IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2020, No. 1, pp. 185–232. DOI:10.13154/tosc.v2020.i1.185-232

# Swap and Rotate: Lightweight Linear Layers for SPN-based Blockciphers

Subhadeep Banik[1], Fatih Balli[1], Francesco Regazzoni[2] and Serge Vaudenay[1]

[1] Security and Cryptography Laboratory (LASEC), École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, {subhadeep.banik,fatih.balli,serge.vaudenay}@epfl.ch
[2] Advanced Learning and Research Institute (ALaRI), University of Lugano, Lugano, Switzerland. regazzoni@alari.ch

**Abstract.** In CHES 2017, Jean et al. presented a paper on "Bit-Sliding" in which the authors proposed lightweight constructions for SPN based block ciphers like AES, PRESENT and SKINNY. The main idea behind these constructions was to reduce the length of the datapath to 1 bit and to reformulate the linear layer for these ciphers so that they require fewer scan flip-flops (which have built-in multiplexer functionality and so larger in area as compared to a simple flip-flop). In this paper, we develop their idea even further in few separate directions.

First, we prove that given an arbitrary linear transformation, it is always possible to construct the linear layer using merely 2 scan flip-flops. This points to an optimistic venue to follow to gain further GE reductions, yet the straightforward application of the techniques in our proof to PRESENT and GIFT leads to inefficient implementations of the linear layer, as reducing ourselves to 2 scan flip-flops setting requires thousands of clock cycles and leads to very high latency.

Equipped with the well-established formalism on permutation groups, we explore whether we can reduce the number of clock cycles to a practical level, i.e. few hundreds, by adding few more pairs of scan flip flops. For PRESENT, we show that 4 (resp. 8, 12) scan flip-flops are sufficient to complete the permutation layer in 384 (resp. 256, 128) clock cycles. For GIFT, we show that 4 (resp. 8, 10) scan flip flops correspond to 320 (resp. 192, 128) clock cycles. Finally, in order to provide the best of the two worlds (i.e. circuit area and latency), we push our scan flip-flop choices even further to completely eliminate the latency incurred by the permutation layer, without compromising our stringent GE budget. We show that not only 12 scan flip flops are sufficient to execute PRESENT permutation in 64 clock cycles, but also the same scan flip flops can be used readily in a combined encryption decryption circuit. Our final design of PRESENT and GIFT beat the record of Jean et al. and Banik et al. in both latency and in circuit-size metric. We believe that the techniques presented in our work can also be used at choosing bit-sliding-friendly linear layer permutations for the future SPN-based designs.

**Keywords:** Lightweight circuit, PRESENT, GIFT, FLIP

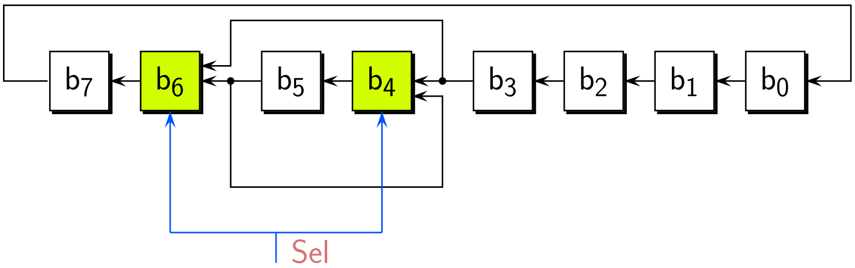
# 1 Introduction

The block cipher family Katan [CDK09] (whose precursor was the stream cipher Trivium [CP08]) and then later Simon [BSS$^+$] were in some sense aimed to achieve a lower limit of lightweight encryption in terms of area occupied in silicon. Both these ciphers have shift register based update functions, which is efficient to implement in ASIC when the length of datapath is reduced to one bit. In CHES 2017, Jean et al. presented the concept of "Bit-Sliding" [JMPS17] in which byte and nibble oriented block ciphers like AES [DR02], PRESENT [BKL$^+$07] and SKINNY [BJK$^+$16] were implemented in hardware by updating

Received: 2019-11-23, Accepted: 2020-01-23, Published: 2020-05-07

only one bit per clock cycle. This was counter-intuitive because the block ciphers in question used 8/4-bit S-boxes and it was not immediately clear how the width of the data-path could be made smaller than the size of the S-box that the block cipher was employing. The main idea behind these constructions was to reformulate the linear layer for these ciphers so that they require fewer scan flip-flops (which have built-in multiplexer functionality at the input port and so larger in area as compared to a simple flip-flop). In particular, the PRESENT linear layer which is essentially a bit permutation over the state, was decomposed as $P_2^4 \circ P_1$, where $P_1$ was a permutation that operated on each 16-bit block of the 64-bit state and $P_2$ is some other permutation. This decomposition allowed the authors of [JMPS17] to implement the linear layer using only 25 scan flip-flops and 39 regular flip-flops, whereas previous implementations [RPLP08] have required all 64 flip-flops holding the state to have additional multiplexer at its input.

**Motivation:** There has been significant interest in the symmetric cryptology community about efficient design/implementation of linear layers of block ciphers that additionally have the maximum distance separable property [KLSW17, LSL+19, DL18, SS16, CLM16, LW17]. The main aim of investing research effort in this direction is to ensure that the construction of the linear layer is efficient in hardware/software. For example, one of the target metrics of optimization is minimizing the number of xor gates that would allow a hardware implementation of the block cipher occupy less silicon area. Some block ciphers like PRESENT and GIFT use simple bit permutations as their linear layers. These do not require xor gates to construct. However the area of these block ciphers are lowest when implemented serially i.e. in which only a fraction of the state is updated in each clock cycle. Usually the lowest hardware footprint is obtained for implementations that update one bit of the state per clock cycle (a technique made popular in [JMPS17]). Consider the following example: imagine a block cipher with an 8-bit state that employs the following bit permutation function $P_{ex}$ as the linear layer, where $P_{ex}(5) = 4$, $P_{ex}(3) = 6$ and $P_{ex}(i) = i - 1 \bmod 8$ for all other $i$.



Since $P_{ex}$ is a composition of a simple swap and rotate function, it can be efficiently implemented using the circuit above using only 2 scan flip-flops (shown in green). A scan flip-flop is a memory element with additional multiplexer functionality, (that occupies more area than a simple flip-flop). Thus, when the Sel signal is 0, the circuit performs a rotate operation *Rot* (which may be required for performing other block cipher operations like adding round key bits or implementing s-box). However when the Sel signal is 1, then the circuit performs the permutation function $P_{ex}$ on the state. Since $P_{ex}$ is simple, it can be implemented with only 2 scan flip-flops. Since more scan flip-flops require more area to implement, this is an attractive proposition for the lightweight context. However $P_{ex}$ is quite simple, is unlikely to be the linear layer candidate for any real world cipher. For more complicated bit permutations, there are therefore 2 dimensions of optimization. First is to investigate if a more complicated permutation can be implemented using 2 scan flip-flops alone. For example a permutation like $\mathcal{P} = P_{ex} \circ Rot \circ P_{ex}$ can be implemented using the above setup in 3 clock cycles (by setting the Sel signal to 1,0,1 in 3 consecutive cycles). One of the results in group theory states that if $P_{ex}$ is chosen properly, any permutation function can be constructed using only $P_{ex}$ and *Rot* [Con]. However the number of cycles required to do so, (which is exactly equal to the minimum distance of $\mathcal{P}$ in a Cayley graph of the corresponding permutation group with $P_{ex}$ and $Rot$ as the generating nodes) may be very high. Thus, for a more practical approach, a second dimension is investigating if increasing the number of scan flip-flops to 4 or 6, a particular permutation can be implemented efficiently in lower number of cycles, so as to not increase the circuit latency too much. Thus we reformulate the problem of implementing bit permutations in hardware from a circuit-theory perspective to one in algebraic group theory. Note that each pair of additional scan flip flops affords us the ability to implement more direct swap functions on the bits of the plaintext state in a single cycle, thus enabling us to implement complicated permutations in lesser number of cycles. On the flip side, they consume more physical area to implement. Thus the exercise of efficient implementation of bit permutation based linear layers amounts to a balancing act between different type of swap/rotate functions.

Lightweight implementation of cryptosystems is an important application in itself, as numerous papers in literature exist in which area optimization is the main goal [JMPS17, MPL$^{+}$11], even at the expense of latency and energy consumption. Moreover area minimization is crucial in applications like medical implants and passive RFID tags (that typically do not use latest CMOS technology) run on extreme tight area budgets. Moreover reducing the area footprint of a system, invariably reduces the power consumption of the system (since CMOS power consumption is proportional to the number of switching gates) which is also an important optimizable design metric. For example, in implantable devices, power is a more crucial metric, as the wearer certainly can not tolerate any rise in operating temperature as a side effect of high power consumption over a number of cycles. It is however well known that most low area implementations of a block cipher (that are generally serialized) are typically not energy efficient [BBR15]. Although energy is an important metric, our research direction is directed towards applications that can not ignore area and power constraints.

We choose PRESENT and GIFT block ciphers for our analysis, because both employ bit permutations as their linear layer. Apart from this, studying these ciphers is of independent interest because of the importance of these designs in the cryptographic community. PRESENT is currently an ISO/IEC standard and an extremely popular in the security community. GIFT is used as the underlying block cipher in 6 of the 33 candidates in the second round of the NIST lightweight cryptography competition [nis19].

**Contribution/Organization (Salient points):** The contributions in the paper can be summarized in the following salient points.

**A. First Direction** The main idea behind [JMPS17] was that the fewer scan flip-flops one uses to construct the circuit is likely to translate into a lowering of the total hardware area of the circuit. Taking this idea forward, in this paper we try to answer the following question: is it possible to construct the linear layer if only 2 of the 64 flip-flops used to store the state are scan flip-flops? The answer is yes and we can always do this by using results of classical permutation theory [Con]. However, even after applying various optimizations to the preliminary ideas of [Con], the least amount of time required to implement a PRESENT/GIFT round function was 1472/1728 cycles which is still very slow. Since latency is also an important lightweight metric we explore a second direction in which we try to decrease latency by minimally increasing the number scan flip-flops.

In Section 2, after introducing some preliminary definitions and notations (along with a brief sketch of the proofs presented in [Con]) in Section 2, the following Sections 3, 4 and 5 summarize the above ideas. The main mathematical background is developed in Section 3. This section is mainly concerned with the PRESENT block cipher. The theory built up in this section is done in various stages: in each stage we try to decrease the number of permutations required to describe the PRESENT bit permutation. Section 4 contains a circuit level description of the cipher along with a cycle by cycle operational details of its functions. Thereafter we extend these ideas to the GIFT block cipher in Section 5.

**B. Second Direction** Naturally, we then investigate if adding more scan flip-flops to the circuit can significantly reduce the number of cycles/area of the circuit. Intuitively this makes sense because more scan flip-flops allow us to execute more transposition operations on the state register in a single clock cycle and hence it could reduce the total number of cycles to implement the bit permutation layer. This could lead to a much smaller size of control bits required to control swaps and keep the area to a minimum. In fact we found that adding 2 or 4 additional scan flip-flops provides us with a reasonable balance between area and throughput.

As a result of the theoretical foundations built in the paper, we construct lightweight implementations of the PRESENT and GIFT [BPP$^+$17] circuits for both encryption (E) and combined encryption+decryption (ED) modes. Both PRESENT and GIFT are block ciphers in which the linear layer is composed with a bit permutation over the internal state. We increase the number of scan flip-flops gradually from 2 to 4 to 6 and so on and observe the reduction of both area and latency. Our smallest implementation of PRESENT at 694 GE and GIFT at 907 GE are not only the lowest reported in the literature so far, but each is achieved at only 64 clock cycles per round, which is faster than both the 68 cycle/round implementation of PRESENT in [JMPS17] and 96 cycle/round implementation of GIFT in [BPP$^+$17]. In the ED mode, the smallest PRESENT and GIFT circuits occupy 786 GE and 1025 GE which are also the smallest reported thus far. (Note that all circuits have been synthesized with the standard cell library CORE90GPHVT v 2.1.a of the STM 90nm CMOS logic process). In Section 6, we summarize the above ideas and introduce necessary mathematical background and hence construct circuits with smaller area and higher throughput.

**C. The Stream Cipher FLIP** We take the ideas forward and look at the stream cipher FLIP [MJSC16] whose core state update function is also a bit permutation. We propose three circuits for FLIP: the first is a direct implementation of the ideas in [Con]. This version however takes time proportional to the cube of the size of the secret key to produce a single keystream bit and is hence not practical. The second circuit we construct takes quadratic time and occupies only 3581 GE. The third circuit we propose uses slightly different ideas for bit swapping and can achieve the FLIP functionality in linear time. This circuit has an area of around 8605 GE. These are the first reported hardware implementations of FLIP. All the results are tabulated in Table 1. Section 8 concludes the paper.

## 2 Preliminaries

We use the symbol $S_n$ to denote the permutation group on $n$ elements. Naturally we have, $|S_n| = n!$ and the group is non-commutative. A $k$-cycle $\pi \in S_n$ (for $1 \leq k \leq n$) is generally expressed as the $k$-tuple $(i_1, i_2, \ldots, i_k)$ which implies

- $\pi(i_1) = i_2,\ \pi(i_2) = i_3,\ \cdots, \pi(i_k) = i_1$, and
- $\pi(i) = i,\ \forall i \notin \{i_1, i_2, \ldots, i_k\}$.

