Efficient Implementation Strategies for Block Ciphers on ARMv8

Bachelorarbeit

Bastian Engel

April 2, 2023

Abstract

Lorem ipsum dolor [1] sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Declaration

I hereby declare that ...

Contents

Chapter 1

Introduction

1.1 Notation

1.2 Block ciphers

Securing communication channels between different parties has been a long-term subject of study for cryptographers and engineers which is essential to our modern world to cope with ever-increasing amounts of devices producing and sharing data. The main way to facilitate high-throughput, confidential communications nowadays is through the use of symmetric cryptography in which two parties share a common secret, called a key, which allows them to encrypt, share and subsequently decrypt messages to achieve confidentiality against third parties. Ciphers can be divided into two categories; block ciphers, which always encrypt fixed-sized messages called blocks, and stream ciphers, which continuously provide encryption for an arbitrarily long, constant stream of data.

A block cipher can be defined as a bijection between the input block (the message) and the output block (the ciphertext). For any block cipher with block size n, we denote the key-dependent encryption and decryption functions as $E_K, D_K : \mathbb{F}_2^n \to \mathbb{F}_2^n$. The simplest way to characterize this bijection is through a lookup table which yields the highest possible performance as each block can be encrypted by one simple lookup depending on the key and the message. This is not practical though due to most ciphers working with block and key sizes $n, |K| \geq 64$. For a block cipher with $n = 64, |K| = 128$, a space of $2^{64}2^{128}64 = 2^{198}$ is necessary. Considering modern consumer hard disks being able to store data in the order of 2^{40}, it is easy to see that a lookup table is wholly impractical. We therefore describe block ciphers algorithmically which opens up possibilities for different tradeoffs and security concerns.

1.2.1 GIFT

`GIFT`[1], first presented in the *CHES 2017* cryptographic hardware and embedded systems conference, is a lightweight block cipher based on a previous design called `PRESENT`, developed in 2007. Its goal is to offer maximum security while being extremely light on resources. Modern battery-powered devices like RFID tags or low-latency operations like on-the-fly disc encryption present strong hardware and power constraints. GIFT aims to be a simple, low-energy cipher suited for these kinds of applications.

`GIFT` comes in two variants; `GIFT-64` working with 64-bit blocks and `GIFT-128` working with 128-bit blocks. In both cases, the key is 128 bits long. The design is a very simple, round-based substitution-permutation network (SPN). One round consists in a sequential application of the confusion layer by means of 4-bit S-boxes and subsequent diffusion through bit permutation. After the bit permutation, a round key is added to the cipher state and the single round is complete. `GIFT-64` uses 28 rounds while `GIFT-128` uses 40 rounds.

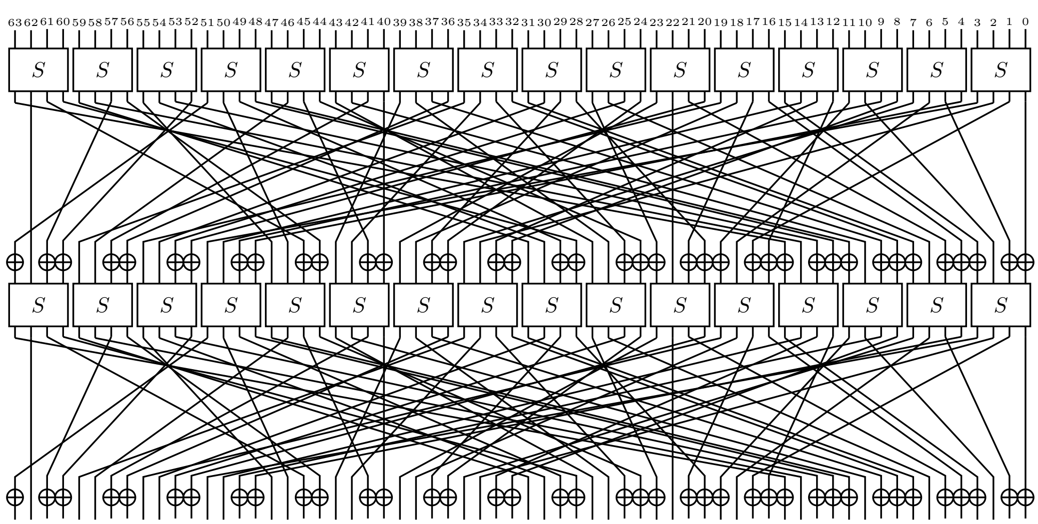

Figure 1.1: Two rounds of GIFT-64

Substitution layer

The input of `GIFT` is split into 4-bit nibbles which are then fed into 16 S-boxes for `GIFT-64` and 32 S-boxes for `GIFT-128`. The S-box $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ is defined as follows:

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(x)$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

Permutation layer

The permutation P works on individual bits and maps bit b_i to $b_{P(i)}, i \in \{0, 1, \ldots, n-1\}$. The different permutations for `GIFT-64` and `GIFT-128` can be expressed by:

$$P_{64}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 16\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4)$$
$$P_{128}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 32\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4)$$

Round key addition

The last step of each round consists in XORing a round key R_i to the cipher state. The new cipher state s_{i+1} after each full round is therefore given by

$$s_{i+1} = P(S(s_i)) \oplus R_i$$

Round key extraction and key schedule

Round key extraction differs for `GIFT-64` and `GIFT-128`. Let $K = k7||k6||\ldots||k0$ denote the 128-bit key state.

GIFT-64 . We extract two 16-bit words $U||V = k_1||k_0$ from the key state. u_i and v_i are XORed to r_{4i+1} and r_{4i} of the round key R respectively.

GIFT-128 . We extract two 32-bit words $U||V = k_5||k4||k1||k_0$ from the key state. u_i and v_i are XORed to r_{4i+2} and b_{4i+1} of the round key R respectively.

In both cases, we additionally XOR a round constant $C = c_5c_4c_3c_2c_1c_0$ to bit positions $n-1, 23, 19, 15, 11, 7, 3$. The round constants are generated using a 6-bit affine linear-feedback shift register and have the following values:

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

The key state is then updated by setting $k_1 \leftarrow k_1 \ggg 2$, $k_0 \leftarrow k_0 \ggg 12$ and rotating the new state 32 bits to the right:

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

1.2.2 Camellia

1.3 The ARMv8 platform

With small devices, embedded processors and ASICs becoming ever more ubiquitous and essential in areas like medicine or automotive design, the need for ...

Chapter 2

Implementation strategies

Due to the structural differences of SPN- and Feistel network-based ciphers, we shall analyze these two separately.

2.1 Strategies for SPN

Three implementation strategies for substitution-permutation networks are introduced by [2]:

- Table-based implementations
- `vperm` implementations
- Bitslice implementations

2.1.1 Table-based

Table-driven programming is a simple way to increase performance of operations by tabulating the results, therefore requiring only a single memory access to acquire the result. This approach is obviously limited to manageable table sizes, so while tabulating a function like the AES S-box $S_{AES} : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ requires only 2^{11} space, tabulating the `GIFT` permutation layer $P_{GIFT} : \mathbb{F}_2^{64} \to \mathbb{F}_2^{64}$ would require 2^{70} space, which is totally unfeasible.

A common approach is to tabulate the output of each S-box, including the diffusion layer, and then XORing the results together. Let n denote the internal cipher state size and s the size of a single S-box in bits. For each S-box $S_i, i \in \{0, \ldots, \frac{n}{s}\}$, we can construct a mapping $T_i : \mathbb{F}_2^s \rightarrow \mathbb{F}_2^n$ representing substitution with subsequent permutation of that single S-box. The cipher state before round key addition is then given by $\bigoplus_{i=0}^{\frac{n}{s}-1} T_i(m_i)$ for each s-bit message chunk m_i. This approach requires space of $\frac{n}{s}|\mathbb{F}_2^s|n = \frac{n^2 2^s}{s}$ bits, which, for `GIFT-64`, results in a manageable size of $\frac{64^2 2^4}{4} = 2^{14}$ bits which equals 16 KiB.

Constructing the tables

For `GIFT-64`, table construction is relatively straightforward and can be done as follows:

Listing 2.1: Table construction algorithm

```
tables <- [][]
for sbox_index from 0 to 15 do
    for sbox_input from 0 to 15 do
        output <- sbox(sbox_input)
        output <- permute(output << (4 * sbox_index))
        tables[sbox_index][sbox_input] <- output
```

Implementing this algorithm gives us the following table representing the first and second S-box.

x	$T_0(x)$	$T_1(x)$	...
0x0	0x1	0x1000000000000	...
0x1	0x8000000020000	0x800000002	...
0x2	0x400000000	0x40000	...
0x3	0x8000400000000	0x800040000	...
0x4	0x400020000	0x40002	...
0x5	0x8000400020001	0x1000800040002	...
0x6	0x20001	0x1000000000002	...
0x7	0x8000000000001	0x1000800000000	...
0x8	0x20000	0x2	...
0x9	0x8000400000001	0x1000800040000	...
0xa	0x8000000020001	0x1000800000002	...
0xb	0x400020001	0x1000000040002	...
0xc	0x400000001	0x1000000040000	...
0xd	0x0	0x0	...
0xe	0x8000000000000	0x800000000	...
0xf	0x8000400020000	0x800040002	...

