x88888888, 3
3: LeftShift a5, a4, 0x11111111, 3
4: LeftShift a5, a4, 0x22222222, 1
```

MapKey[2]. For this sub-round we consider the key state as a bitstring i.e., as it is. The bitsliced representation is mapped to the fixsliced counterpart through a series of `LeftShift` and `RightShift`, as defined in algorithm 5.

Algorithm 5 MapKey[2] Sublayer

Require:
a0 - the source bitstring;
a5 - the destination register;

```
1: li a5, 0x00200400
2: and a5, a0, a5
3: LeftShift a5, a0, 0x00000002, 30
4: RightShift a5, a0, 0x00801000, 3
5: LeftShift a5, a0, 0x00000008, 27
6: RightShift a5, a0, 0x02004000, 6
7: LeftShift a5, a0, 0x00000020, 24
8: RightShift a5, a0, 0x08010000, 9
9: LeftShift a5, a0, 0x00000080, 21
10: RightShift a5, a0, 0x20040000, 12
11: LeftShift a5, a0, 0x00000200, 18
12: RightShift a5, a0, 0x80100000, 15
13: LeftShift a5, a0, 0x00000801, 15
14: RightShift a5, a0, 0x00400000, 18
15: LeftShift a5, a0, 0x00002004, 12
16: RightShift a5, a0, 0x01000000, 21
17: LeftShift a5, a0, 0x00008010, 9
18: RightShift a5, a0, 0x04000000, 24
19: LeftShift a5, a0, 0x00020040, 6
20: RightShift a5, a0, 0x10000000, 27
21: LeftShift a5, a0, 0x00080100, 3
22: RightShift a5, a0, 0x40000000, 30
```

MapKey[3]. Similarly with `MapKey`[2], we could not find a more efficient way of

implementing this sublayer for round 3, other than a series of `LeftShift` and `RightShift`, as presented in algorithm 6.

Algorithm 6 MapKey^3 Sublayer

Require:
a0 - the source bitstring;
a4 - the destination register;

```
li t0, 0x00200400
and a4, a0, t0
LeftShift a4, a0, 0x00000001, 30
RightShift a4, a0, 0x00400800, 3
LeftShift a4, a0, 0x00000002, 27
RightShift a4, a0, 0x00801000, 6
LeftShift a4, a0, 0x00000004, 24
RightShift a4, a0, 0x01002000, 9
LeftShift a4, a0, 0x00000008, 21
RightShift a4, a0, 0x02004000, 12
LeftShift a4, a0, 0x00000010, 18
RightShift a4, a0, 0x04008000, 15
LeftShift a4, a0, 0x00010020, 15
RightShift a4, a0, 0x08000000, 18
LeftShift a4, a0, 0x00020040, 12
RightShift a4, a0, 0x10000000, 21
LeftShift a4, a0, 0x00040080, 9
RightShift a4, a0, 0x20000000, 24
LeftShift a4, a0, 0x00080100, 6
RightShift a4, a0, 0x40000000, 27
LeftShift a4, a0, 0x00100200, 3
RightShift a4, a0, 0x80000000, 30
```

MapKey^4. An efficient implementation of `MapKey` for round 4 requires the representation of the key state as an 8×4 matrix i.e.,

$$b_{31}b_{30}...b_1b_0 \equiv \begin{bmatrix} b_{31} & b_{30} & b_{29} & b_{28} \\ b_{27} & b_{26} & b_{25} & b_{24} \\ b_{23} & b_{22} & b_{21} & b_{20} \\ b_{19} & b_{18} & b_{17} & b_{16} \\ b_{15} & b_{14} & b_{13} & b_{12} \\ b_{11} & b_{10} & b_9 & b_8 \\ b_7 & b_6 & b_5 & b_4 \\ b_3 & b_2 & b_1 & b_0 \end{bmatrix}$$

Similarly to `MapKey`^{1}, the transpose of this matrix is computed. The rows of the resulting 8×4 matrix are swapped as presented in algorithm 7. The result is the required fixsliced representation.

Algorithm 7 MapKey^4 Swap

Require:
a4 - the source bitstring (after transpose);
a5 - the destination register;

```
RightShift a5, a4, 0xff000000, 24
RightShift a5, a4, 0x00ff0000, 8
LeftShift a5, a4, 0x0000ff00, 8
LeftShift a5, a4, 0x000000ff, 24
```