**Table 1:** Tabulation of Results (unless stated otherwise, power reported at 10 MHz. Total latency refers to number of cycles required to encrypt one block of 64 bits. For completeness, comparison with the most lightweight circuits of a few eStream finalist candidates is also included) [1]: Synthesized using IBM 130nm CMOS process, [2]: Power reported at 100 KHz

| | Design | # Swaps | Area (GE) | Power ($\mu$W) | Latency | | Ref |
|---|---|---|---|---|---|---|---|
| | | | | | Per round (or bit) | Total | |
| 1 | PRESENT (E) | 1 | 943 | 40.0 | 1536 | 47760 | Section 4 |
| | | 6 | 694 | 34.5 | 64 | 2128 | Section 6.5 |
| | | | 847[1] | 0.43[2] | 68 | 2252 | [JMPS17] |
| 2 | PRESENT (ED) | 1 | 1039 | 41.4 | 1536 | 47760 | Section 4 |
| | | 6 | 786 | 34.2 | 64 | 2128 | Section 6.5 |
| | | | 1238 | 56.0 | 17 | 547 | [BBR17b] |
| 3 | GIFT (E) | 1 | 1132 | 49.8 | 1792 | 50304 | Section 5 |
| | | 6 | 907 | 41.3 | 64 | 1920 | Section 6.5 |
| | | | 930 | 35.9 | 96 | 2816 | [BPP+17] |
| 4 | GIFT (ED) | 1 | 1290 | 52.6 | 1792 | 50304 | Section 5 |
| | | 6 | 1055 | 43.6 | 64 | 1920 | Section 6.5 |
| 5 | FLIP | 2nd ckt | 3581 | 164.9 | $\approx 2^{17}$ | $\approx 2^{26}$ | Section 7 |
| | | 3rd ckt | 8605 | 171.9 | 530 | 33920 | Section 7 |
| 6 | Grain v1[HJMM08] | | 1005 | 38.9 | | 225 | |
| 7 | Grain 128[HJMM08] | | 1455 | 57.8 | | 321 | |
| 8 | Trivium[CP08] | | 1584 | 75.6 | | 1217 | |

This is a permutation of order equal to $k$. A transposition (or a swap) $\tau \in S_n$ is a 2-cycle. Denote by $\mathbb{A}_\pi$ the set of *active* elements in the permutation, i.e. $\{i_1, i_2, \ldots, i_k\}$. In general, if $\pi$ is a composition of several cycles of different orders, then define

$$\mathbb{A}_\pi = \{x : \ \pi(x) \neq x\}.$$

The cycles $\pi_1$ and $\pi_2$ of orders $k_1$ and $k_2$ respectively are called disjoint if $\mathbb{A}_{\pi_1}$ and $\mathbb{A}_{\pi_2}$ are disjoint, i.e. have no elements in common. It is easy to see all disjoint cycles commute under the composition operation. It is well known that every permutation in $S_n$ can be expressed as a composition of disjoint $k$-cycles, uniquely up to ordering of the $k$-cycles. To begin discussions, we cite a couple of results from [Con].

**Lemma 1.** **[Con, Theorem 2.1]** *For $n \geq 2$, $S_n$ is generated by its transpositions.*

The above is not particularly difficult to prove. We know that the identity permutation can be written as $\tau^2$ where $\tau$ is any transposition. As stated above, any permutation can be expressed as compositions of $k$-cycles, and any $k$-cycle $(i_1, i_2, \ldots, i_k)$ can be written as $(i_1, i_2) \circ (i_2, i_3) \circ \cdots \circ (i_{k-1}, i_k)$ and so the result follows.

**Lemma 2.** **[Con, Theorem 2.5]** *For $n \geq 2$, $S_n$ is generated by the transposition $(1, 2)$ and the $n$-cycle $(1, 2, \ldots, n)$.*

A rigorous proof of the above lemma may be found in [Con], but for the benefit of the reader we give the sketch idea. First note that the set $G_1 = \{(1, 2),\ (2, 3),\ \cdots, (n-1, n)\}$ also generates $S_n$. That is because any arbitrary transposition $(i, j)$ can be obtained by the composition $(i, i{+}1) \circ (i{+}1, j) \circ (i, i{+}1)$, where the first and third transpositions are already in $G_1$. If $|i{+}1{-}j| > 1$, then $(i{+}1, j)$ can be further written as $(i{+}1, i{+}2) \circ (i{+}2, j) \circ (i{+}1, i{+}2)$, and so on, until the term in the middle is in $G_1$. Given the following identity

$$\pi \circ (i_1, i_2, \ldots, i_k) \circ \pi^{-1} = (\pi(i_1), \pi(i_2), \ldots, \pi(i_k)),$$

for all $k$-cycles and $\pi \in S_n$, it is possible to show that any transposition of the form $(i, i+1)$ can be generated by $(1,2)$ and the $n$-cycle $(1,2,\ldots,n)$. This is true since, if we denote $\sigma = (1,2,\ldots,n)$, then we have

$$\sigma^{i-1} \circ (1,2) \circ \sigma^{-(i-1)} = (\sigma^{i-1}(1), \sigma^{i-1}(2)) = (i, i+1).$$

This completes the proof.

# 3 Application to PRESENT

The bit-permutation layer in PRESENT specifies that the $i$-th state bit is moved to the $P(i)$-th position after application of the permutation layer. Let us look at the unique decomposition of $P$ into its disjoint $k$-cycles. The disjoint decomposition of $P$ consists of a total of twenty 3-cycles, where the remaining four points are fixed. The 3-cycles are listed as follows:

- $(1,16,4)$, $(2,32,8)$, $(3,48,12)$, $(5,17,20)$, $(6,33,24)$,
- $(7,49,28)$, $(9,18,36)$, $(10,34,40)$, $(11,50,44)$, $(13,19,52)$,
- $(14,35,56)$, $(15,51,60)$, $(22,37,25)$, $(23,53,29)$, $(26,38,41)$,
- $(27,54,45)$, $(30,39,57)$, $(31,55,61)$, $(43,58,46)$, $(47,59,62)$.

Let the above 3-cycles be labeled by the symbols $c_0$ to $c_{19}$. Note that since all the $c_i$'s are disjoint, the composition of all of them in any order will result in $P$. Each $c_i$ may be further expressed as a composition of two swaps: $c_i = s_i \circ t_i$(note that $s_i$ and $t_i$ do not commute). Table 2 lists all such decompositions explicitly.

**Table 2:** Decomposition of the 3-cycle $c_i$'s into swaps for the PRESENT permutation

| $i$ | $c_i$ | $s_i \circ t_i$ | $i$ | $c_i$ | $s_i \circ t_i$ |
|---|---|---|---|---|---|
| 0 | $(1,16,4)$ | $(4,16) \circ (1,4)$ | 10 | $(14,35,56)$ | $(14,35) \circ (35,56)$ |
| 1 | $(2,32,8)$ | $(8,32) \circ (2,8)$ | 11 | $(15,51,60)$ | $(15,51) \circ (51,60)$ |
| 2 | $(3,48,12)$ | $(12,48) \circ (3,12)$ | 12 | $(22,37,25)$ | $(25,37) \circ (22,25)$ |
| 3 | $(5,17,20)$ | $(5,17) \circ (17,20)$ | 13 | $(23,53,29)$ | $(29,53) \circ (23,29)$ |
| 4 | $(6,33,24)$ | $(24,33) \circ (6,24)$ | 14 | $(26,38,41)$ | $(26,38) \circ (38,41)$ |
| 5 | $(7,49,28)$ | $(28,49) \circ (7,28)$ | 15 | $(27,54,45)$ | $(45,54) \circ (27,45)$ |
| 6 | $(9,18,36)$ | $(9,18) \circ (18,36)$ | 16 | $(30,39,57)$ | $(30,39) \circ (39,57)$ |
| 7 | $(10,34,40)$ | $(10,34) \circ (34,40)$ | 17 | $(31,55,61)$ | $(31,55) \circ (55,61)$ |
| 8 | $(11,50,44)$ | $(44,50) \circ (11,44)$ | 18 | $(43,58,46)$ | $(46,58) \circ (43,46)$ |
| 9 | $(13,19,52)$ | $(13,19) \circ (19,52)$ | 19 | $(47,59,62)$ | $(47,59) \circ (59,62)$ |

Note that if we were to compose a permutation consisting of application of all the $t_i$'s (in any order) followed by application of all the $s_i$'s (again in any order) we would get back $P$. That is to say

$$P = s_{b_0} \circ s_{b_1} \circ \cdots \circ s_{b_{19}} \circ t_{a_0} \circ t_{a_1} \circ \cdots \circ t_{a_{19}},$$

where $a_0, a_1, \ldots a_{19}$ and $b_0, b_1, \ldots b_{19}$ are any arbitrary orderings of the set $\{0, 1, \ldots, 19\}$. We will prove a generalized form of the above statement in the following lemma.

**Lemma 3.** *Let $\pi$ be a permutation in $S_n$ whose disjoint cycle decomposition consists of the cycles $c_0, c_1, \dots, c_{m-1}$ each with orders $i_0, i_1 \dots, i_{m-1}$ respectively (with $\sum_{j=0}^{m-1} i_j = n$), i.e.*

$$\pi = c_0 \circ c_1 \circ \cdots \circ c_{m-1}.$$

*Let $i_0 \leq i_1 \leq \cdots \leq i_{m-1}$. Let each $c_j$ be expressed as composition of $i_j - 1$ transpositions $s_j(1), s_j(2), \dots, s_j(i_j - 1)$. So we have*

$$\begin{array}{lccccccc}
 & s_{m-1}(i_{m-1}-1)\circ & \cdots \quad \circ & \cdots & \circ \cdots \circ & s_{m-1}(2) \circ & s_{m-1}(1) & = c_{m-1} \\
 & & & \vdots & & & & \\
 & & s_j(i_j-1)\circ & \cdots & \circ \cdots \circ & s_j(2) \circ & s_j(1) & = c_j \\
 & & & \vdots & & & & \\
 & & & s_0(i_0-1)\circ & \cdots \circ & s_0(2) \circ & s_0(1) & = c_0 \\
\hline\hline
\textit{Sets:} & \chi_{i_{m-1}-1} & \chi_{i_j-1} & \chi_{i_0-1} & \cdots & \chi_2 & \chi_1 &
\end{array}$$

*Define the set $\chi_k = \{s_{m-1}(k), s_{m-2}(k), \dots\}$ (for $1 \leq k < i_{m-1}$) as explained above. Let $\theta_k$ be the composition of all transpositions in $\chi_k$ in any arbitrary order. Then we must have*

**A** *Each $\theta_k$ is invariant of the order in which the transpositions in $\chi_k$ are applied.*

**B** *We must have $\pi = \theta_{i_{m-1}-1} \circ \cdots \theta_{i_j-1} \circ \cdots \theta_2 \circ \theta_1$.*

*Proof.* Please see Appendix A. □

The PRESENT permutation $P$ follows a specific instance of the above lemma, with $m = 20$ and $i_0 = i_1 = \cdots = i_{19} = 3$. Thus the fact that

$$P = s_{b_0} \circ s_{b_1} \circ \cdots \circ s_{b_{19}} \circ t_{a_0} \circ t_{a_1} \circ \cdots \circ t_{a_{19}}$$

is a corollary of the above lemma.

## 3.1 Implementation using 2 scan flip-flops

Lemma 2 already states that any permutation in $S_n$ can be generated by the cycles $(1, 2, \dots, n)$ and $(1, 2)$. In a typical serial implementation, the cycle $(1, 2, \dots, n)$ naturally appears as the rotation operation of the pipeline, constructed from $n$ flip flops. The swap $(1, 2)$ can be realized by simply replacing two of these flip-flops with scan flip-flops. Therefore, Lemma 2 implies the existence of PRESENT permutation realization with only 2 scan flip-flops. Therefore, we explore the number of cycles applying the PRESENT permutation takes, i.e. by deriving the decomposition sequence with a straightforward application of the above formalism.

In order to be compatible with the order of bit addressing used in block ciphers [BKL⁺07, BPP⁺17], we relabel the set of 64 elements by the indices $\{63, 62, \dots, 0\}$. After this relabeling, we can analogously claim that $S_{64}$ is generated by the cycles $w = (62, 63)$ and $r = (0, 1, 2, \dots, 63)$. The idea is to implement all the transpositions $t_i$ followed by all the $s_i$'s. In order to do so, let us first see how any arbitrary transposition can be implemented only using $r$ and $w$.