The tables for `GIFT-128` can be generated in a similar way by looping through all 32 S-boxes instead of 16 on line 3.

2.1.2 Using `vperm`

Nowadays, most instructions set architectures support single-instruction, multiple-data processing. The idea of such an SIMD system is to work on multiple data stored in vectors at once to speed up calculations. For A64, two types of vector processing are available:

1. Advanced SIMD, known as NEON
2. Scalable Vector Extension (SVE)

We will take a look at NEON as this is the type of vector processing supported by the Cortex-A73 processor.

ARM Neon

The register file of the NEON unit is made up of 32 quad-word (128-bit) registers `V[0-31]`, each extending the standard 64-bit floating-point registers `D[0-31]`. These registers are divided into equally sized lanes on which the vector instructions operate. Valid ways to interpret for example the register `V0` are:

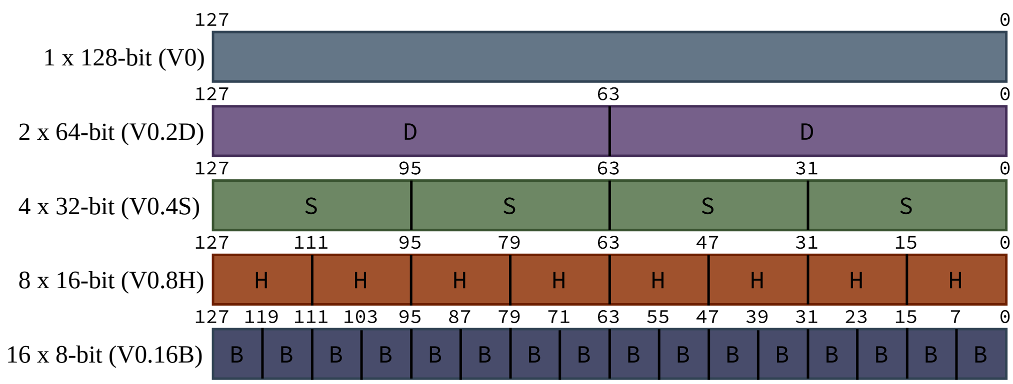

Figure 2.1: Divisions of the V0 register

NEON instructions interpret their operands' layouts (i.e. lane count and width) through the use of suffixes such as `.4S` or `.8H`. For example, adding eight 16-bit halfwords from register `V1` and `V2` together and storing the result in `V0` can be done as follows:

```
ADD V0.8H, V1.8H, V2.8H
```

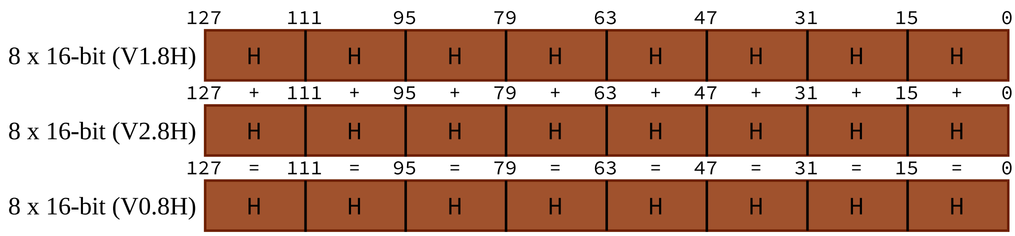

Figure 2.2: Addition of two vector registers

The plenitude of different processing instructions allow flexible ways to further speed up algorithms having reached their optimizational limit on non-SIMD platforms. `vperm`, a general term standing for *vector permute*, is a common instruction on SIMD machines. Called `TBL` on NEON, it is used for parallel table lookups and arbitrary permutations. It takes two inputs to perform a lanewise lookup:

1. A register with lookup values
2. One or more registers containing data

S-box lookup

This instruction can be used to implement S-box lookup of all 16 S-boxes in a single instruction. We do this by packing our 64-bit cipher state $s = s_{15}||s_{14}||\dots||s_0$ into a vector register V_0. Because we can only operate on whole bytes, we put each 4-bit S-box into an 8-bit lane which neatly fits into the 128-bit registers. We then put the S-box itself into register V_1 which will be used as the data register for the table lookup.

The confusion layer can now be performed through one `TBL` instruction:

```
TBL V0.16B, V1.16B, V0.16B
```

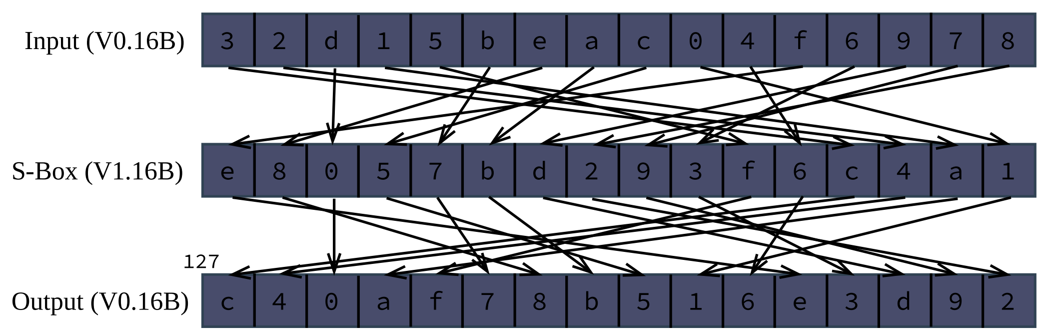

Figure 2.3: Performing the S-Box lookup in parallel

2.1.3 Bitslicing

Bitslicing refers to the technique of splitting up n bits into m slices to achieve a more efficient representation to operate on. The structure of `GIFT` naturally offers possibilities for bitslicing. We split the cipher state bits $b_{63}b_{62}\dots b_0$ into four slices $S_i, i \in \{0,1,2,3\}$ such that the i-th slice contains all i-th bits of the individual S-boxes. This is equivalent to transposing the bit matrix.

$$S = \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} b_{60}b_{56}b_{52}\dots b_0 \\ b_{61}b_{57}b_{53}\dots b_1 \\ b_{62}b_{58}b_{54}\dots b_2 \\ b_{63}b_{59}b_{55}\dots b_3 \end{bmatrix}$$

Parallel S-Boxes

This representation offers multiple advantages. We first note that computation of the S-box can be executed in parallel, similar to the `vperm` technique above. This can be done by finding an algorithmic way to apply the S-box which has already been proposed by the original `GIFT` authors:

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

This is very efficient as it only requires six XOR-, three AND and one OR operation.

An important property of the permutation is the fact that bits always stay in their slice. This means we can decompose the permutation P into four permutations $P_i, i \in \{0, 1, 2, 3\}$ and apply these permutations separately to each slice. One possible way to implement a permutation P_i in software is to mask off all bits individually, shift them to their correct position and OR them together:

$$P_i(S_i) = \bigvee_{k=0}^{15} (S_i \wedge m_i) \ll s_i$$

This approach requires 47 operations, meaning all four permutations require over 150 operations which would present a major bottleneck to the round function. We can improve on this by working on multiple message blocks at once and using the aforementioned `vperm` instruction to implement the bit shuffling. We then need only four instructions for the complete diffusion layer.

Using `vperm` for slice permutation

We cannot use the `TBL` instruction directly as we need to shuffle individual bits, but the smallest data we can operate on are bytes. We therefore encrypt $8n$ messages at once which allows us to create bytewise groupings. These messages are put into $4m$ registers with register R_{4i} containing S_0, register R_{4i+1} containing S_1 and so forth. With block size BS and register size RS, the following must hold:

$$8n \cdot BS = 4m \cdot RS$$

In the case of `GIFT-64` with $BS = 64$ and ARM NEON with $RS = 128$, we get

$$8n \cdot 64 = 4m \cdot 128 \Leftrightarrow n = m$$

$n = m = 1$ would be a valid choice which yields eight messages divided into four registers. We choose $n = m = 2$ so we can directly utilize the algorithm for bit packing presented by the original GIFT authors, although it is simple to adapt this algorithm to only four registers and eight messages by adjusting the `SWAPMOVE` shift and mask values.