**Implementing a transposition $(x, y)$ for $(x > y)$ and $x, y \in [0, 63]$:** Let $\overline{x} = 63 - x$,

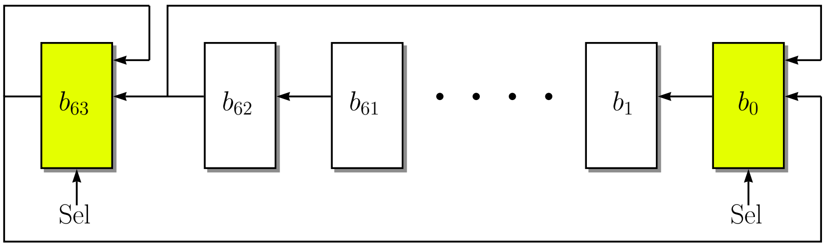


**Figure 1:** Shift register circuit with (subsequent) 2 scan flip-flops

$\overline{y} = 63 - y$. As per the proofs outlined in Lemmas 1 and 2, we have :

$$\begin{aligned}(x,y) &= (x,x-1)\circ(x-1,y)\circ(x,x-1)\\ &= (x,x-1)\circ(x-1,x-2)\circ(x-2,y)\circ(x-1,x-2)\circ(x,x-1)\\ &= (x,x-1)\circ(x-1,x-2)\circ\cdots\circ(y+1,y)\circ\cdots\circ(x-1,x-2)\circ(x,x-1)\\ &= (r^{-\overline{x}}\circ w\circ r^{\overline{x}})\circ(r^{-1-\overline{x}}\circ w\circ r^{1+\overline{x}})\circ\cdots\circ(r^{1-\overline{y}}\circ w\circ r^{\overline{y}-1})\circ\cdots\circ\\ &\quad (r^{-1-\overline{x}}\circ w\circ r^{1+\overline{x}})\circ(r^{-\overline{x}}\circ w\circ r^{\overline{x}})\\ &= r^{-\overline{x}}\circ w\circ(r^{-1}\circ w)^{x-y-1}\circ(r\circ w)^{x-y-1}\circ r^{\overline{x}}\\ &= r^{64-\overline{x}}\circ w\circ(r^{63}\circ w)^{x-y-1}\circ(r\circ w)^{x-y-1}\circ r^{\overline{x}}\\ &= r^{1+x}\circ w\circ(r^{63}\circ w)^{x-y-1}\circ(r\circ w)^{x-y-1}\circ r^{63-x}\end{aligned}$$

Given the decomposition $(x,y)$ in terms of $r$ and $w$ as given above, the next question naturally arises as to how to implement it using 2 scan flip-flops. Consider the circuit in Figure 1. It consists of an array of 64 flip-flops, with the 2 at the extreme ends being scan flip-flops controlled by a Sel signal. When Sel is 0, the data in the flip-flops simply rotate bitwise towards the left. When Sel is 1, the $b_{63}$ bit is held in place, and the data in the remaining 63 flip-flops is rotated left bitwise. Implementing a particular permutation $\pi \in S_{64}$ on this circuit, essentially tries to answer the following question: If we consider $b_i(t),\ i \in [0,63],\ t \geq 0$ to be the bit value stored on the $i^{th}$ flip-flop at time $t$, does there exist some sequence of Sel signals $s_0, s_1, \ldots, s_{T-1}$ such that for all $b_0(0), ..., b_{63}(0)$, setting Sel to $s_t$ at clock cycle $t$ implies that $b_{\pi(i)}(T) = b_i(0)$ for all $i$. The length $T$ of the sequence is the number of clock cycles needed to perform the permutation $\pi$.

**Lemma 4.** *Considering the circuit in Figure 1, implementing an arbitrary swap operation $(x,y)$ requires at most $64(x-y)$ clock cycles.*

*Proof.* To begin with, note that $r$ is a function that performs a rotation operation by one location towards the left. In Figure 1, setting the select signal Sel to 0, causes the shift register to implement the $r$ function, as data follows the circular path marked in the bottom. Setting Sel to 1, brings about the following transformation:

$$(b_{63}, b_{62}, b_{61}, \ldots, b_1, b_0) \rightarrow (b_{63}, b_{61}, b_{60}, \ldots, b_0, b_{62})$$

This is same as applying the function $(r \circ w)$. It is easy to see that $r$ and $(r \circ w)$ also generate $S_{64}$. Thus by controlling the Sel signal, we can make the shift register circuit alternate between $r$ and $v = (r \circ w)$ functions. Note that $(x,y)$ can be rewritten in blocks of 64 operations each, in the following manner:

$$\begin{aligned}(x,y) &= r^{1+x}\circ w\circ(r^{63}\circ w)^{x-y-1}\circ(r\circ w)^{x-y-1}\circ r^{63-x}\\ &= [r^{x}\circ v\circ r^{63-x}]\circ[r^{x-1}\circ v\circ r^{64-x}]\circ\cdots\circ[r^{y+2}\circ v\circ r^{61-y}]\circ[r^{y+1}\circ v^{x-y}\circ r^{63-x}]\end{aligned}$$

Each block of operations in square braces in the above equation is a set of 64 operations, and thus would take 64 clock cycles to execute using the shift register circuit. Since there are a total of $(x - y)$ braces, the result follows. □

**Corollary 1.** *Employing the shift register circuit in figure 1, one round of the* PRESENT *bit permutation can be executed in 36480 clock cycles.*

*Proof.* The idea is to execute the PRESENT permutation $P$ by executing each of the transpositions $t_i$ and then $s_i$ sequentially. Denoting $t_i = (x_i, y_i)$ and $s_i = (x_{20+i}, y_{20+i})$ for $i \in [0, 19]$, (with $x_i > y_i$) the number of clock cycles can be calculated as $\sum_{i=0}^{39} 64 \cdot (x_i - y_i) = 36480$. □

This result is a pessimistic one since it implies that to perform the PRESENT encryption operation on a shift register based circuit as given in Figure 1, would result in heavy loss of throughput. In the following subsections, we will try to see if the number of operations can be reduced in any way.

## 3.2 Decreasing the number of operations

Before we outline the method used to reduce the number of operations, let us look at the following definition.

**Definition 1.** As in Lemma 4, let $\pi$ be a permutation in $S_n$ whose disjoint cycle decomposition consists of the cycles $c_0, c_1, \ldots, c_{m-1}$ each with orders $i_0, i_1 \ldots, i_{m-1}$ respectively. Let each $c_j$ be expressed as composition of $i_j - 1$ transpositions $s_j(1), s_j(2), \ldots, s_j(i_j - 1)$. Denote the transposition $s_j(k) = (x_j(k), y_j(k))$ with $x_j(k) > y_j(k)$. $\pi$ is said to be a special permutation of the type $\kappa$, if $\kappa$ is the largest integer for which the following holds:

$$x_j(k) - y_j(k) \equiv 0 \bmod \kappa, \quad \forall\, j \in [0, m-1], \forall\, k \in [0, i_j - 1]$$

It is easy to see from Table 2, that the PRESENT permutation $P$ is a special permutation of type 3. Before we proceed, let us look at a result concerning special permutations of type $\kappa$.

**Lemma 5.** *Let $G_\kappa$ denote the set of all the special permutations of $S_{64}$ of type $\kappa$. Then $G_\kappa$ can be generated by the permutations $w_\kappa = (63 - \kappa, 63)$ and $r = (0, 1, \ldots, 63)$.*

*Proof.* The proof is similar to the ideas already discussed. For a detailed proof, please see Appendix B. □

The next step naturally is to see how any transposition $(x, y)$ with $x \equiv y \bmod \kappa$ can be implemented in a shift register structure using only 2 scan flip-flops using a method that requires lesser number of cycles as compared to the previous construction. We try to address this is the next lemma.

**Lemma 6.** *Consider the circuit in Figure 2. Implementing an arbitrary swap operation $(x, y)$ with $x > y$ and $x \equiv y \bmod \kappa$ using it can be implemented in $\frac{64(x-y)}{\kappa} = 64z$ clock cycles.*

*Proof.* As before, setting Sel to 0, executes the rotate function $r$. Setting Sel to 1, achieves the following transformation:

$$(b_{63}, b_{62}, b_{61}, \ldots, b_1, b_0) \rightarrow (b_{62}, b_{61}, \ldots, b_{64-\kappa}, b_{63},\ b_{62-\kappa}, b_{61-\kappa}, \ldots, b_0, b_{63-\kappa})$$

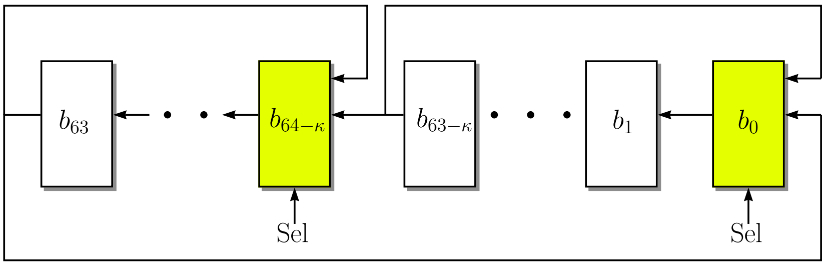


**Figure 2:** Shift register circuit with ($\kappa$-respecting) 2 scan flip-flops

This is same as applying the transformation $v_\kappa = r \circ w_\kappa$. Thus, as before, controlling $\mathsf{Sel}$ makes the circuit alternate between $r$ and $v_\kappa$ operations. As before we express $(x, y)$ in blocks of 64 operations:

$$\begin{aligned}(x,y) &= r^{1+x} \circ w_\kappa \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ (r^\kappa \circ w_\kappa)^{z-1} \circ r^{63-x} \\ &= [r^x \circ v_\kappa \circ r^{63-x}] \circ [r^{x-\kappa} \circ v_\kappa \circ r^{63-x+\kappa}] \circ \cdots \circ [r^{x-(z-2)\kappa} \circ v_\kappa \circ r^{63-x+(z-2)\kappa}] \circ \\ &\quad [r^{y+1} \circ (r^{\kappa-1} \circ v_\kappa)^z \circ r^{63-x}]\end{aligned}$$

Operations in each of the square braces take 64 cycles and since there are exactly $z$ such braces, the result follows. □

**Corollary 2.** *Using the shift register circuit in Figure 2, one round of the* PRESENT *bit permutation $P$ can be executed in 12160 clock cycles.*

*Proof.* We have already noted that $P$ is a special permutation of type 3. As in the previous corollary, let $t_i = (x_i, y_i)$ and $s_i = (x_{20+i}, y_{20+i})$ for $i \in [0, 19]$, (with $x_i > y_i$). For performing all the $t_i$'s followed by all the $s_i$'s sequentially, the number of clock cycles can be calculated as $\sum_{i=0}^{39} 64 \cdot \frac{(x_i - y_i)}{3} = 12160$. □

By using the modified shift register structure, we obtain a threefold increase of throughput in computation of the PRESENT permutation. However, this is still way too slow, and in the subsequent sections we will try to find if the computations can be further sped up.

## 3.3 Further reduction

Until so far, we were executing each transposition operation sequentially, i.e. one after the other. However in the interest of speeding up computations, let us investigate if it is at all possible to execute some of the swap operations concurrently.

**Definition 2.** Let $\sigma = (x, y)$ be a transposition in $S_{64}$ with $x > y$. $\overrightarrow{\mathsf{Sel}}_\sigma$ to be the vector of $\mathsf{Sel}$ signals that achieves the computation of $\sigma$ using the circuit in Figure 2. The length of $\overrightarrow{\mathsf{Sel}}_\sigma$ is therefore $\frac{64(x-y)}{\kappa}$. For example, let $\kappa = 3$, as in PRESENT. Consider $\sigma = (60, 51)$, for which $z = 3$. We have

$$\begin{aligned}\sigma &= [r^x \circ v_\kappa \circ r^{63-x}] \circ [r^{x-\kappa} \circ v_\kappa \circ r^{63-x+\kappa}] \circ \cdots \circ [r^{x-(z-2)\kappa} \circ v_\kappa \circ r^{63-x+(z-2)\kappa}] \circ \\ &\quad [r^{y+1} \circ (r^{\kappa-1} \circ v_\kappa)^z \circ r^{63-x}] \\ &= [r^{60} \circ v_3 \circ r^3] \circ [r^{57} \circ v_3 \circ r^6] \circ [r^{52} \circ (r^2 \circ v_3)^3 \circ r^3] \\ \overrightarrow{\mathsf{Sel}}_\sigma &= \underset{\longleftarrow \text{ Increasing Index}}{\underleftarrow{0^{60}\ 1\ 0^3 \quad 0^{57}\ 1\ 0^6 \quad 0^{52}\ 0^2 1\ 0^2 1\ 0^2 1\ 0^3}}\end{aligned}$$

Note that to keep notations consistent as to the order of application of the permutations, the rightmost element in the vector is denoted as the $0^{th}$ element, and the index is increased as we go left. This is consistent with the order of application in the composition notation, in which the rightmost permutation of the composition is applied first. Let us now re-write the permutations $r$ and $v_\kappa$ in functional form:

$$r(\alpha) = (\alpha+1) \bmod 64, \quad v_\kappa(\alpha) = \begin{cases} 64-\kappa, & \text{if } \alpha = 63, \\ 0, & \text{if } \alpha = 63-\kappa, \\ (\alpha+1) \bmod 64, & \text{otherwise.} \end{cases}$$

We can see that $r$ and $v_\kappa$ differ on only two inputs 63 and $63-\kappa$. By stretching notations slightly, let $\overrightarrow{\mathsf{Sel}}_p$ also denote a random 64-bit binary vector that implements the permutation $p$ when fed to the $\mathsf{Sel}$ port of the circuit in Figure 2 over 64 consecutive clock cycles. Let $\mathbb{B}_p$ be the set of elements that denote the positions of 1's in $\overrightarrow{\mathsf{Sel}}_p$. From the functional equations of $r$ and $v_\kappa$, it is not difficult to deduce that (a simple code in any programming language is sufficient to do this) $\mathbb{A}_p = \mathbb{U}_p \cup \mathbb{V}_p$, where

$$\mathbb{U}_p = \{63-\alpha : \alpha \in \mathbb{B}_p\}, \ \mathbb{V}_p = \{63-\alpha-\kappa \bmod 64 : \alpha \in \mathbb{B}_p\}$$

It is also possible to deduce $p$ from $\mathbb{B}_p$. If $\mathbb{B}_p$ contains elements $b, b+\kappa, b+2\kappa, \ldots, b+(l-1)\kappa$ which are in an arithmetic sequence with common difference $\kappa$ then we will have

$$p(63-b-i\kappa) = 63-b-(i-1)\kappa, \ \forall i \in [1,l], \ \text{and} \ \ p(63-b) = 63-b-l\kappa$$

For all other elements $\hat{b}$ in $\mathbb{B}_p$ that are not part of an arithmetic sequence with common difference $\kappa$, we have $p(63-\hat{b}) = 63-\hat{b}-\kappa$ and $p(63-\hat{b}-\kappa) = 63-\hat{b}$. For all other elements we have $p(b) = b$.