Packing the data into bitslice format

Let $a, b, \ldots, p$ be sixteen messages of length 64 with subscripts denoting individual bits. We first put these messages into eight SIMD registers $V_0, V_1, \ldots, V_7$:

$$\begin{aligned} V_0 &= b||a & V_4 &= j||i \\ V_1 &= d||c & V_5 &= l||k \\ V_2 &= f||e & V_6 &= n||m \\ V_3 &= h||g & V_7 &= p||o \end{aligned}$$

We then use the `SWAPMOVE` technique to bring the data into bitslice format. This operation operates on two registers A, B using mask M and shift value N. It swaps bits in A masked by $(M \ll N)$ with bits in B masked by M in using only three XOR-, one AND- and two shift operations.

$$\begin{aligned} &\text{SWAPMOVE}(A, B, M, N): \\ &\quad T = ((A \gg N) \oplus B) \wedge M \\ &\quad B = B \oplus T \\ &\quad A = A \oplus (T \ll N) \end{aligned}$$

One caveat of this approach is the fact that NEON registers cannot be shifted in their entirety due to the fact bits are not able to cross lanes. This leads to the problem of being able to shift at most two lanes of 64 bits at once. We thus need to implement the `shr(V,n)` and `shl(V,n)` operations on our own. This can be done by first extracting the 64-bit lanes a, b out of $V = b||a$, shifting the lanes individually and finally shifting and ORing the crossing bits back into the other lane.

$$
\begin{aligned}
&\text{shl}(V, n): \\
&\qquad a, b &&= V[0], V[1] \\
&\qquad c &&= (a \gg (64 - n)) \\
&\qquad a &&= (a \ll n) \\
&\qquad b &&= (b \ll n) \vee c \\
&\qquad V[0], V[1] &&= a, b
\end{aligned}
$$

The following operations group all ith bits of the messages $a, c, \dots, o$ into bytes and puts these into the lower half of the registers $V_{i \bmod 8}$. The same is done for messages $b, d, \dots, p$, only differing in that the bytes are put into the upper half of the registers.

$$
\begin{array}{ll}
\text{SWAPMOVE}(V_0, V_1, 0x5555\dots55, 1) & \text{SWAPMOVE}(V_4, V_5, 0x5555\dots55, 1) \\
\text{SWAPMOVE}(V_2, V_3, 0x5555\dots55, 1) & \text{SWAPMOVE}(V_6, V_7, 0x5555\dots55, 1) \\
\text{SWAPMOVE}(V_0, V_2, 0x3333\dots33, 2) & \text{SWAPMOVE}(V_4, V_6, 0x3333\dots33, 2) \\
\text{SWAPMOVE}(V_1, V_3, 0x3333\dots33, 2) & \text{SWAPMOVE}(V_5, V_7, 0x3333\dots33, 2) \\
\text{SWAPMOVE}(V_0, V_4, 0x0f0f\dots0f, 4) & \text{SWAPMOVE}(V_1, V_5, 0x0f0f\dots0f, 4) \\
\text{SWAPMOVE}(V_2, V_6, 0x0f0f\dots0f, 4) & \text{SWAPMOVE}(V_3, V_7, 0x0f0f\dots0f, 4)
\end{array}
$$

With $Ax = o_x m_x k_x j_x g_x e_x c_x a_x$ and $Bx = p_x n_x l_x i_x h_x f_x d_x b_x$ denoting byte groups, our data now has the following permutation-friendly format:

n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V_0	$B56$	$B48$	$B40$	$B32$	$B24$	$B16$	$B8$	$B0$	$A56$	$A48$	$A40$	$A32$	$A24$	$A16$	$A8$	$A0$
V_1	$B57$	$B49$	$B41$	$B33$	$B25$	$B17$	$B9$	$B1$	$A57$	$A49$	$A41$	$A33$	$A25$	$A17$	$A9$	$A1$
V_2	$B58$	$B50$	$B42$	$B34$	$B26$	$B18$	$B10$	$B2$	$A58$	$A50$	$A42$	$A34$	$A26$	$A18$	$A10$	$A2$
V_3	$B59$	$B51$	$B43$	$B35$	$B27$	$B19$	$B11$	$B3$	$A59$	$A51$	$A43$	$A35$	$A27$	$A19$	$A11$	$A3$
V_4	$B60$	$B52$	$B44$	$B36$	$B28$	$B20$	$B12$	$B4$	$A60$	$A52$	$A44$	$A36$	$A28$	$A20$	$A12$	$A4$
V_5	$B61$	$B53$	$B45$	$B37$	$B29$	$B21$	$B13$	$B5$	$A61$	$A53$	$A45$	$A37$	$A29$	$A21$	$A13$	$A5$
V_6	$B62$	$B54$	$B46$	$B38$	$B30$	$B22$	$B14$	$B6$	$A62$	$A54$	$A46$	$A38$	$A30$	$A22$	$A14$	$A6$
V_7	$B63$	$B55$	$B47$	$B39$	$B31$	$B23$	$B15$	$B7$	$A63$	$A55$	$A47$	$A39$	$A31$	$A23$	$A15$	$A7$

Although this would already work, we prefer to have only bits of the same messages in each register - otherwise the permutation would need to operate on two source registers with the added requirement of storing the pre-permutation values for the first four registers, slowing down the round function through superfluous load/stores. This transformation is trivial by use of `TBL` with two data source operands. The final data format we operate on is as follows:

n	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V_0	$A60$	$A56$	$A52$	$A48$	$A44$	$A40$	$A36$	$A32$	$A28$	$A24$	$A20$	$A16$	$A12$	$A8$	$A4$	$A0$
V_1	$A61$	$A57$	$A53$	$A49$	$A45$	$A41$	$A37$	$A33$	$A29$	$A25$	$A21$	$A17$	$A13$	$A9$	$A5$	$A1$
V_2	$A62$	$A58$	$A54$	$A50$	$A46$	$A42$	$A38$	$A34$	$A30$	$A26$	$A22$	$A18$	$A14$	$A10$	$A6$	$A2$
V_3	$A63$	$A59$	$A55$	$A51$	$A47$	$A43$	$A39$	$A35$	$A31$	$A27$	$A23$	$A19$	$A15$	$A11$	$A7$	$A3$
V_4	$B60$	$B56$	$B52$	$B48$	$B44$	$B40$	$B36$	$B32$	$B28$	$B24$	$B20$	$B16$	$B12$	$B8$	$B4$	$B0$
V_5	$B61$	$B57$	$B53$	$B49$	$B45$	$B41$	$B37$	$B33$	$B29$	$B25$	$B21$	$B17$	$B13$	$B9$	$B5$	$B1$
V_6	$B62$	$B58$	$B54$	$B50$	$B46$	$B42$	$B38$	$B34$	$B30$	$B26$	$B22$	$B18$	$B14$	$B10$	$B6$	$B2$
V_7	$B63$	$B59$	$B55$	$B51$	$B47$	$B43$	$B39$	$B35$	$B31$	$B27$	$B23$	$B19$	$B15$	$B11$	$B7$	$B3$

We can now create permutation tables using the specification of the individual slice permutations P_i which are then applied to V_i and V_{i+4} respectively:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_0(j)$	0	12	8	4	1	13	9	5	2	14	10	6	3	15	11	7
$P_1(j)$	4	0	12	8	5	1	13	9	6	2	14	10	7	3	15	11
$P_2(j)$	8	4	0	12	9	5	1	13	10	6	2	14	11	7	3	15
$P_3(j)$	12	8	4	0	13	9	5	1	14	10	6	2	15	11	7	3

One thing to take note of is the original permutation values only show where a given byte should land, not which byte belongs to a certain position - i.e. for P_0, byte 1 should land in position 12, but the byte belonging to position 1 is byte 4. Because `TBL` works in the latter way, we have to do some trivial rearrangements.

Assuming the correct permutation values are put into registers V_8, V_9, V_{10}, V_{11}, this now allows us to compute the permutation layer for all 16 blocks in only eight permutation instructions.

```
TBL V0, V0, V8     TBL V1, V1, V9
TBL V4, V4, V8     TBL V5, V5, V9
TBL V2, V2, V10    TBL V3, V3, V11
TBL V6, V6, V10    TBL V7, V7, V11
```

Round key function

In contrast to packing and unpacking of data which is only done once in the beginning and end, a round key is derived for every round, so the round key derivation function needs to be as fast as possible. A simple but naive approach for one round would be to generate a single round key, copy it 15 times and pack the resulting registers similar to how we proceed with the messages. Due to the cost of packing the messages, this is prohibitivly expensive. Because we know where each byte group ends up after packing, we can directly XOR the round key bits to the correct position. Extending these bits to bytes can then be done simply by repeatedly shifting and ORing the registers together.

2.2 Implementation

Implementations in the C programming language for the presented strategies can be found in Appendix A. Although directly writing Assembler code could result in a small performance benefit, this generally increases the work necessary by an order of magnitude for only limited results. Instruction-level optimization and in particular register allocation is left to the compiler.

2.2.1 NEON intrinsics

The header file <arm_neon.h> provides ARM-specific data and function definitions including vector data types and C functions for working with these vectors. These functions are known as NEON intrinsics [3] and give the programmer a high-level interface to most NEON instructions. Major advantages of this approach include the ease of development as the compiler takes over register allocation and load/store operations as well as performance benefits through compiler optimizations.

Standard vector data types have the format `uintnxm_t` with lane width n in bits and and lane count m. Array types of the format `uintnxmxc_t`, $c \in \{2, 3, 4\}$ are also defined which are used in operations requiring multiple parameters like `TBL` or pairwise load/stores. Intrinsics include the operation name and lane data format as well as an optional `q` suffix to indicate operation on a 128-bit register. Multiplying eight pairs of 16-bit numbers `a,b` for example can be done via the following:

```
uint16x8_t result = vmulq_u16(a, b);
```

In this case, the compiler allocates vector registers for `a`, `b` and `result` and assembles the intrinsic to `MUL Vr.8H, Va.8H, Vb.8H`. Necessary loads and stores for the result and parameters are also handled automatically. Of special interest to us are the following intrinsics, each existing in different variants with different lane widths and also array types:

Intrinsic		Description
`uint8x16_t`	`vreinterpretq_u8_u64(uint64x2_t)`	Explicit casting
`uint64_t`	`vgetq_lane_u64(void)`	Extract a single lane
`void`	`vsetq_lane_u64(uint64_t)`	Insert a single lane
`uint64x2_t`	`vdupq_n_u64(uint64_t)`	Initialize all lanes to same value
`void`	`vst1q_u64(uint64_t*, uint64x2_t)`	Store from register to memory
`uint64x2_t`	`vld1q_u64(uint64_t*, uint64x2_t)`	Load from memory to register
`uint8x16_t`	`veorq_u8(uint8x16_t, uint8x16_t)`	bitwise XOR
`uint8x16_t`	`vandq_u8(uint8x16_t, uint8x16_t)`	bitwise AND
`uint8x16_t`	`vorrq_u8(uint8x16_t, uint8x16_t)`	bitwise OR
`uint8x16_t`	`vmvnq_u8(uint8x16_t)`	bitwise NOT
`uint8x16_t`	`vqtbl2q_u8(uint8x16_t, uint8x16_t)`	permutation (`TBL`)

2.3 Strategies for Camellia

Chapter 3

Evaluation

Acknowledgements

I want to thank ...

Appendix A

C implementations

A.1 Implementations for SPN

A.1.1 Table-based

Listing A.1: `gift_table.h`

```
#ifndef GIFT_TABLE_H
#define GIFT_TABLE_H

#include <stdint.h>

#define ROUNDS_GIFT_64 28

void gift_64_table_generate_round_keys(uint64_t round_keys[restrict
    ROUNDS_GIFT_64],
                                 const uint64_t key[restrict 2]);

uint64_t gift_64_table_subperm(const uint64_t cipher_state);

// can only encrypt using table technique!
uint64_t gift_64_table_encrypt(const uint64_t m, const uint64_t key[restrict 2]);

#endif
```

Listing A.2: `gift_table.c`

```
#include "gift_table.h"

#include <stdlib.h>
#include <string.h>

static const int round_constant[] = {
        // rounds 0-15
        0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B,
```

```c
        0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E,
        // rounds 16-31
        0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30,
        0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
        // rounds 32-47
        0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
        0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04
};

static const uint64_t tables[16][16] = {
        { 0x0000000000000001UL, 0x0008000000020000UL, 0x0000000400000000UL, 0
            x0008000400000000UL, 0x0000000400020000UL, 0x0008000400020001UL, 0
            x0000000000020001UL, 0x0008000000000001UL, 0x0000000000020000UL, 0
            x0008000400000001UL, 0x0008000000020001UL, 0x0000000400020001UL, 0
            x0000000400000001UL, 0x0000000000000000UL, 0x0008000000000000UL, 0
            x0008000400020000UL },
        { 0x0001000000000000UL, 0x0000000800000002UL, 0x0000000000040000UL, 0
            x0000000800040000UL, 0x0000000000040002UL, 0x0001000800040002UL, 0
            x0001000000000002UL, 0x0001000800000000UL, 0x0000000000000002UL, 0
            x0001000800040000UL, 0x0001000800000002UL, 0x0001000000040002UL, 0
            x0001000000040000UL, 0x0000000000000000UL, 0x0000000800000000UL, 0
            x0000000800040002UL },
        { 0x0000000100000000UL, 0x0002000000080000UL, 0x0000000000000004UL, 0
            x0000000000080004UL, 0x0002000000000004UL, 0x0002000100080004UL, 0
            x0002000100000000UL, 0x0000000100080000UL, 0x0002000000000000UL, 0
            x0000000100080004UL, 0x0002000100080000UL, 0x0002000100000004UL, 0
            x0000000100000004UL, 0x0000000000000000UL, 0x0000000000080000UL, 0
            x0002000000080004UL },
        { 0x0000000000010000UL, 0x0000000200000008UL, 0x0004000000000000UL, 0
            x0004000000000008UL, 0x0004000200000000UL, 0x0004000200010008UL, 0
            x0000000200010000UL, 0x0000000000010008UL, 0x0000000200000000UL, 0
            x0004000000010008UL, 0x0000000200010008UL, 0x0004000200010000UL, 0
            x0004000000010000UL, 0x0000000000000000UL, 0x0000000000000008UL, 0
            x0004000200000008UL },
        { 0x0000000000000010UL, 0x0080000000200000UL, 0x0000004000000000UL, 0
            x0080004000000000UL, 0x0000004000200000UL, 0x0080004000200010UL, 0
            x0000000000200010UL, 0x0080000000000010UL, 0x0000000000200000UL, 0
            x0080004000000010UL, 0x0080000000200010UL, 0x0000004000200010UL, 0
            x0000004000000010UL, 0x0000000000000000UL, 0x0080000000000000UL, 0
            x0080004000200000UL },
        { 0x0010000000000000UL, 0x0000008000000020UL, 0x0000000000400000UL, 0
            x0000008000400000UL, 0x0000000000400020UL, 0x0010008000400020UL, 0
            x0010000000000020UL, 0x0010008000000000UL, 0x0000000000000020UL, 0
            x0010008000400000UL, 0x0010008000000020UL, 0x0010000000400020UL, 0
            x0010000000400000UL, 0x0000000000000000UL, 0x0000008000000000UL, 0
            x0000008000400020UL },
        { 0x0000010000000000UL, 0x0020000000800000UL, 0x0000000000000040UL, 0
            x0000000000800040UL, 0x0020000000000040UL, 0x0020010000800040UL, 0
            x0020010000000000UL, 0x0000010000800000UL, 0x0020000000000000UL, 0
            x0000010000800040UL, 0x0020010000800000UL, 0x0020010000000040UL, 0
            x0000010000000040UL, 0x0000000000000000UL, 0x0000000000800000UL, 0
            x0020000000800040UL },
        { 0x0000000000100000UL, 0x0000020000000080UL, 0x0040000000000000UL, 0
            x0040000000000080UL, 0x0040020000000000UL, 0x0040020000100080UL, 0
            x0000020000100000UL, 0x0000000000100080UL, 0x0000020000000000UL, 0
            x0040000000100080UL, 0x0000020000100080UL, 0x0040020000100000UL, 0