**Example 1.** For example if $\mathbb{B}_p = \{6, 9, 19, 29, 53, 56, 60, 61\}$ with $\kappa = 3$, we see that we have 2 arithmetic sequences of common difference 3: $6, 9$ and $53, 56$. So we have $\mathbb{A}_p = \{0, 2, 3, 4, 7, 10, 31, 34, 41, 44, 51, 54, 57, 63\}$ We have $p = (51, 54, 57) \circ (4, 7, 10) \circ (44, 41) \circ (34, 31) \circ (3, 0) \circ (2, 63)$.

Consider every 64 bit block of the $\overrightarrow{\mathsf{Sel}}_\sigma$ vector. Let $\pi_i$ (for $i = 0$ to $z-1$) be the composition of all the permutations in the $i^{th}$ 64-bit block. Let us use the notation

$$\overrightarrow{\mathsf{Sel}}_\sigma = \overrightarrow{\mathsf{Sel}}_{\pi_{z-1}} || \overrightarrow{\mathsf{Sel}}_{\pi_{z-2}} || \cdots || \overrightarrow{\mathsf{Sel}}_{\pi_2} || \overrightarrow{\mathsf{Sel}}_{\pi_1} || \overrightarrow{\mathsf{Sel}}_{\pi_0}.$$

Of course we have $\sigma = \pi_{z-1} \circ \pi_{z-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$. In the above example, for $\sigma = (60, 51)$ we have $\mathbb{B}_{\pi_0} = \{3, 6, 9\}$, $\mathbb{B}_{\pi_1} = \{6\}$, $\mathbb{B}_{\pi_2} = \{3\}$. Generalizing the above we can see that $\mathbb{B}_{\pi_0} = \{63-x, 63-x-\kappa, \ldots, 63-y-\kappa\}$. $\mathbb{B}_{\pi_0}$ only contains elements that are $\overline{x} = 63 - x \bmod \kappa$. And we have that $\mathbb{B}_{\pi_i} \subset \mathbb{B}_{\pi_0}$, $\forall \ i > 0$. From the analysis presented above, it can be deduced that for all $i$,

$$\pi_i(\alpha) = \alpha, \ \forall \alpha \ \not\equiv x \bmod \kappa.$$

This is because the 1's (equivalently $v_\kappa$'s) in this block appear at distances of $\kappa$. If we apply each function in $\pi_i$ one by one, for any input $\alpha \not\equiv x \bmod \kappa$, the corresponding input to $v_\kappa$ is never 63 or $63-\kappa$, and so a plain rotation is effectively executed. Therefore all the $\pi_i$'s perform shuffling on only a subset of elements that are congruent to $x \bmod \kappa$ and leave the others untouched. From the equation $\mathbb{A}_p = \mathbb{U}_p \cup \mathbb{V}_p$, we can also deduce that $\mathbb{A}_{\pi_0} = \{x, x-\kappa, x-2\kappa, \ldots, y\}$. Thus each $\pi_i$ is effectively a permutation function on only a subset of $\{0, 1, 2, 3, \ldots, 63\}$ that are congruent to $x \bmod \kappa$.

**Lemma 7.** *Let $\overrightarrow{\mathsf{Sel}}_{p_1}$ and $\overrightarrow{\mathsf{Sel}}_{p_2}$ be two 64 bit signal vectors implementing permutations $p_1$ and $p_2$ on the circuit of Figure 2. If $\mathbb{A}_{p_1} \cap \mathbb{A}_{p_2} = \varnothing$, then $p_1 \circ p_2$ can be concurrently executed on this circuit using the signal vector $\overrightarrow{\mathsf{Sel}}_{p_1} \upharpoonleft \overrightarrow{\mathsf{Sel}}_{p_2}$, where $\upharpoonleft$ denotes a bitwise OR operation on the vectors.*

*Proof.* See Appendix C. □

**Lemma 8.** *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two special transpositions ($x_i > y_i$, $i = 1, 2$) in $S_{64}$ of type $\kappa$. Without loss of generality let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. Let the respective decompositions are denoted by the symbols $\pi_i$ and $\theta_i$, i.e. $\sigma_1 = \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$ and $\sigma_2 = \theta_{z_2-1} \circ \theta_{z_2-2} \circ \cdots \circ \theta_2 \circ \theta_1 \circ \theta_0$. $\overrightarrow{\mathsf{Sel}}_{\sigma_1}$ and $\overrightarrow{\mathsf{Sel}}_{\sigma_2}$ may not be of the same length, in which case append $64(z_1 - z_2)$ zeroes to $\overrightarrow{\mathsf{Sel}}_{\sigma_2}$[1] to make them of the same length. If $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} = \varnothing$, then it is possible to execute $\sigma_1$ and $\sigma_2$ concurrently on the circuit in Figure 2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles. Let $\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2}$ be the vector of* Sel *signals required to achieve this. Then $\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2} = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|}\ \overrightarrow{\mathsf{Sel}}_{\sigma_2}$.*

*Proof.* See Appendix D. □

The above result may be extended to a set of any number of special transpositions $\sigma_i$ ($i = 1$ to $k$) of the type $\kappa$, provided that the respective $\mathbb{A}_{\pi_0}$ sets are pairwise disjoint. In that case we have

$$\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_k} = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|}\ \overrightarrow{\mathsf{Sel}}_{\sigma_2} \hat{|}\ \cdots \hat{|}\ \overrightarrow{\mathsf{Sel}}_{\sigma_k}$$

**Corollary 3.** *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two transpositions ($x_i > y_i$, $i = 1, 2$) in $S_{64}$ such that $x_1 - y_1 \equiv x_2 - y_2 \bmod \kappa$, and $x_1 \not\equiv x_2 \bmod \kappa$. Without loss of generality let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. As before, let the respective decompositions are denoted by the symbols $\pi_i$ and $\theta_i$ and append $64(z_1 - z_2)$ zeroes to $\overrightarrow{\mathsf{Sel}}_{\sigma_2}$ to make the two $\overrightarrow{\mathsf{Sel}}$ vectors of the same length. It is possible to execute $\sigma_1$ and $\sigma_2$ concurrently on the circuit in Figure 2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles by using $\overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|}\ \overrightarrow{\mathsf{Sel}}_{\sigma_2}$ as the select signal vector.*

*Proof.* We have already seen that for any transposition $\sigma = (x, y) = \pi_{z-1} \circ \cdots \circ \pi_0$, we have $\mathbb{A}_{\pi_0} = \{x, x - \kappa, x - 2\kappa, \ldots, y\}$. Thus $\mathbb{A}_{\pi_0}$ contains elements that are only congruent to $x \bmod \kappa$. Since $x_1, y_1$ and $x_2, y_2$ belong to different equivalence classes modulo $\kappa$, $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} = \varnothing$. Thus the result follows. □

**Corollary 4.** *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two transpositions ($x_i > y_i$, $i = 1, 2$) in $S_{64}$ such that $y_1 > x_2$. Let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. Let the respective decompositions are denoted by the symbols $\pi_i$ and $\theta_i$. Then after making the $\overrightarrow{\mathsf{Sel}}$ vectors of the same length by appending zeroes, it is possible to execute $\sigma_1$ and $\sigma_2$ concurrently on the circuit in Figure 2 and achieve $\sigma_1 \circ \sigma_2$ in $64 \cdot z_1$ clock cycles by using $\overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|}\ \overrightarrow{\mathsf{Sel}}_{\sigma_2}$ as the select signal vector.*

*Proof.* We have $\mathbb{A}_{\pi_0} = \{x_1, x_1 - \kappa, x_1 - 2\kappa, \ldots, y_1\}$ and $\mathbb{A}_{\theta_0} = \{x_2, x_2 - \kappa, x_2 - 2\kappa, \ldots, y_2\}$. Since $y_1 > x_2$, clearly $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} = \varnothing$. Thus the result follows. □

We can use the results in the above two corollaries to further reduce the execution time of the PRESENT permutation. We have to execute all the transpositions $t_i$ followed by the transpositions $s_i$. The idea is to execute as many permutations concurrently which have pairwise disjoint $\mathbb{A}_{\pi_0}$'s. We can easily partition the transpositions modulo $\kappa = 3$. Transpositions that are in different classes modulo 3 can obviously be executed concurrently. Also transpositions in the same class modulo 3, which have disjoint $\mathbb{A}_{\pi_0}$'s can also be executed together. For the $t_i$'s we can think of the following solution given in Table 3, that takes $(11 + 7 + 1) \cdot 64 = 704 + 448 + 64 = 1216$ cycles. All the swaps in $i^{th}$ group can be executed concurrently, thereby reducing the number of cycles.

A similar construction for the $s_i$'s will take $(12 + 12 + 7 + 4) \cdot 64 = 2240$ cycles. So a total of $1216 + 2240 = 3456$ cycles are required which is already way better than our previous construction of 12160 cycles.

[1] Since $r^{64}$ is the identity function, this does not affect either permutation

**Table 3:** Concurrent execution of the $t_i$'s in the PRESENT permutation

| Group | mod3 | $t_i$ | $\max(x_i - y_i)$ | #Cycles |
|---|---|---|---|---|
| 1 | 0 | $(57,39),\ (36,18),\ (12,3)$ | 33 | 704 |
| | 1 | $(61,55),\ (52,19),\ (4,1)$ | | |
| | 2 | $(62,59),\ (44,11),\ (8,2)$ | | |
| 2 | 0 | $(60,51),\ (45,27),\ (24,6)$ | 21 | 448 |
| | 1 | $(46,43),\ (40,34),\ (28,7)$ | | |
| | 2 | $(56,35),\ (29,23),\ (20,17)$ | | |
| 3 | 1 | $(25,22)$ | 3 | 64 |
| | 2 | $(41,38)$ | | |

**Table 4:** Concurrent execution of the $s_i$'s in the PRESENT permutation

| Group | mod3 | $s_i$ | $\max(x_i - y_i)$ | #Cycles |
|---|---|---|---|---|
| 1 | 0 | $(51,15)$ | 36 | 768 |
| | 1 | $(55,31),\ (19,13)$ | | |
| | 2 | $(53,29),\ (17,5)$ | | |
| 2 | 0 | $(48,12)$ | 36 | 768 |
| | 1 | $(58,46),\ (34,10)$ | | |
| | 2 | $(59,47),\ (32,8)$ | | |
| 3 | 0 | $(54,45),\ (39,30),\ (18,9)$ | 21 | 448 |
| | 1 | $(49,28),\ (16,4)$ | | |
| | 2 | $(50,44),\ (35,14)$ | | |
| 4 | 0 | $(33,24)$ | 12 | 256 |
| | 1 | $(37,25)$ | | |
| | 2 | $(38,26)$ | | |

## 3.4 Final Optimization

In this final subsection we see if the number of clock cycles can be further optimized. Specifically we want to see if it is possible to implement transpositions $\sigma_1$ and $\sigma_2$ concurrently, even if the corresponding $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} \neq \varnothing$. We start with a well known result in permutation theory.

**Theorem 1.** *For every permutation $\sigma \in S_{64}$, and every transposition $(x, y) \in S_{64}$:*

$$f = \sigma \circ (x, y) = (\sigma(x), \sigma(y)) \circ \sigma$$

The above is not difficult to prove, $\forall \alpha \notin \{x, y\}$, we have $f(\alpha) = \sigma(\alpha)$. And both sides evaluate to $f(x) = \sigma(y)$ and $f(y) = \sigma(x)$.