            x0040000000100000UL, 0x0000000000000000UL, 0x0000000000000080UL, 0
```

```
            x0040002000000080UL },
        { 0x0000000000000100UL, 0x0800000002000000UL, 0x0000040000000000UL, 0
            x0800040000000000UL, 0x0000040002000000UL, 0x0800040002000100UL, 0
            x0000000002000100UL, 0x0800000000000100UL, 0x0000000002000000UL, 0
            x0800040000000100UL, 0x0800000002000100UL, 0x0000040002000100UL, 0
            x0000040000000100UL, 0x0000000000000000UL, 0x0800000000000000UL, 0
            x0800040002000000UL },
        { 0x0100000000000000UL, 0x0000080000000200UL, 0x0000000004000000UL, 0
            x0000080004000000UL, 0x0000000004000200UL, 0x0100080004000200UL, 0
            x0100000000000200UL, 0x0100080000000000UL, 0x0000000000000200UL, 0
            x0100080004000000UL, 0x0100080000000200UL, 0x0100000004000200UL, 0
            x0100000004000000UL, 0x0000000000000000UL, 0x0000080000000000UL, 0
            x0000080004000200UL },
        { 0x0000010000000000UL, 0x0200000008000000UL, 0x0000000000000400UL, 0
            x0000000008000400UL, 0x0200000000000400UL, 0x0200010008000400UL, 0
            x0200010000000000UL, 0x0000010008000000UL, 0x0200000000000000UL, 0
            x0000010008000400UL, 0x0200010008000000UL, 0x0200010000000400UL, 0
            x0000010000000400UL, 0x0000000000000000UL, 0x0000000008000000UL, 0
            x0200000008000400UL },
        { 0x0000000001000000UL, 0x0000020000000800UL, 0x0400000000000000UL, 0
            x0400000000000800UL, 0x0400020000000000UL, 0x0400020001000800UL, 0
            x0000020001000000UL, 0x0000000001000800UL, 0x0000020000000000UL, 0
            x0400000001000800UL, 0x0000020001000800UL, 0x0400020001000000UL, 0
            x0400000001000000UL, 0x0000000000000000UL, 0x0000000000000800UL, 0
            x0400020000000800UL },
        { 0x0000000000001000UL, 0x8000000020000000UL, 0x0000400000000000UL, 0
            x8000400000000000UL, 0x0000400020000000UL, 0x8000400020001000UL, 0
            x0000000020001000UL, 0x8000000000001000UL, 0x0000000020000000UL, 0
            x8000400000001000UL, 0x8000000020001000UL, 0x0000400020001000UL, 0
            x0000400000001000UL, 0x0000000000000000UL, 0x8000000000000000UL, 0
            x8000400020000000UL },
        { 0x1000000000000000UL, 0x0000800000002000UL, 0x0000000040000000UL, 0
            x0000800040000000UL, 0x0000000040002000UL, 0x1000800040002000UL, 0
            x1000000000002000UL, 0x1000800000000000UL, 0x0000000000002000UL, 0
            x1000800040000000UL, 0x1000800000002000UL, 0x1000000040002000UL, 0
            x1000000040000000UL, 0x0000000000000000UL, 0x0000800000000000UL, 0
            x0000800040002000UL },
        { 0x0001000000000000UL, 0x2000000080000000UL, 0x0000000000004000UL, 0
            x0000000080004000UL, 0x2000000000004000UL, 0x2001000080004000UL, 0
            x2001000000000000UL, 0x0001000080000000UL, 0x2000000000000000UL, 0
            x0001000080004000UL, 0x2001000080000000UL, 0x2001000000004000UL, 0
            x0001000000004000UL, 0x0000000000000000UL, 0x0000000080000000UL, 0
            x2000000080004000UL },
        { 0x0000000010000000UL, 0x0002000000008000UL, 0x4000000000000000UL, 0
            x4000000000008000UL, 0x4002000000000000UL, 0x4002000010008000UL, 0
            x0002000010000000UL, 0x0000000010008000UL, 0x0002000000000000UL, 0
            x4000000010008000UL, 0x0002000010008000UL, 0x4002000010000000UL, 0
            x4000000010000000UL, 0x0000000000000000UL, 0x0000000000008000UL, 0
            x4002000000008000UL }
};

void gift_64_table_generate_round_keys(uint64_t round_keys[restrict
    ROUNDS_GIFT_64],
                                       const uint64_t key[restrict 2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
```

```
                int v = (key_state[0] >> 0 ) & 0xffff;
                int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                round_keys[round] = 0UL;
                for (size_t i = 0; i < 16; i++) {
                        int key_bit_v   = (v >> i)  & 0x1;
                        int key_bit_u   = (u >> i)  & 0x1;
                        round_keys[round] ^= (uint64_t)key_bit_v << (i * 4 + 0);
                        round_keys[round] ^= (uint64_t)key_bit_u << (i * 4 + 1);
                }

                // add single bit
                round_keys[round] ^= 1UL << 63;

                // add round constants
                round_keys[round] ^= ((round_constant[round] >> 0) & 0x1) << 3;
                round_keys[round] ^= ((round_constant[round] >> 1) & 0x1) << 7;
                round_keys[round] ^= ((round_constant[round] >> 2) & 0x1) << 11;
                round_keys[round] ^= ((round_constant[round] >> 3) & 0x1) << 15;
                round_keys[round] ^= ((round_constant[round] >> 4) & 0x1) << 19;
                round_keys[round] ^= ((round_constant[round] >> 5) & 0x1) << 23;

                // update key state
                int k0 = (key_state[0] >> 0 ) & 0xffffUL;
                int k1 = (key_state[0] >> 16) & 0xffffUL;
                k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
                k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
                key_state[0] >>= 32;
                key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
                key_state[1] >>= 32;
                key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
        }
}

uint64_t gift_64_table_subperm(const uint64_t cipher_state)
{
        uint64_t new_cipher_state = 0;

        for (size_t i = 0; i < 16; i++) {
                int nibble = (cipher_state >> (i * 4)) & 0xf;
                new_cipher_state ^= tables[i][nibble];
        }

        return new_cipher_state;
}

uint64_t gift_64_table_encrypt(const uint64_t m, const uint64_t key[restrict 2])
{
        uint64_t c = m;

        // generate round keys
        uint64_t round_keys[ROUNDS_GIFT_64];
        gift_64_table_generate_round_keys(round_keys, key);

        // round loop
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
```

```
                c = gift_64_table_subperm(c);
                c ^= round_keys[round];
        }

        return c;
}
```