**Lemma 9.** *Let $\sigma_1 = (x_1, y_1)$ and $\sigma_2 = (x_2, y_2)$ be two special disjoint transpositions ($x_i > y_i$, $i = 1, 2$) in $S_{64}$ of type $\kappa$. Without loss of generality let $\ell_1 = (x_1 - y_1) \geq (x_2 - y_2) = \ell_2$, and $z_i = \frac{\ell_i}{\kappa}$. Let the respective decompositions be denoted by the symbols $\pi_i$ and $\theta_i$, i.e. $\sigma_1 = \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$ and $\sigma_2 = \theta_{z_2-1} \circ \theta_{z_2-2} \circ \cdots \circ \theta_2 \circ \theta_1 \circ \theta_0$. Let us have $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} \neq \varnothing$. Denote by $p = (\pi[i \to 0](x_2), \pi[i \to 0](y_2))$, for some $i \in [0, z_1 - 1]$. Let the decomposition of $p$ be denoted as*

$$p = \gamma_{q-1} \circ \gamma_{q-2} \circ \cdots \circ \gamma_1 \circ \gamma_0.$$

*Now denote* $\overrightarrow{\mathsf{Sel}}_1 = \overrightarrow{\mathsf{Sel}}_{\pi_{z_1-1}} || \overrightarrow{\mathsf{Sel}}_{\pi_{z_1-2}} || \cdots || \overrightarrow{\mathsf{Sel}}_{\pi_{i+1}}$ *and* $\overrightarrow{\mathsf{Sel}}_2 = \overrightarrow{\mathsf{Sel}}_p$*. After appending with zeroes to make* $\overrightarrow{\mathsf{Sel}}_1$ *and* $\overrightarrow{\mathsf{Sel}}_2$ *of the same length, the following vector*

$$\overrightarrow{\mathsf{Sel}}_1 \upharpoonleft\!\!\!\upharpoonright \overrightarrow{\mathsf{Sel}}_2 \; || \; \overrightarrow{\mathsf{Sel}}_{\pi[i \to 0]}$$

*will execute* $\sigma_1 \circ \sigma_2$ *on the circuit in Figure 2, if* $\mathbb{B}_{\gamma_0} \cap \mathbb{B}_{\pi_{i+1}} = \varnothing$*.*

*Proof.* See Appendix E. □

**Example 2.** An immediate application of the above is to construct a $\overrightarrow{\mathsf{Sel}}$ vector to execute $(51, 15)$ and $(48, 12)$ concurrently on the PRESENT circuit. In the previous subsection we had executed them sequentially which had cost us $12 \cdot 64 = 768$ cycles each. Start with $\sigma_1 = (48, 12)$. We have $\mathbb{B}_{\pi_0} = \{15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48\}$, $\mathbb{B}_{\pi_j} = \{48 - 3j\}$ for $1 \le j \le 11$. For $\sigma_2 = (51, 15)$, we observe that $(\pi_0(51), \pi_0(15)) = (51, 18)$. If we let $p = (51, 18)$, then $\mathbb{B}_{\gamma_0} = \{12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42\}$. Since $\mathbb{B}_{\gamma_0} \cap \mathbb{B}_{\pi_1} = \varnothing$, this choice of $p$ will work. Also $\overrightarrow{\mathsf{Sel}}_1$ and $\overrightarrow{\mathsf{Sel}}_2$ will be of the same length, due to which zero padding is also not required. So we have $\mathbb{B}_{\gamma_j} = \{42 - 3j\}$ for all $1 \le j \le 10$. Since $\overrightarrow{\mathsf{Sel}}_{\sigma_1 \circ \sigma_2} = \overrightarrow{\mathsf{Sel}}_1 \upharpoonleft\!\!\!\upharpoonright \overrightarrow{\mathsf{Sel}}_2 \; || \; \overrightarrow{\mathsf{Sel}}_{\pi[i \to 0]}$, denoting the decomposition of $\sigma_1 \circ \sigma_2$ by the symbols $\eta_j$, we have $\mathbb{B}_{\eta_0} = \mathbb{B}_{\pi_0}$, $\mathbb{B}_{\eta_{j+1}} = \mathbb{B}_{\pi_{j+1}} \cup \mathbb{B}_{\gamma_j}$ for $0 \le j \le 10$. This therefore implements $\sigma_1 \circ \sigma_2$ in only $12 \cdot 64 = 768$ cycles.

In the next lemma, we take things forward. If $\sigma$ is a permutation which implements a set of disjoint transpositions (instead of just a single transposition), it may be possible to implement another transposition $\sigma'$ concurrently along with $\sigma$ if certain conditions are met.

**Lemma 10.** *Let* $\sigma$ *be a special permutation of type* $\kappa$ *that is a composition of several pairwise disjoint transpositions. Let* $\sigma' = (x, y)$ *be a special transposition* $(x > y)$ *of type* $\kappa$*, that is also pairwise disjoint with each of the transpositions that compose* $\sigma$*. Let the respective decompositions are denoted by the symbols* $\pi_i$ *and* $\theta_i$*, i.e.* $\sigma = \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0$ *and* $\sigma' = \theta_{z_2-1} \circ \theta_{z_2-2} \circ \cdots \circ \theta_2 \circ \theta_1 \circ \theta_0$*. Let us have* $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} \ne \varnothing$*. Denote by* $p = (\pi[i \to 0](x), \pi[i \to 0](y))$*, for some* $i \in [0, z_1 - 1]$*. Let the decomposition of* $p$ *be denoted as*

$$p = \gamma_{q-1} \circ \gamma_{q-2} \circ \cdots \circ \gamma_1 \circ \gamma_0.$$

*Now denote* $\overrightarrow{\mathsf{Sel}}_1 = \overrightarrow{\mathsf{Sel}}_{\pi_{z_1-1}} || \overrightarrow{\mathsf{Sel}}_{\pi_{z_1-2}} || \cdots || \overrightarrow{\mathsf{Sel}}_{\pi_{i+1}}$ *and* $\overrightarrow{\mathsf{Sel}}_2 = \overrightarrow{\mathsf{Sel}}_p$*. After appending with zeroes to make* $\overrightarrow{\mathsf{Sel}}_1$ *and* $\overrightarrow{\mathsf{Sel}}_2$ *of the same length, the following vector*

$$\overrightarrow{\mathsf{Sel}}_1 \upharpoonleft\!\!\!\upharpoonright \overrightarrow{\mathsf{Sel}}_2 \; || \; \overrightarrow{\mathsf{Sel}}_{\pi[i \to 0]}$$

*will execute* $\sigma \circ \sigma'$ *on the circuit in Figure 2, if* $\mathbb{B}_{\gamma_0} \cap \mathbb{B}_{\pi_{i+1}} = \varnothing$*.*

*Proof.* We give a sketch of the proof as a complete analytical proof is likely to be quite complicated. The idea is similar to the ideas explained in the proof of Lemma 9. Note that $\mathbb{B}_{\pi_{i+j+1}}$ $(j \ge 0)$ will be the union of the corresponding $\mathbb{B}$ sets of the several transpositions that compose $\sigma$. One has to iterate the "disjoincy" arguments introduced in Lemma 9, for $\mathbb{B}_{\gamma_0}$ and each of those $\mathbb{B}$ sets to arrive at a proof. □

**Example 3.** Let us construct a $\overrightarrow{\mathsf{Sel}}$ vector for all the $s_i$'s in PRESENT that are congruent to 0 mod 3. The transpositions are $(51, 15)$, $(48, 12)$, $(54, 45)$, $(39, 30)$, $(18, 9)$, $(33, 24)$. We already have a $\overrightarrow{\mathsf{Sel}}$ vector for $(51, 15) \circ (48, 12)$ in the previous example.

1. To start, we have $\sigma = (51, 15) \circ (48, 12)$, $\mathbb{B}_{\pi_0} = \{15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48\}$, $\mathbb{B}_{\pi_1} = \{12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45\}$, and $\mathbb{B}_{\pi_j} = \{48 - 3j, 45 - 3j\}$ for all $2 \le j \le 11$. Let $\sigma' = (54, 45)$. Now $(\pi[1 \to 0](54), \pi[1 \to 0](45)) = (54, 51)$. If $p = (54, 51)$ then $\mathbb{B}_{\gamma_0} = \{9\}$. which is disjoint with $\mathbb{B}_{\pi_2} = \{42, 39\}$. Since $\mathbb{B}_{\gamma_0}$ has only one element it is sufficient to generate $p$. So we have $\mathbb{B}_{\eta_2} = \mathbb{B}_{\pi_2} \cup \mathbb{B}_{\gamma_0} = \{42, 39, 9\}$. For all other $j$, we have $\mathbb{B}_{\eta_j} = \mathbb{B}_{\pi_j}$. This will give us $\sigma \circ \sigma'$.

2. This time $\sigma = (51, 15) \circ (48, 12) \circ (54, 45)$. Shifting notations, we have $\mathbb{B}_{\pi_0} = \{15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48\}$, $\mathbb{B}_{\pi_1} = \{12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45\}$, $\mathbb{B}_{\pi_2} = \{42, 39, 9\}$, and $\mathbb{B}_{\pi_j} = \{48 - 3j, 45 - 3j\}$ for all $3 \leq j \leq 11$. Let $\sigma' = (33, 24)$. Now $(\pi[1 \to 0](33), \pi[1 \to 0](24)) = (39, 30)$. If $p = (39, 30)$ then $\mathbb{B}_{\gamma_0} = \{24, 27, 30\}$. which is disjoint with $\mathbb{B}_{\pi_2} = \{42, 39, 9\}$. We have $\mathbb{B}_{\gamma_1} = \{27\}$ and $\mathbb{B}_{\gamma_2} = \{24\}$. So we have $\mathbb{B}_{\eta_2} = \mathbb{B}_{\pi_2} \cup \mathbb{B}_{\gamma_0} = \{42, 39, 30, 27, 24, 9\}$, $\mathbb{B}_{\eta_3} = \mathbb{B}_{\pi_3} \cup \mathbb{B}_{\gamma_1} = \{39, 36, 27\}$, $\mathbb{B}_{\eta_4} = \mathbb{B}_{\pi_4} \cup \mathbb{B}_{\gamma_2} = \{36, 33, 24\}$. For all other $j$, we have $\mathbb{B}_{\eta_j} = \mathbb{B}_{\pi_j}$.

3. Now $\sigma = (51, 15) \circ (48, 12) \circ (54, 45) \circ (33, 24)$. Continuing in this manner, we can add $(39, 30)$ and $(18, 9)$ to this chain. This completes the construction for all the $s_i$'s of the form 0 mod 3. Let us enumerate the sets explicitly

$$\begin{aligned}
\mathbb{B}_{\eta_0} &= \{15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48\} \\
\mathbb{B}_{\eta_1} &= \{12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45\} \\
\mathbb{B}_{\eta_2} &= \{42, 39, 30, 27, 24, 9\}, \ \mathbb{B}_{\eta_3} = \ \{39, 36, 27, 21, 18\} \\
\mathbb{B}_{\eta_4} &= \{51, 48, 45, 36, 33, 24, 18\}, \ \mathbb{B}_{\eta_5} = \ \{48, 33, 30\} \\
\mathbb{B}_{\eta_6} &= \{45, 30, 27\}, \mathbb{B}_{\eta_j} = \ \{48 - 3j, 45 - 3j\}, \ \forall\ 7 \leq j \leq 11
\end{aligned}$$

This therefore constructs all the $s_i$'s of the PRESENT permutation that are congruent to 0 mod 3 in $12 \cdot 64 = 768$ cycles. We can use a similar approach to construct $\overrightarrow{\mathsf{Sel}}$ vectors for the $s_i$'s that are congruent to $1, 2$ mod 3, that can operate in $8 \cdot 64 = 512$ cycles each. After padding of each of them to 768 cycles, a simple application of lemma 8 and corollary 3, allows us to construct the $\overrightarrow{\mathsf{Sel}}$ vector for the composition of all the $s_i$'s by computing $\overrightarrow{\mathsf{Sel}}_{0 \bmod 3} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{1 \bmod 3} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{2 \bmod 3}$ that operates in just 768 cycles. Table 12 in the appendix lists the $\mathbb{B}$ vectors for all the $s_i$ transpositions that are $1, 2$ mod 3. Using a similar approach one can construct a similar $\overrightarrow{\mathsf{Sel}}$ vector for the composition of all $t_i$'s that operate in $11 \cdot 64 = 704$ cycles. The $\mathbb{B}$ vectors for the $s_i$ transpositions are also listed in Table 12. Since our strategy is to execute the composition of the $t_i$'s followed by the $s_i$'s this approach takes $704 + 768 = \mathbf{1472}$ cycles, which is the best we could manage.

# 4 The PRESENT circuit using 2 scan flip-flops

Using the mathematical background presented in the previous section, we present our construction of the PRESENT circuit. The circuit for the datapath and the keypath are described in Figures 3 and 4 respectively. Note that although it appears that the circuit employs two s-boxes, one for the data and keypaths each, in fact there is only one s-box circuit with a multiplexer in front which accepts inputs from the state and key registers at different periods in the encryption cycle. We defer the cycle by cycle operational aspects of the circuit to Appendix G.

## 4.1 Area and throughput results

The number of cycles taken to compute the encryption is therefore $80 + 1536 \cdot 31 + 64 = 47760$. This is around 24 times slower than the implementation in [JMPS17]. Assuming that the $\overrightarrow{\mathsf{Sel}}$ vector is stored as a look-up-table within the circuit, then after synthesizing the circuit using the STM 90nm standard cell library, the synthesized circuit occupies logic area of around 943 GE which is also 100 GE more than the implementation in [JMPS17]. This is much more than we expected because of the following reason:

- A single round requires the Sel signal to be micro-controlled for 1472 cycles. This can only be done if the corresponding bit values are available as a lookup table inside the circuit.