A.1.2 Using **vperm**

Listing A.3: `gift_vec_sbox.h`

```
#ifndef GIFT_VEC_SBOX_H
#define GIFT_VEC_SBOX_H

#include <stdint.h>
#include <arm_neon.h>

#define ROUNDS_GIFT_64 28

// expose for benchmarking
uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state);
uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state);
uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state);
uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state);
void       gift_64_vec_sbox_generate_round_keys(uint8x16_t round_keys[
    ROUNDS_GIFT_64],
                                                const uint64_t key[2]);

// construct tables
void gift_64_vec_sbox_init(void);

uint64_t gift_64_vec_sbox_encrypt(const uint64_t m, const uint64_t key[2]);
uint64_t gift_64_vec_sbox_decrypt(const uint64_t c, const uint64_t key[2]);

#endif
```

Listing A.4: `gift_vec_sbox.c`

```
#include "gift_vec_sbox.h"

#include <stdint.h>
#include <arm_neon.h>
#include <string.h>

static uint64_t sbox_vec_u64[2] = {
        0x09030f060c040a01UL, 0x0e080005070b0d02UL
};

static uint64_t sbox_vec_inv_u64[2] = {
0x0b040c020608000dUL, 0x050f09030a01070eUL
};

static uint8x16_t sbox_vec;
```

```
static uint8x16_t sbox_vec_inv;

#define U64_TO_V128(V,M)\
        V = vsetq_lane_u64(\
        (uint64_t)((M >> 4 * 0) & 0xf) << 8 * 0 |\
        (uint64_t)((M >> 4 * 1) & 0xf) << 8 * 1 |\
        (uint64_t)((M >> 4 * 2) & 0xf) << 8 * 2 |\
        (uint64_t)((M >> 4 * 3) & 0xf) << 8 * 3 |\
        (uint64_t)((M >> 4 * 4) & 0xf) << 8 * 4 |\
        (uint64_t)((M >> 4 * 5) & 0xf) << 8 * 5 |\
        (uint64_t)((M >> 4 * 6) & 0xf) << 8 * 6 |\
        (uint64_t)((M >> 4 * 7) & 0xf) << 8 * 7,\
        V, 0);\
        V = vsetq_lane_u64(\
        (uint64_t)((M >> 4 * 8)  & 0xf) << 8 * 0 |\
        (uint64_t)((M >> 4 * 9)  & 0xf) << 8 * 1 |\
        (uint64_t)((M >> 4 * 10) & 0xf) << 8 * 2 |\
        (uint64_t)((M >> 4 * 11) & 0xf) << 8 * 3 |\
        (uint64_t)((M >> 4 * 12) & 0xf) << 8 * 4 |\
        (uint64_t)((M >> 4 * 13) & 0xf) << 8 * 5 |\
        (uint64_t)((M >> 4 * 14) & 0xf) << 8 * 6 |\
        (uint64_t)((M >> 4 * 15) & 0xf) << 8 * 7,\
        V, 1);

static const size_t perm_64[] = {
        0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3,
        4, 21, 38, 55, 52, 5, 22, 39, 36, 53, 6, 23, 20, 37, 54, 7,
        8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
        12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
};

static const size_t perm_64_inv[] = {
        0, 5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
        12, 1, 6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
        8, 13, 2, 7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
        4, 9, 14, 3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51
};

static const int round_constant[] = {
        // rounds 0-15
        0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E,
        // rounds 16-31
        0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
        // rounds 32-47
        0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04
};

uint8x16_t gift_64_vec_sbox_subcells(const uint8x16_t cipher_state)
{
        return vqtbl1q_u8(sbox_vec, cipher_state);
}

uint8x16_t gift_64_vec_sbox_subcells_inv(const uint8x16_t cipher_state)
{
```

```
        return vqtbl1q_u8(sbox_vec_inv, cipher_state);
}

uint8x16_t gift_64_vec_sbox_permute(const uint8x16_t cipher_state)
{
        // collect into 64-bit register (faster)
        uint64_t new_cipher_state = 0UL;

        // S-box 0-7
        uint64_t boxes = vgetq_lane_u64(cipher_state, 0);
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        int bit = (boxes >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64[box * 4 + i
                            ];
                }
        }

        // S-box 8-15
        boxes = vgetq_lane_u64(cipher_state, 1);
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        int bit = (boxes >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64[(box + 8) *
                            4 + i];
                }
        }

        uint8x16_t ret;
        U64_TO_V128(ret, new_cipher_state);

        return ret;
}

uint8x16_t gift_64_vec_sbox_permute_inv(const uint8x16_t cipher_state)
{
        // collect into 64-bit register (faster)
        uint64_t new_cipher_state = 0;

        // S-box 0-7
        uint64_t boxes = vgetq_lane_u64(cipher_state, 0);
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        int bit = (boxes >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64_inv[box * 4
                            + i];
                }
        }

        // S-box 8-15
        boxes = vgetq_lane_u64(cipher_state, 1);
        for (size_t box = 0; box < 8; box++) {
                for (size_t i = 0; i < 4; i++) {
                        int bit = (boxes >> (box * 8 + i)) & 0x1;
                        new_cipher_state |= (uint64_t)bit << perm_64_inv[(box +
                            8) * 4 + i];
                }
```

```
        }

        uint8x16_t ret;
        U64_TO_V128(ret, new_cipher_state);

        return ret;
}

void gift_64_vec_sbox_generate_round_keys(uint8x16_t round_keys[ROUNDS_GIFT_64],
                                          const uint64_t key[2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                int v = (key_state[0] >> 0 ) & 0xffff;
                int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                uint64_t round_key = 0UL;
                for (size_t i = 0; i < 16; i++) {
                        int key_bit_v   = (v >> i)  & 0x1;
                        int key_bit_u   = (u >> i)  & 0x1;
                        round_key ^= (uint64_t)key_bit_v << (i * 4 + 0);
                        round_key ^= (uint64_t)key_bit_u << (i * 4 + 1);
                }

                // add single bit
                round_key ^= 1UL << 63;

                // add round constants
                round_key ^= ((round_constant[round] >> 0) & 0x1) << 3;
                round_key ^= ((round_constant[round] >> 1) & 0x1) << 7;
                round_key ^= ((round_constant[round] >> 2) & 0x1) << 11;
                round_key ^= ((round_constant[round] >> 3) & 0x1) << 15;
                round_key ^= ((round_constant[round] >> 4) & 0x1) << 19;
                round_key ^= ((round_constant[round] >> 5) & 0x1) << 23;

                // pack into vector register
                U64_TO_V128(round_keys[round], round_key)

                // update key state
                int k0 = (key_state[0] >> 0 ) & 0xffffUL;
                int k1 = (key_state[0] >> 16) & 0xffffUL;
                k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
                k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
                key_state[0] >>= 32;
                key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
                key_state[1] >>= 32;
                key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
        }
}

void gift_64_vec_sbox_init(void)
{
        // construct sbox_vec
        sbox_vec = vld1q_u64(sbox_vec_u64);

        // construct sbox_vec_inv
```

```c
	sbox_vec_inv = vld1q_u64(sbox_vec_inv_u64);
}

uint64_t gift_64_vec_sbox_encrypt(const uint64_t m, const uint64_t key[2])
{
	// pack into vector register
	uint8x16_t c;
	U64_TO_V128(c, m);

	// generate round keys
	uint8x16_t round_keys[ROUNDS_GIFT_64];
	gift_64_vec_sbox_generate_round_keys(round_keys, key);

	// round loop
	for (int round = 0; round < ROUNDS_GIFT_64; round++) {
		c = gift_64_vec_sbox_subcells(c);
		c = gift_64_vec_sbox_permute(c);
		c = veorq_u8(c, round_keys[round]);
	}

	// unpack
	uint64_t ret = 0UL;
	ret |= (uint64_t)vgetq_lane_u8(c, 0) << 4 * 0;
	ret |= (uint64_t)vgetq_lane_u8(c, 1) << 4 * 1;
	ret |= (uint64_t)vgetq_lane_u8(c, 2) << 4 * 2;
	ret |= (uint64_t)vgetq_lane_u8(c, 3) << 4 * 3;
	ret |= (uint64_t)vgetq_lane_u8(c, 4) << 4 * 4;
	ret |= (uint64_t)vgetq_lane_u8(c, 5) << 4 * 5;
	ret |= (uint64_t)vgetq_lane_u8(c, 6) << 4 * 6;
	ret |= (uint64_t)vgetq_lane_u8(c, 7) << 4 * 7;
	ret |= (uint64_t)vgetq_lane_u8(c, 8) << 4 * 8;
	ret |= (uint64_t)vgetq_lane_u8(c, 9) << 4 * 9;
	ret |= (uint64_t)vgetq_lane_u8(c, 10) << 4 * 10;
	ret |= (uint64_t)vgetq_lane_u8(c, 11) << 4 * 11;
	ret |= (uint64_t)vgetq_lane_u8(c, 12) << 4 * 12;
	ret |= (uint64_t)vgetq_lane_u8(c, 13) << 4 * 13;
	ret |= (uint64_t)vgetq_lane_u8(c, 14) << 4 * 14;
	ret |= (uint64_t)vgetq_lane_u8(c, 15) << 4 * 15;

	return ret;
}

uint64_t gift_64_vec_sbox_decrypt(const uint64_t c, const uint64_t key[2])
{
	// pack into vector register
	uint8x16_t m;
	U64_TO_V128(m, c);

	// generate round keys
	uint8x16_t round_keys[ROUNDS_GIFT_64];
	gift_64_vec_sbox_generate_round_keys(round_keys, key);

	// round loop (in reverse)
	for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
		m = veorq_u8(m, round_keys[round]);
		m = gift_64_vec_sbox_permute_inv(m);
		m = gift_64_vec_sbox_subcells_inv(m);
```

```
        }

        // unpack
        uint64_t ret = 0UL;
        ret |= (uint64_t)vgetq_lane_u8(m, 0) << 4 * 0;
        ret |= (uint64_t)vgetq_lane_u8(m, 1) << 4 * 1;
        ret |= (uint64_t)vgetq_lane_u8(m, 2) << 4 * 2;
        ret |= (uint64_t)vgetq_lane_u8(m, 3) << 4 * 3;
        ret |= (uint64_t)vgetq_lane_u8(m, 4) << 4 * 4;
        ret |= (uint64_t)vgetq_lane_u8(m, 5) << 4 * 5;
        ret |= (uint64_t)vgetq_lane_u8(m, 6) << 4 * 6;
        ret |= (uint64_t)vgetq_lane_u8(m, 7) << 4 * 7;
        ret |= (uint64_t)vgetq_lane_u8(m, 8) << 4 * 8;
        ret |= (uint64_t)vgetq_lane_u8(m, 9) << 4 * 9;
        ret |= (uint64_t)vgetq_lane_u8(m, 10) << 4 * 10;
        ret |= (uint64_t)vgetq_lane_u8(m, 11) << 4 * 11;
        ret |= (uint64_t)vgetq_lane_u8(m, 12) << 4 * 12;
        ret |= (uint64_t)vgetq_lane_u8(m, 13) << 4 * 13;
        ret |= (uint64_t)vgetq_lane_u8(m, 14) << 4 * 14;
        ret |= (uint64_t)vgetq_lane_u8(m, 15) << 4 * 15;

        return ret;
}
```

A.1.3 Bitslicing

Listing A.5: `gift_vec_sliced.h`

```
#ifndef GIFT_VEC_SLICED_H
#define GIFT_VEC_SLICED_H

#include <stdint.h>
#include <arm_neon.h>

#define ROUNDS_GIFT_64 28

// expose for benchmarking
uint8x16_t shl(uint8x16_t v, int n);
uint8x16_t shr(uint8x16_t v, int n);
void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b,
    uint8x16_t m, int n);
void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2]);
void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2]);

void gift_64_vec_sliced_subcells(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_permute(uint8x16x4_t cipher_state[restrict 2]);
void gift_64_vec_sliced_permute_inv(uint8x16x4_t cipher_state[2]);
void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t round_keys[restrict
    ROUNDS_GIFT_64][2],
                                            const uint64_t key[restrict 2]);

void gift_64_vec_sliced_init(void);

```

```
void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
                                const uint64_t m[restrict 16],
                                const uint64_t key[restrict 2]);
void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
                                const uint64_t c[restrict 16],
                                const uint64_t key[restrict 2]);

#endif
```

Listing A.6: `gift_vec_sliced.c`

```
#include "gift_vec_sliced.h"

#include <stddef.h>
#include <stdio.h>
#include <string.h>

static uint64_t pack_shf_u64[4] = {
        0x1303120211011000UL, 0x1707160615051404UL, // S0/S1/S2/S3
        0x1b0b1a0a19091808UL, 0x1f0f1e0e1d0d1c0cUL, // S4/S5/S6/S7
};

static uint64_t pack_shf_inv_u64[4] = {
        0x0e0c0a0806040200UL, 0x1e1c1a1816141210UL, // S0/S1/S2/S3
        0x0f0d0b0907050301UL, 0x1f1d1b1917151311UL, // S4/S5/S6/S7
};

static uint64_t perm_u64[8] = {
        0x0f0b07030c080400UL, 0x0d0905010e0a0602UL, // S0/S4
        0x0c0804000d090501UL, 0x0e0a06020f0b0703UL, // S1/S5
        0x0d0905010e0a0602UL, 0x0f0b07030c080400UL, // S2/S6
        0x0e0a06020f0b0703UL, 0x0c0804000d090501UL  // S3/S7
};


static uint64_t perm_inv_u64[8] = {
        0x05090d0104080c00UL, 0x070b0f03060a0e02UL, // S0/S4
        0x090d0105080c0004UL, 0x0b0f03070a0e0206UL, // S1/S5
        0x0d0105090c000408UL, 0x0f03070b0e02060aUL, // S2/S6
        0x0105090d0004080cUL, 0x03070b0f02060a0eUL  // S3/S7
};

static uint8x16x2_t pack_shf;
static uint8x16x2_t pack_shf_inv;
static uint8x16x4_t perm;
static uint8x16x4_t perm_inv;

static uint8x16_t pack_mask_0;
static uint8x16_t pack_mask_1;
static uint8x16_t pack_mask_2;

static const int round_constant[] = {
        // rounds 0-15
        0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0
            x39, 0x33, 0x27, 0x0E,
        // rounds 16-31
```

```
        0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
        // rounds 32-47
        0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04
};

uint8x16_t shl(uint8x16_t v, int n)
{
        uint64_t l[2];
        vst1q_u64(l, v);
        l[1] = (l[1] << n) | (l[0] >> (64 - n));
        l[0] <<= n;
        return vreinterpretq_u8_u64(vld1q_u64(l));
}

uint8x16_t shr(uint8x16_t v, int n)
{
        uint64_t l[2];
        vst1q_u64(l, v);
        l[0] = l[0] >> n | (((l[1] << (64 - n)) >> (64 - n)) << (64 - n));
        l[1] >>= n;
        return vreinterpretq_u8_u64(vld1q_u64(l));
}

void gift_64_vec_sliced_swapmove(uint8x16_t *restrict a, uint8x16_t *restrict b, uint8x16_t m, int n)
{

        uint8x16_t t = vandq_u8(veorq_u8(shr(*a, n), *b), m);
        *b = veorq_u8(*b, t);
        *a = veorq_u8(*a, shl(t, n));
}

void gift_64_vec_sliced_bits_pack(uint8x16x4_t m[restrict 2])
{
        // take care not to shift mask bits out of the register
        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[0].val[2], &m[0].val[3], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[1].val[2], &m[1].val[3], pack_mask_0, 1);

        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[0].val[1], &m[0].val[3], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[1].val[1], &m[1].val[3], pack_mask_1, 2);

        // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[1].val[0], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[2], &m[1].val[2], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[1], &m[1].val[1], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[3], &m[1].val[3], pack_mask_2, 4);

        // same plaintext slice bits into same register (so we only have to do
        // what we are doing here once instead of every round)
        uint8x16x2_t pairs[4] = {
                { .val = { m[0].val[0], m[1].val[0] }},
```

```
                { .val = { m[0].val[1], m[1].val[1] }},
                { .val = { m[0].val[2], m[1].val[2] }},
                { .val = { m[0].val[3], m[1].val[3] }},
        };

        m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[0]);
        m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[0]);
        m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[0]);
        m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[0]);

        m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf.val[1]);
        m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf.val[1]);
        m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf.val[1]);
        m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf.val[1]);
}

void gift_64_vec_sliced_bits_unpack(uint8x16x4_t m[restrict 2])
{
        // same plaintext slice bits into same register (so we only have to do
        // what we are doing here once instead of every round)
        uint8x16x2_t pairs[4] = {
                { .val = { m[0].val[0], m[1].val[0] }},
                { .val = { m[0].val[1], m[1].val[1] }},
                { .val = { m[0].val[2], m[1].val[2] }},
                { .val = { m[0].val[3], m[1].val[3] }},
        };

        m[0].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[0]);
        m[0].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[0]);
        m[0].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[0]);
        m[0].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[0]);

        m[1].val[0] = vqtbl2q_u8(pairs[0], pack_shf_inv.val[1]);
        m[1].val[1] = vqtbl2q_u8(pairs[1], pack_shf_inv.val[1]);
        m[1].val[2] = vqtbl2q_u8(pairs[2], pack_shf_inv.val[1]);
        m[1].val[3] = vqtbl2q_u8(pairs[3], pack_shf_inv.val[1]);

        // take care not to shift mask bits out of the register
        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[0].val[2], &m[0].val[3], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[1], pack_mask_0, 1);
        gift_64_vec_sliced_swapmove(&m[1].val[2], &m[1].val[3], pack_mask_0, 1);

        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[0].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[0].val[1], &m[0].val[3], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[1].val[0], &m[1].val[2], pack_mask_1, 2);
        gift_64_vec_sliced_swapmove(&m[1].val[1], &m[1].val[3], pack_mask_1, 2);

        // make bytes (a0 b0 c0 d0 a4 b4 c4 d4 -> a0 b0 c0 d0 e0 f0 g0 h0)
        gift_64_vec_sliced_swapmove(&m[0].val[0], &m[1].val[0], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[2], &m[1].val[2], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[1], &m[1].val[1], pack_mask_2, 4);
        gift_64_vec_sliced_swapmove(&m[0].val[3], &m[1].val[3], pack_mask_2, 4);
}

void gift_64_vec_sliced_subcells(uint8x16x4_t cs[restrict 2])
{
```

```
        cs[0].val[1] = veorq_u8(cs[0].val[1], vandq_u8(cs[0].val[0], cs[0].val[2]));
        uint8x16_t t = veorq_u8(cs[0].val[0], vandq_u8(cs[0].val[1], cs[0].val[3]));
        cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
        cs[0].val[0] = veorq_u8(cs[0].val[3], cs[0].val[2]);
        cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
        cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
        cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
        cs[0].val[3] = t;

        cs[1].val[1] = veorq_u8(cs[1].val[1], vandq_u8(cs[1].val[0], cs[1].val[2]));
        t            = veorq_u8(cs[1].val[0], vandq_u8(cs[1].val[1], cs[1].val[3]));
        cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
        cs[1].val[0] = veorq_u8(cs[1].val[3], cs[1].val[2]);
        cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
        cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
        cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
        cs[1].val[3] = t;
}

void gift_64_vec_sliced_subcells_inv(uint8x16x4_t cs[restrict 2])
{
        uint8x16_t t = cs[0].val[3];
        cs[0].val[2] = veorq_u8(cs[0].val[2], vandq_u8(t, cs[0].val[1]));
        cs[0].val[0] = vmvnq_u8(cs[0].val[0]);
        cs[0].val[1] = veorq_u8(cs[0].val[1], cs[0].val[0]);
        cs[0].val[3] = veorq_u8(cs[0].val[0], cs[0].val[2]);
        cs[0].val[2] = veorq_u8(cs[0].val[2], vorrq_u8(t, cs[0].val[1]));
        cs[0].val[0] = veorq_u8(t, vandq_u8(cs[0].val[1], cs[0].val[3]));
        cs[0].val[1] = veorq_u8(cs[0].val[1], vandq_u8(cs[0].val[0], cs[0].val[2]));

        t            = cs[1].val[3];
        cs[1].val[2] = veorq_u8(cs[1].val[2], vandq_u8(t, cs[1].val[1]));
        cs[1].val[0] = vmvnq_u8(cs[1].val[0]);
        cs[1].val[1] = veorq_u8(cs[1].val[1], cs[1].val[0]);
        cs[1].val[3] = veorq_u8(cs[1].val[0], cs[1].val[2]);
        cs[1].val[2] = veorq_u8(cs[1].val[2], vorrq_u8(t, cs[1].val[1]));
        cs[1].val[0] = veorq_u8(t, vandq_u8(cs[1].val[1], cs[1].val[3]));