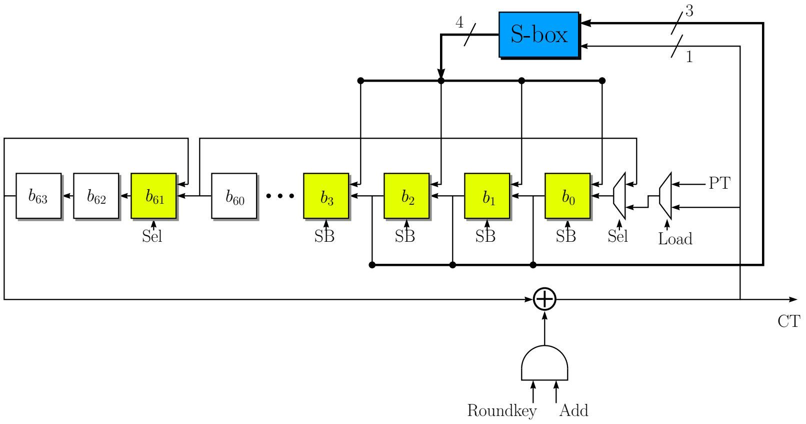


**Figure 3:** The PRESENT datapath

- This requires around 200 B of storage and as seen in Figure 13, it requires around 170 GE of circuit area.
- Thus, the 2 scan flip-flop circuit proves counterproductive and is of only theoretical interest.

## 4.2 Circuit for combined encryption and decryption

The approach outlined in the previous section is surprisingly effective when we try to implement a PRESENT circuit that can offer the combined functionalities of encryption and decryption. As already pointed out in [BBR16, BBR17a, JMPS17], such circuits are useful in implementing modes of operation like ELmD, CBC that require access to both the block cipher and its inverse operation. Our strategy to execute the PRESENT permutation $P$ was to first execute the transpositions $t_i$ in any order and then the transpositions $s_i$ again in any order. In order to execute the inverse permutation $P^{-1}$ it is enough to reverse the order: first execute the $s_i$'s followed by the $t_i$'s as explained below.

In order to understand why this is so, let us denote the composition of all $s_i$'s as $\mathcal{S}$ and the composition of all $t_i$'s as $\mathcal{T}$, so that we have $P = \mathcal{S} \circ \mathcal{T}$. Both $\mathcal{S}$ and $\mathcal{T}$ are compositions of disjoint transpositions. Transpositions are involutary functions, which is to say they are self-inverses. Since all the $t_i$'s in $\mathcal{T}$ are disjoint, $\mathcal{T}^{-1}$ is again the composition of all the

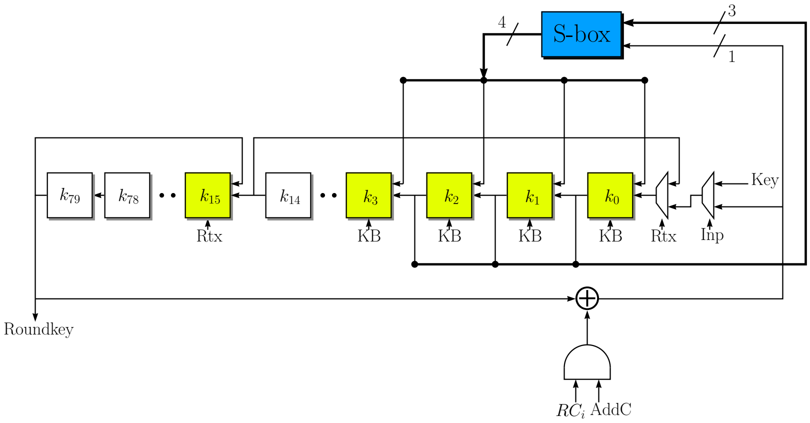


**Figure 4:** The PRESENT Keypath

$t_i$'s therefore equal to $\mathcal{T}$. The same is true for $\mathcal{S}$. So we have $P^{-1} = \mathcal{T}^{-1} \circ \mathcal{S}^{-1} = \mathcal{T} \circ \mathcal{S}$. Thus the inverse permutation can be executed on the same circuit by shuffling around the $\overrightarrow{\mathsf{Sel}}$ vector. The operational levels of the circuit are deferred to Appendix G.

The area occupied by the circuit when synthesized with the standard cell library of the STM 90nm CMOS process, is around 1040 GE. This is around 200 GE less than the previous best reported implementation of the combined circuit for PRESENT in [BBR17b], which occupies around 1240 GE. However the area is still high due to the inclusion of the control bits as a lookup table.

# 5 Application to GIFT

GIFT was a block cipher designed by Banik et al. [BPP+17] and presented at CHES 2017, with a view to strengthen the cryptographic properties of PRESENT by redesigning the permutation layer and keyschedule. It is a block cipher with an SPN round function in which the linear layer is a bit permutation similar to PRESENT.

The following can be said about the permutation function $G$ used in GIFT:

1. It is a special permutation of type $\kappa = 4$.

2. It can be decomposed into fourteen 4-cycles and two 2-cycles all of which are pairwise disjoint. Additionally it has 4 fixed points.

3. Each 4-cycle can be decomposed into three transpositions $s_i \circ t_i \circ u_i$. The decomposition is shown in the following table.

**Table 5:** Decomposition of the $c_i$'s in the GIFT permutation

| $i$ | $c_i$ | $s_i \circ t_i \circ u_i$ | $i$ | $c_i$ | $s_i \circ t_i \circ u_i$ |
|---|---|---|---|---|---|
| 0 | $(1,17,21,5)$ | $(5,17) \circ (17,21) \circ (1,5)$ | 8 | $(11,19,55,47)$ | $(19,47) \circ (19,55) \circ (11,47)$ |
| 1 | $(2,34,32,10)$ | $(10,34) \circ (34,42) \circ (2,10)$ | 9 | $(13,33,25,53)$ | $(13,25) \circ (25,53) \circ (13,33)$ |
| 2 | $(3,51,63,15)$ | $(15,51) \circ (51,63) \circ (3,15)$ | 10 | $(14,50,46,58)$ | $(14,46) \circ (46,58) \circ (14,50)$ |
| 3 | $(4,48,12,16)$ | $(12,16) \circ (16,48) \circ (4,16)$ | 11 | $(20,52,60,28)$ | $(28,52) \circ (52,60) \circ (20,28)$ |
| 4 | $(6,18,38,26)$ | $(18,26) \circ (18,38) \circ (6,26)$ | 12 | $(23,39,43,27)$ | $(27,39) \circ (39,43) \circ (23,27)$ |
| 5 | $(7,35,59,31)$ | $(31,35) \circ (35,59) \circ (7,31)$ | 13 | $(24,36,56,44)$ | $(36,44) \circ (36,56) \circ (24,44)$ |
| 6 | $(8,32)$ | $(8,32) \circ \quad \circ$ | 14 | $(30,54)$ | $(30,54) \circ \quad \circ$ |
| 7 | $(9,49,29,37)$ | $(29,37) \circ (37,49) \circ (9,37)$ | 15 | $(41,57,61,45)$ | $(45,57) \circ (57,61) \circ (41,45)$ |

As per the strategies outlined in the case of PRESENT, we try to implement all the $u_i$'s first, followed by the $t_i$'s and $s_i$'s, since as per Lemma 3, we would have then constructed $G$. Furthermore for each of the $u_i$'s, $t_i$'s and $s_i$' we construct composite $\overrightarrow{\mathsf{Sel}}$ vectors by finding $\overrightarrow{\mathsf{Sel}}$ vectors for each equivalence class modulo 4 and then doing a bitwise OR. Each of the transposition sets can be implemented in $9 \times 64 = 576$ cycles. This implies that $G$ can be executed in $3 \times 576 = 1728$ cycles. The results are tabulated in Table 13 in the appendix.

## 5.1 Circuit details

Since the structure of GIFT is similar to PRESENT, the circuit for the datapath is exactly the same as in Figure 3, with the obvious exception that the scan flip-flop is used in the $60^{th}$ instead of the $61^{st}$ location. The sequence of operations in GIFT is only slightly different from PRESENT and we leave the operational details of the circuit to Appendix H. The encryption only circuit occupies 1132 GE which is around 200 GE higher than the implementation reported in [BPP+17], on account of the large control bit table required in the circuit. The encryption+decryption circuit occupies 1290 GE in hardware and is the first reported synthesis results for the combined circuit for this block cipher.

# 6 Adding more scan flip-flops

In this section, we look at trade-off between the number of scan flip-flops and the latency of the permutation layer. In other words, we employ multiple scan flip-flops to complete the permutation layer operation in at most few hundreds of cycles.

In order to understand how more scan flip-flops can be accommodated, let us start with the basics. Let $\mathsf{rot}$ denote the simple rotation operation of the pipeline, that is $\mathsf{rot} \in S_{64}$ such that $\mathsf{rot}(i) = i+1 \bmod 64$. Then we additionally introduce swap-then-rotate operations to this pipeline. A swap-then-rotate operation is denoted with $\mathsf{swp}_{(x,y)}$, and it first swaps $x$ and $y$, and then rotates the pipeline. Namely,

$$\mathsf{swp}_{(x,y)}(x) = y + 1 \bmod 64, \qquad \mathsf{swp}_{(x,y)}(y) = x + 1 \bmod 64$$

and the others elements remain untouched. We have already seen that a swap-then-rotate operation can be done in the pipeline quite efficiently, i.e. requires only two extra muxes before inputs of flip flops $x+1$ and $y+1$. Our technique for reducing the number of required gates of the permutation layers of both PRESENT and GIFT is through realizing them with $\mathsf{rot}$ and as few as possible $\mathsf{swp}$ operations. We extend the notion into multiple-swaps-then-rotate in a natural way. If $\{x, y\} \cap \{z, t\} = \emptyset$, then $\mathsf{swp}_{(x,y),(z,t)}$ corresponds to swapping both $(x, y)$ and $(z, t)$ first, and then rotating the pipeline by one position. We invite the reader's attention to the difference between the *swap operations*, e.g. $\mathsf{swp}_{(x,y)}$, and *swap permutations*, e.g. 2-cycle $(x, y)$.

## 6.1 $4 \times 4$ matrix transposition with swaps

For simplicity, first imagine a pipeline that consists of $4 \times 4$ bits (see Figure 5). Suppose that the pipeline supports only $\mathsf{rot}$ and $\mathsf{swp}_{(11,14)}$ operations. These two permutations are given in their mathematical forms in Table 6, where $\mathsf{rot}(i)$ denotes the final position of the bit $i$ after $\mathsf{rot}$ is executed.

Our claim is that the usual $4 \times 4$ matrix transposition $\tau$ can be written in terms of $\mathsf{rot}$ and $\mathsf{swp}_{(11,14)}$ permutations. Namely, our formula is $\tau = \mathsf{seq3} \circ \mathsf{seq2} \circ \mathsf{seq1}$ where

$$\begin{aligned}\mathsf{seq1} &:= \mathsf{rot}^4 \circ \Big[\mathsf{rot} \circ \mathsf{swp}^3_{(11,14)}\Big]^3 \\ \mathsf{seq2} &:= \mathsf{rot}^8 \circ \Big[\mathsf{rot} \circ \mathsf{swp}^2_{(11,14)} \circ \mathsf{rot}\Big]^2 \\ \mathsf{seq3} &:= \mathsf{rot}^{13} \circ \mathsf{swp}_{(11,14)} \circ \mathsf{rot}^2.\end{aligned}$$

This is demonstrated in Figure 5, and derivation of the sequence is explained in Section 6.4. In conclusion, performing a transposition $\tau$ requires three full rotations of the pipeline, i.e. takes $3 \times 16$ cycles, with a single swap operation.

In order to optimize the number of clock cycles spent for each $\tau$ application, we can add one or two more swaps into the pipeline. Hence, there is a trade-off between the number of cycles and the circuit area required to execute the permutation. The sequences of operations with two and three swap operations are demonstrated in Figure 6, and further explanations regarding working mechanism of $\tau$ is given in Section 6.4.

**Table 6:** Mathematical forms of some permutations over $S_{16}$

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathsf{rot}(i)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 |
| $\mathsf{swp}_{(11,14)}(i)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 14 | 12 | 13 | 11 | 15 | 0 |
| $\tau(i)$ | 0 | 4 | 8 | 12 | 1 | 5 | 9 | 13 | 2 | 6 | 10 | 14 | 3 | 7 | 11 | 15 |
| $\sigma(i)$ | 0 | 5 | 10 | 15 | 12 | 1 | 6 | 11 | 8 | 13 | 2 | 7 | 4 | 9 | 14 | 3 |

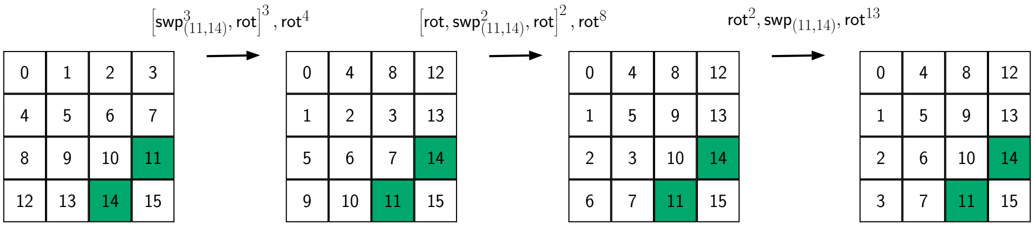

$[\mathsf{swp}^3_{(11,14)}, \mathsf{rot}]^3, \mathsf{rot}^4$
$[\mathsf{rot}, \mathsf{swp}^2_{(11,14)}, \mathsf{rot}]^2, \mathsf{rot}^8$
$\mathsf{rot}^2, \mathsf{swp}_{(11,14)}, \mathsf{rot}^{13}$
0 1 2 3 / 4 5 6 7 / 8 9 10 11 / 12 13 14 15
0 4 8 12 / 1 2 3 13 / 5 6 7 14 / 9 10 11 15
0 4 8 12 / 1 5 9 13 / 2 3 10 14 / 6 7 11 15
0 4 8 12 / 1 5 9 13 / 2 6 10 14 / 3 7 11 15

**Figure 5:** A transposition can be done with $\mathsf{rot}$ and $\mathsf{swp}_{(11,14)}$ in $3 \times 16$ cycles. The operations separated by comma are executed in leftmost-first fashion. The cells corresponding to fixed swap positions of $\mathsf{swp}_{(11,14)}$ are marked with green.