        cs[1].val[1] = veorq_u8(cs[1].val[1], vandq_u8(cs[1].val[0], cs[1].val[2]));
}

void gift_64_vec_sliced_permute(uint8x16x4_t cs[restrict 2])
{
        cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm.val[0]);
        cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm.val[1]);
        cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm.val[2]);
        cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm.val[3]);

        cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm.val[0]);
        cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm.val[1]);
        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm.val[2]);
        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm.val[3]);
```

```
}

void gift_64_vec_sliced_permute_inv(uint8x16x4_t cs[restrict 2])
{
        cs[0].val[0] = vqtbl1q_u8(cs[0].val[0], perm_inv.val[0]);
        cs[0].val[1] = vqtbl1q_u8(cs[0].val[1], perm_inv.val[1]);
        cs[0].val[2] = vqtbl1q_u8(cs[0].val[2], perm_inv.val[2]);
        cs[0].val[3] = vqtbl1q_u8(cs[0].val[3], perm_inv.val[3]);

        cs[1].val[0] = vqtbl1q_u8(cs[1].val[0], perm_inv.val[0]);
        cs[1].val[1] = vqtbl1q_u8(cs[1].val[1], perm_inv.val[1]);
        cs[1].val[2] = vqtbl1q_u8(cs[1].val[2], perm_inv.val[2]);
        cs[1].val[3] = vqtbl1q_u8(cs[1].val[3], perm_inv.val[3]);
}

void gift_64_vec_sliced_generate_round_keys(uint8x16x4_t round_keys[restrict ROUNDS_GIFT_64][2],
                                            const uint64_t key[restrict 2])
{
        uint64_t key_state[] = {key[0], key[1]};
        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                int v = (key_state[0] >> 0 ) & 0xffff;
                int u = (key_state[0] >> 16) & 0xffff;

                // add round key (RK=U||V)
                // (slice 2 stays unused)
                uint64_t round_key[6] = { 0x0UL };
                for (size_t i = 0; i < 8; i++) {
                        int key_bit_v   = (v >> (i + 0)) & 0x1;
                        int key_bit_u   = (u >> (i + 0)) & 0x1;
                        round_key[0]    ^= (uint64_t)key_bit_v << (i * 8);
                        round_key[2]    ^= (uint64_t)key_bit_u << (i * 8);

                        key_bit_v       = (v >> (i + 8)) & 0x1;
                        key_bit_u       = (u >> (i + 8)) & 0x1;
                        round_key[1]    ^= (uint64_t)key_bit_v << (i * 8);
                        round_key[3]    ^= (uint64_t)key_bit_u << (i * 8);
                }

                // add single bit
                round_key[5] ^= 1UL << (7 * 8);

                // add round constants
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 0) & 0x1) << (0 * 8);
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 1) & 0x1) << (1 * 8);
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 2) & 0x1) << (2 * 8);
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 3) & 0x1) << (3 * 8);
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 4) & 0x1) << (4 * 8);
                round_key[4] ^= ((uint64_t)(round_constant[round] >> 5) & 0x1) << (5 * 8);

                // extend bits to bytes
```

```
		for (size_t i = 0; i < 6; i++) {
			round_key[i] |= round_key[i] << 1;
			round_key[i] |= round_key[i] << 2;
			round_key[i] |= round_key[i] << 4;
		}

		round_keys[round][0].val[0] = vsetq_lane_u64(round_key[0],
		    round_keys[round][0].val[0], 0);
		round_keys[round][0].val[0] = vsetq_lane_u64(round_key[1],
		    round_keys[round][0].val[0], 0);
		round_keys[round][0].val[1] = vsetq_lane_u64(round_key[2],
		    round_keys[round][0].val[1], 0);
		round_keys[round][0].val[1] = vsetq_lane_u64(round_key[3],
		    round_keys[round][0].val[1], 0);
		round_keys[round][0].val[2] = vdupq_n_u8(0);
		round_keys[round][0].val[3] = vsetq_lane_u64(round_key[4],
		    round_keys[round][0].val[3], 0);
		round_keys[round][0].val[3] = vsetq_lane_u64(round_key[5],
		    round_keys[round][0].val[3], 0);
		round_keys[round][1]        = round_keys[round][0];

		// update key state
		int k0 = (key_state[0] >> 0 ) & 0xffffUL;
		int k1 = (key_state[0] >> 16) & 0xffffUL;
		k0 = (k0 >> 12) | ((k0 & 0xfff) << 4);
		k1 = (k1 >> 2 ) | ((k1 & 0x3  ) << 14);
		key_state[0] >>= 32;
		key_state[0] |= (key_state[1] & 0xffffffffUL) << 32;
		key_state[1] >>= 32;
		key_state[1] |= ((uint64_t)k0 << 32) | ((uint64_t)k1 << 48);
	}
}

void gift_64_vec_sliced_init(void)
{
	// bit packing shuffle
	pack_shf = vld1q_u8_x2((uint8_t*)&pack_shf_u64[0]);

	// inverse bit packing shuffle
	pack_shf_inv = vld1q_u8_x2((uint8_t*)&pack_shf_inv_u64[0]);

	// permutations
	perm = vld1q_u8_x4((uint8_t*)&perm_u64[0]);

	// inverse permutations
	perm_inv = vld1q_u8_x4((uint8_t*)&perm_inv_u64[0]);

	// packing masks
	pack_mask_0 = vdupq_n_u8(0x55);
	pack_mask_1 = vdupq_n_u8(0x33);
	pack_mask_2 = vdupq_n_u8(0x0f);
}

void gift_64_vec_sliced_encrypt(uint64_t c[restrict 16],
                                const uint64_t m[restrict 16],
                                const uint64_t key[restrict 2])
{
```

```
        uint8x16x4_t s[2];
        s[0] = vld1q_u8_x4((uint8_t*)&m[0]);
        s[1] = vld1q_u8_x4((uint8_t*)&m[8]);
        gift_64_vec_sliced_bits_pack(s);

        uint8x16x4_t round_keys[ROUNDS_GIFT_64][2];
        gift_64_vec_sliced_generate_round_keys(round_keys, key);

        for (int round = 0; round < ROUNDS_GIFT_64; round++) {
                gift_64_vec_sliced_subcells(s);
                gift_64_vec_sliced_permute(s);

                s[0].val[0] = veorq_u8(s[0].val[0], round_keys[round][0].val[0]);
                s[0].val[1] = veorq_u8(s[0].val[1], round_keys[round][0].val[1]);
                s[0].val[2] = veorq_u8(s[0].val[2], round_keys[round][0].val[2]);
                s[0].val[3] = veorq_u8(s[0].val[3], round_keys[round][0].val[3]);
                s[1].val[0] = veorq_u8(s[1].val[0], round_keys[round][1].val[0]);
                s[1].val[1] = veorq_u8(s[1].val[1], round_keys[round][1].val[1]);
                s[1].val[2] = veorq_u8(s[1].val[2], round_keys[round][1].val[2]);
                s[1].val[3] = veorq_u8(s[1].val[3], round_keys[round][1].val[3]);
        }

        gift_64_vec_sliced_bits_unpack(s);
        vst1q_u8_x4((uint8_t*)&c[0], s[0]);
        vst1q_u8_x4((uint8_t*)&c[8], s[1]);
}

void gift_64_vec_sliced_decrypt(uint64_t m[restrict 16],
                                const uint64_t c[restrict 16],
                                const uint64_t key[restrict 2])
{
        uint8x16x4_t s[2];
        s[0] = vld1q_u8_x4((uint8_t*)&c[0]);
        s[1] = vld1q_u8_x4((uint8_t*)&c[8]);
        gift_64_vec_sliced_bits_pack(s);

        uint8x16x4_t round_keys[ROUNDS_GIFT_64][2];
        gift_64_vec_sliced_generate_round_keys(round_keys, key);

        for (int round = ROUNDS_GIFT_64 - 1; round >= 0; round--) {
                s[0].val[0] = veorq_u8(s[0].val[0], round_keys[round][0].val[0]);
                s[0].val[1] = veorq_u8(s[0].val[1], round_keys[round][0].val[1]);
                s[0].val[2] = veorq_u8(s[0].val[2], round_keys[round][0].val[2]);
                s[0].val[3] = veorq_u8(s[0].val[3], round_keys[round][0].val[3]);
                s[1].val[0] = veorq_u8(s[1].val[0], round_keys[round][1].val[0]);
                s[1].val[1] = veorq_u8(s[1].val[1], round_keys[round][1].val[1]);
                s[1].val[2] = veorq_u8(s[1].val[2], round_keys[round][1].val[2]);
                s[1].val[3] = veorq_u8(s[1].val[3], round_keys[round][1].val[3]);

                gift_64_vec_sliced_permute_inv(s);
                gift_64_vec_sliced_subcells_inv(s);
        }

        gift_64_vec_sliced_bits_unpack(s);
        vst1q_u8_x4((uint8_t*)&m[0], s[0]);
        vst1q_u8_x4((uint8_t*)&m[8], s[1]);
}
```

Appendix B

Lorem dolor

Bibliography

[1] Subhadeep Banik et al. "GIFT: A Small Present". In: Aug. 2017, pp. 321–345. ISBN: 978-3-319-66786-7. DOI: 10.1007/978-3-319-66787-4_16.

[2] Ryad Benadjila et al. "Implementing Lightweight Block Ciphers on x86 Architectures". In: Selected Areas in Cryptography – SAC 2013. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisoněk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 324–351. ISBN: 978-3-662-43414-7.

[3] ARM Limited. Arm Neon Intrinsics Reference. 2022.