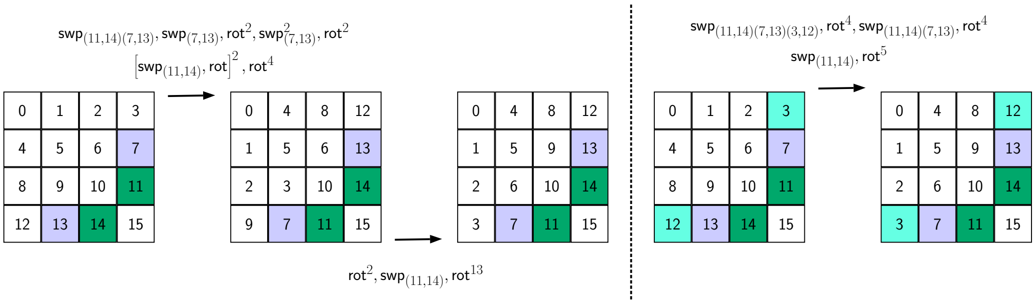

$\mathsf{swp}_{(11,14)(7,13)}, \mathsf{swp}_{(7,13)}, \mathsf{rot}^2, \mathsf{swp}^2_{(7,13)}, \mathsf{rot}^2$
$[\mathsf{swp}_{(11,14)}, \mathsf{rot}]^2, \mathsf{rot}^4$
$\mathsf{rot}^2, \mathsf{swp}_{(11,14)}, \mathsf{rot}^{13}$
$\mathsf{swp}_{(11,14)(7,13)(3,12)}, \mathsf{rot}^4, \mathsf{swp}_{(11,14)(7,13)}, \mathsf{rot}^4$
$\mathsf{swp}_{(11,14)}, \mathsf{rot}^5$
0 1 2 3 / 4 5 6 7 / 8 9 10 11 / 12 13 14 15
0 4 8 12 / 1 5 6 13 / 2 3 10 14 / 9 7 11 15
0 4 8 12 / 1 5 9 13 / 2 6 10 14 / 3 7 11 15
0 1 2 3 / 4 5 6 7 / 8 9 10 11 / 12 13 14 15
0 4 8 12 / 1 5 9 13 / 2 6 10 14 / 3 7 11 15

**Figure 6:** Transposition $\tau$ can also be done in 128 (resp. 64) cycles with 2 (resp. 3) swap operations.

**Table 7:** Specifications of PRESENT bit-permutation layer. Vertical matrix $M_3$ and a horizontal matrix $N_0$ are colored with green and blue colors respectively and their shared bits are colored with dark blue-green.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P(i)$ | 0 | 16 | 32 | 48 | 1 | 17 | 33 | 49 | 2 | 18 | 34 | 50 | 3 | 19 | 35 | 51 |
| $i$ | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| $P(i)$ | 4 | 20 | 36 | 52 | 5 | 21 | 37 | 53 | 6 | 22 | 38 | 54 | 7 | 23 | 39 | 55 |
| $i$ | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| $P(i)$ | 8 | 24 | 40 | 56 | 9 | 25 | 41 | 57 | 10 | 26 | 42 | 58 | 11 | 27 | 43 | 59 |
| $i$ | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| $P(i)$ | 12 | 28 | 44 | 60 | 13 | 29 | 45 | 61 | 14 | 30 | 46 | 62 | 15 | 31 | 47 | 63 |

## 6.2 From Transpositions to PRESENT Permutation

Now we show how to decompose the PRESENT permutation $P$. The crucial observation is that the permutation $P$ from $S_{64}$ can be written in terms of eight applications of $\tau$ from $S_{16}$ so long as we chop 64 bits into four 16-bit matrices in a careful manner. We further use the pipeline rotation to use the same swap operation to perform $\tau$ operation on different sub-matrices of the pipeline.

In the first pass, we divide 64 positions $\{0, \dots, 63\}$ into four *vertical* disjoint matrices, that is we construct $M_0, M_1, M_2, M_3$ such that $i$-th row, $j$-th column of $M_r$ is $16i + j + 4r$ (where columns/rows are indexed from 0 to 3). Then we apply $\tau$ on each $M_r$. In the second pass, we construct *horizontal* matrices $N_r$ such that $i$-th row, $j$-th column of $N_r$ is $4i + j + 16r$. Again, $\tau$ is applied over each $N_r$. The choices of 16 indices for $M_3$ and $N_0$ are demonstrated in Table 7.

More formally, let $Z$ denote an ordered subset $\{z_0, z_1, \dots, z_{15}\}$ of $\{0, \dots, 63\}$ (equivalently $Z$ can be considered as a $4 \times 4$ matrix). Then we define the permutation $\tau_Z \in S_{64}$ as applying $\tau \in S_{16}$ over $Z$ while keeping the other 48 bits untouched. Which is to say, given $i \in \{0, \dots, 63\}$, if $i = z_j$ for some $j$ then $\tau_Z(i) = \tau_Z(z_j) = z_{\tau(j)}$, and otherwise (if $i \notin Z$) then $\tau_Z(i) = i$. Our claim is that $P = \tau_{N_0} \circ \tau_{N_1} \circ \tau_{N_2} \circ \tau_{N_3} \circ \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$.

For a particular choice of $Z$, we need to consider that $\tau_Z \in S_{64}$ can differ from $\tau \in S_{16}$ in two ways: 1) the positions of $(x, y)$ in $\mathsf{swp}_{(x,y)}$ operation and 2) how many $\mathsf{rot} \in S_{64}$ application it takes to complete a full rotation in $Z$. For the former, we need to choose $(z_{11}, z_{14})$ as swap positions instead of $(11, 14)$. For the latter, we need to update our schedule of operations. For instance $M_3$ requires 64 cycles of $\mathsf{rot}$ to complete its full rotation instead of 16. That means during $\tau_{M_r}$ operations, $\mathsf{rot} \in S_{64}$ that rotates the pipeline is actually different than the one we used previously, i.e. $\mathsf{rot} \in S_{16}$, to formulate $\tau \in S_{16}$. In particular, since the pipeline consists of 64 bits, it takes 16 cycles for the second row of $M$ to move to its first row. Hence, we need to update our decomposition sequences to interleave $\tau_{M_r}$ operations.

We interleave $\tau_M$ operations as follows. Given

$$\mathsf{seq1} := \mathsf{rot}^{16} \circ \left[\mathsf{rot} \circ \mathsf{swp}^3_{(47,62)}\right]^{12}$$

$$\mathsf{seq2} := \mathsf{rot}^{32} \circ \left[\mathsf{rot} \circ \mathsf{swp}^2_{(47,62)} \circ \mathsf{rot}\right]^{8}$$

$$\mathsf{seq3} := \mathsf{rot}^{48} \circ \left[\mathsf{rot} \circ \mathsf{swp}_{(47,62)} \circ \mathsf{rot}^2\right]^{4}$$

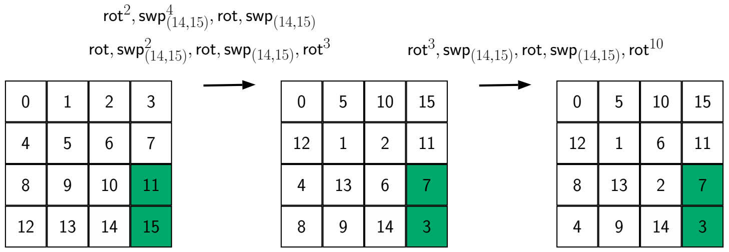

$\mathsf{rot}^2, \mathsf{swp}^4_{(14,15)}, \mathsf{rot}, \mathsf{swp}_{(14,15)}$
$\mathsf{rot}, \mathsf{swp}^2_{(14,15)}, \mathsf{rot}, \mathsf{swp}_{(14,15)}, \mathsf{rot}^3$
$\mathsf{rot}^3, \mathsf{swp}_{(14,15)}, \mathsf{rot}, \mathsf{swp}_{(14,15)}, \mathsf{rot}^{10}$
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
0 5 10 15
12 1 2 11
4 13 6 7
8 9 14 3
0 5 10 15
12 1 6 11
8 13 2 7
4 9 14 3

**Figure 7:** Performing $\sigma$ with $\mathsf{swp}_{(11,15)}$ and $\mathsf{rot}$ in 2 full rounds, i.e $2 \times 64$ cycles.

then $\mathsf{seq3} \circ \mathsf{seq2} \circ \mathsf{seq1} = \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$. And for $\tau_N$ operations, given

$$\mathsf{seq4} := \left[\mathsf{rot}^4 \circ \left[\mathsf{rot} \circ \mathsf{swp}^3_{(59,62)}\right]^3\right]^4$$

$$\mathsf{seq5} := \left[\mathsf{rot}^8 \circ \left[\mathsf{rot} \circ \mathsf{swp}^2_{(59,62)} \circ \mathsf{rot}\right]^2\right]^4$$

$$\mathsf{seq6} := \left[\mathsf{rot}^{13} \circ \mathsf{swp}_{(59,62)} \circ \mathsf{rot}^2\right]^4$$

then $\mathsf{seq6} \circ \mathsf{seq5} \circ \mathsf{seq4} = \tau_{N_0} \circ \tau_{N_1} \circ \tau_{N_2} \circ \tau_{N_3}$. Finally, $P = \mathsf{seq6} \circ \mathsf{seq5} \circ \mathsf{seq4} \circ \mathsf{seq3} \circ \mathsf{seq2} \circ \mathsf{seq1}$. The full worked-out schedules and decomposition of PRESENT permutation is given in Table 9. Note that we require 2 different swaps (therefore 4 scan flip-flops), to work this out. Each seqi requires 64 cycles and hence the permutation can be realized with $6 \cdot 64$ cycles.

## 6.3 From Transpositions to GIFT Permutation

The decomposition of GIFT permutation $G$ is slightly different than $P$. We choose our matrices such that the first operation becomes transposition $\tau$ over nibbles instead of bits, and the second one consists of series of ad hoc swaps described as the permutation $\sigma$ in Table 6. In the same fashion, $\sigma$ is a permutation over $S_{16}$, but we can extend it to $S_{64}$ by defining $\sigma_Z$ for $Z$ being an ordered subset of $\{0, \dots, 63\}$ as before.

Performing $G$ takes four applications of $\tau$ followed by four applications of $\sigma$. We choose our matrices as follows. The $i$-th row, $j$-th column of $M_r$ is $4i+16j+r$. Then we apply $\tau$ on $M_r$ matrices. In the second pass, the $i$-th row, $j$-th column of $N_r$ is $16i+j+4r$. Again, $\sigma$ is applied over $N_r$ matrices. Our finding is that $G = \sigma_{N_0} \circ \sigma_{N_1} \circ \sigma_{N_2} \circ \sigma_{N_3} \circ \tau_{M_0} \circ \tau_{M_1} \circ \tau_{M_2} \circ \tau_{M_3}$. The sequence of operations to realize $G$ with minimum number of swaps are presented in Table 10.

## 6.4 Reducing Cycles and Decryption

In order to complete $P$ and $G$ permutations in fewer number of cycles, we can introduce one or two more additional swap operations to the pipeline. The sequence of operations for a reduced number of cycles are given in Tables 9, 10. Below we explain the intuition of how we can derive a sequence of swap and rotations to execute transposition $\tau$, and how we can trade one or two more swap operations with a number of cycles.

In Figure 5, the execution of $\tau$ in three rounds, i.e. $3 \times 16$ cycles, is given. For simplicity, consider $\tau$ in terms of an (arbitrary-order) composition of the swaps: $(1,4)$, $(2,8)$, $(3,12)$, $(6,9)$, $(7,13)$, $(11,14)$. Equipped with $\mathsf{swp}_{(11,14)}$ (denoted with green cells) that swaps two neighbors of the rightmost bottom cell (i.e. the cells initially storing $11, 14$), we begin constructing our sequence. Since both $\mathsf{swp}$ and $\mathsf{rot}$ operations move the pipeline (the direction of the pipeline is such that the bit at $i$ moves to $\mathsf{rot}(i)$ as given in Table 6), we

**Table 8:** Specifications of GIFT bit-permutation layer.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $G(i)$ | 0 | 17 | 34 | 51 | 48 | 1 | 18 | 35 | 32 | 49 | 2 | 19 | 16 | 33 | 50 | 3 |
| $i$ | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| $G(i)$ | 4 | 21 | 38 | 55 | 52 | 5 | 22 | 39 | 36 | 53 | 6 | 23 | 20 | 37 | 54 | 7 |
| $i$ | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| $G(i)$ | 8 | 25 | 42 | 59 | 56 | 9 | 26 | 43 | 40 | 57 | 10 | 27 | 24 | 41 | 58 | 11 |
| $i$ | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| $G(i)$ | 12 | 29 | 46 | 63 | 60 | 13 | 30 | 47 | 44 | 61 | 14 | 31 | 28 | 45 | 62 | 15 |

can actually perform swap between any pair of bits initially located at row $x$, column $y$ and row $x-1$, column $y+1$ for $x = 1, 2, 3$, $y = 0, 1, 2$. All we need to do is to wait until the sufficient number of rot/swp operations are performed so that bit that is initially at row $x$, column $y$ arrives at 14 and the pair can be swapped during that cycle with $\mathsf{swp}_{(11,14)}$. However, notice that the operation $\mathsf{swp}_{(11,14)}$ does not allow us to swap bits that do not initially share a corner point, such as bits located at $(7, 13)$ pair. We tackle this by splitting those swaps into multiple pair-swap operations, e.g. $(7, 13)$ can also be done with a sequence of $(13, 10)$, $(10, 7)$, $(13, 10)$ pair-swaps, as they are supported by $\mathsf{swp}_{(11,14)}$ operation. Similarly, we can split combination of $(3, 12)$, $(6, 9)$ pairs into the sequence $(12, 9)$, $(9, 6)$, $(6, 3)$, $(12, 9)$, $(9, 6)$, $(12, 9)$ that is supported by $\mathsf{swp}_{(11,14)}$.

In the two previous examples, a crucial observation is that a sequence might require more than one round of rotation of the pipeline. This is caused by the pipeline movement. For instance, suppose that we need to execute a sequence of pair-swaps such that $(x_0, y_0)$ pair comes later than $(x_1, y_1)$ but $(x_0, y_0)$ arrives to its swap position earlier than $(x_1, y_1)$. Then we need another round to perform $(x_0, y_0)$, as the pipeline only moves in one direction. Similarly, repetition of swaps causes extra rounds to be incurred, as swaps themselves also move the pipeline. This is the exact reason why for instance the sequence $(12, 9)$, $(9, 6)$, $(6, 3)$, $(12, 9)$, $(9, 6)$, $(12, 9)$ requires 3 passes in the case of a single swap operation, as $(12, 9)$ repeats thrice. Here the trade-off is much clearer to see. If we relax our restriction on the number of allowed swap operations, e.g. by adding two more swap operations, then we can find better sequences which require less rounds. Below, we consider the case of two swaps.

Suppose that the pipeline supports $\mathsf{swp}_{(11,14)}$ and $\mathsf{swp}_{(7,13)}$ besides rot as given in Figure 6. The second swap operation implies that we can swap pairs such as $(12, 6)$, $(9, 3)$, $(13, 7)$, $(8, 2)$ in an atomic fashion (that is we do not need to split them into sequences of swaps) by waiting sufficient number of rot/swp operations so that the pair of bits move into cells indexed with $(13, 7)$. As the only remaining swap that cannot be done in atomic fashion is $(3, 12)$, we can split it into the sequence $(12, 6), (9, 3), (6, 3), (12, 9)$, which is supported by $\mathsf{swp}_{(7,13)}$, $\mathsf{swp}_{(7,13)}$, $\mathsf{swp}_{(11,14)}$, $\mathsf{swp}_{(11,14)}$ operations respectively. Indeed, one can observe that this sequence of swaps are actually run as soon as their pairs are moved into the correct swap positions in Figure 6.

Finally, if we can spare three swap operations in total, e.g. $\mathsf{swp}_{(11,14)}$, $\mathsf{swp}_{(7,13)}$, $\mathsf{swp}_{(3,12)}$, then we can perform $\tau$ in just one round. This is given in Figure 6. In order to give the intuition, we further explain how we derive the sequence of operations. Recall that we need to perform swaps $(1, 4)$, $(2, 8)$, $(3, 12)$, $(6, 9)$, $(7, 13)$, $(11, 14)$. At the very first cycle, we perform three pair-swaps of $(11, 14)$, $(7, 3)$ and $(3, 12)$ simultaneously, i.e. running $\mathsf{swp}_{(11,14),(7,13),(3,12)}$. Note that since the cells of swaps are not overlapping, we can perform multiple swaps in the same cycle. For the pair-swap of $(6, 9)$ and $(2, 8)$, we

**Table 9:** Cycle vs. Mux trade-off for PRESENT

| | # swaps | round | cycles | decomposition |
|---|---|---|---|---|
| PRESENT | 2 | 1 | 0-47 | $[\mathsf{swp}^3_{(47,62)}, \mathsf{rot}]^{12}$ |
| | | | 48-63 | $\mathsf{rot}^{16}$ |
| | | 2 | 0-31 | $[\mathsf{rot}, \mathsf{swp}^2_{(47,62)}, \mathsf{rot}]^{8}$ |
| | | | 32-63 | $\mathsf{rot}^{32}$ |
| | | 3 | 0-15 | $[\mathsf{rot}^2, \mathsf{swp}_{(47,62)}, \mathsf{rot}]^{4}$ |
| | | | 16-63 | $\mathsf{rot}^{48}$ |
| | | 4 | 0-63 | $([\mathsf{swp}^3_{(59,62)}, \mathsf{rot}]^3, \mathsf{rot}^4)^4$ |
| | | 5 | 0-63 | $([\mathsf{rot}, \mathsf{swp}^2_{(59,62)}, \mathsf{rot}]^2, \mathsf{rot}^8)^4$ |
| | | 6 | 0-63 | $[\mathsf{rot}^2, \mathsf{swp}_{(59,62)}, \mathsf{rot}^{13}]^4$ |
| PRESENT | 4 | 1 | 0-15 | $[\mathsf{swp}_{(47,62),(31,61)}, \mathsf{swp}_{(31,61)}, \mathsf{rot}^2]^4$ |
| | | | 16-47 | $[\mathsf{swp}^2_{(31,61)}, \mathsf{rot}^2]^4, [\mathsf{swp}_{(47,62)}, \mathsf{rot}]^8$ |
| | | | 48-64 | $\mathsf{rot}^{16}$ |
| | | 2 | 0-63 | $[\mathsf{rot}^2, \mathsf{swp}_{(47,62)}, \mathsf{rot}]^4, \mathsf{rot}^{48}$ |
| | | 3 | 0-3, 16-19, 32-35, 48-51 | $\mathsf{swp}_{(59,62),(55,61)}, \mathsf{swp}_{(55,61)}, \mathsf{rot}^2$ |
| | | | 4-15, 20-31, 36-47, 52-63 | $\mathsf{swp}^2_{(55,61)}, \mathsf{rot}^2, [\mathsf{swp}_{(59,62)}, \mathsf{rot}]^2, \mathsf{rot}^4$ |
| | | 4 | 0-63 | $[\mathsf{rot}^2, \mathsf{swp}_{(59,62)}, \mathsf{rot}^{13}]^4$ |
| PRESENT | 6 | 1 | 0-15 | $[\mathsf{swp}_{(47,62),(31,61),(15,60)}, \mathsf{rot}^3]^4$ |
| | | | 16-31 | $[\mathsf{rot}, \mathsf{swp}_{(47,62),(31,61)}, \mathsf{rot}^2]^4$ |
| | | | 32-63 | $[\mathsf{rot}^2, \mathsf{swp}_{(47,62)}, \mathsf{rot}]^4, \mathsf{rot}^{16}$ |
| | | 2 | 0-7, 16-23, 32-39, 48-55 | $\mathsf{swp}_{(59,62),(55,61),(51,60)}, \mathsf{rot}^4, \mathsf{swp}_{(59,62),(55,61)}, \mathsf{rot}^2$ |
| | | | 8-15, 24-31, 40-47, 56-63 | $\mathsf{rot}^2, \mathsf{swp}_{(59,62)}, \mathsf{rot}^5$ |

use the 6-th cycle to run $\mathsf{swp}_{(11,14),(7,13)}$, as they will be located in cells marked with green and purple in Figure 6. Finally, the pair-swap $(1,4)$ will arrive to green cells at 11-th cycle, so we can run $\mathsf{swp}_{(11,14)}$.

The similar tradeoff also applies for $\sigma$ permutation, as it also consists solely of some set of pairs to swap as shown in Figure 7. The main difference is that all pair-swaps can be done atomically with only two swap operations, making it more efficient compared to $\tau$.

One might notice that neither of $G$ and $P$ are involution, that is $P(P(i)) = i$ does not hold, meaning the permutation logic for encryption cannot be readily used in decryption. A straightforward idea for decryption that avoids adding extra gates could be based on the fact that $P^3$ and $G^4$ are identity permutations. Hence one can repeat $P$ and $G$ two and three times respectively to get their inverse permutation. However, this is not an optimal solution, as it double or triples the number of cycles required for the inverse permutation layer, making decryption significantly more costly than encryption.

On the other hand, we draw attention to the fact that our decomposed permutations $\tau$ and $\sigma$ are involutions, as they only swap pair of elements. Hence, for decryption, we only need to change the order of executions. As an example, for PRESENT we only need to run $\tau_{N_r}$ permutations in the first pass, and $\tau_{M_r}$ in the second pass for decryption. The number of cycles and tradeoffs remain exactly same. No extra gates or cycles are required. In conclusion, the advantage of decomposing a permutation with our swap-based technique is twofold: it adds quite small amount of gates (2 extra muxes for each swap), and it readily supports decryption with no extra cost, even if the composed permutation is not an involution and might seem to require some extra gates for its inverse.

## 6.5 Lowering latency further

This section is dedicated to the goal of decreasing the latency. In Section 6, we have shown that various PRESENT and GIFT implementations can be realized with very small

**Table 10:** Cycle vs. Mux trade-off for GIFT

| | # swaps | round | cycles | decomposition |
|---|---|---|---|---|
| GIFT | 2 | 1 | 0-63 | $[\mathsf{swp}^{12}_{(47,62)}, \mathsf{rot}^4]^3, \mathsf{rot}^{16}$ |
| | | 2 | 0-63 | $[\mathsf{rot}^4, \mathsf{swp}^{8}_{(47,62)}, \mathsf{rot}^4]^2, \mathsf{rot}^{32}$ |
| | | 3 | 0-63 | $\mathsf{rot}^8, \mathsf{swp}^{4}_{(47,62)}, \mathsf{rot}^{52}$ |
| | | 4 | 0-31 | $[\mathsf{rot}^2, \mathsf{swp}^{2}_{(47,63)}]^4, [\mathsf{swp}^{2}_{(47,63)}, \mathsf{rot}, \mathsf{swp}_{(47,63)}]^4$ |
| | | | 32-63 | $[\mathsf{rot}, \mathsf{swp}^{2}_{(47,63)}, \mathsf{rot}]^4, [\mathsf{swp}_{(47,63)}, \mathsf{rot}^3]^4$ |
| | | 5 | 0-63 | $[\mathsf{rot}^3, \mathsf{swp}_{(47,63)}]^4, [\mathsf{rot}, \mathsf{swp}_{(47,63)}, \mathsf{rot}^2]^4, \mathsf{rot}^{32}$ |
| GIFT | 4 | 1 | 0-23 | $\mathsf{swp}^{4}_{(47,62),(31,61),(15,60)}, \mathsf{rot}^{16}, \mathsf{swp}^{4}_{(47,62),(31,61)}$ |
| | | | 25-63 | $\mathsf{rot}^{16}, \mathsf{swp}^{4}_{(47,62)}, \mathsf{rot}^{20}$ |
| | | 2 | 0-31 | $[\mathsf{rot}^2, \mathsf{swp}^{2}_{(47,63)}]^4, [\mathsf{swp}^{2}_{(47,63)}, \mathsf{rot}, \mathsf{swp}_{(47,63)}]^4$ |
| | | | 32-63 | $[\mathsf{rot}, \mathsf{swp}^{2}_{(47,63)}, \mathsf{rot}]^4, [\mathsf{swp}_{(47,63)}, \mathsf{rot}^3]^4$ |
| | | 3 | 0-63 | $[\mathsf{rot}^3, \mathsf{swp}_{(47,63)}]^4, [\mathsf{rot}, \mathsf{swp}_{(47,63)}, \mathsf{rot}^2]^4, \mathsf{rot}^{32}$ |
| GIFT | 5 | 1 | 0-23 | $\mathsf{swp}^{4}_{(47,62),(31,61),(15,60)}, \mathsf{rot}^{16}, \mathsf{swp}^{4}_{(47,62),(31,61)}$ |
| | | | 25-63 | $\mathsf{rot}^{16}, \mathsf{swp}^{4}_{(47,62)}, \mathsf{rot}^{20}$ |
| | | 2 | 0-31 | $[\mathsf{rot}^2, \mathsf{swp}_{(47,63)}, \mathsf{swp}_{(30,62)}]^4, [\mathsf{swp}_{(47,63)}, \mathsf{swp}_{(30,62)}, \mathsf{rot}^2]^4$ |
| | | | 32-63 | $[\mathsf{rot}^2, \mathsf{swp}_{(47,63)}, \mathsf{rot}]^4, [\mathsf{swp}_{(30,62)}, \mathsf{rot}^3]^4$ |

additional cost, i.e. 4 to 12 scan flip-flops (i.e. 2 to 6 swaps). Even though our approach achieves roughly 20 % reduction in the circuit area, it causes the latency of the circuit to increase to threefold. Hence, in this section we show that by carefully arranging all swap operations to run concurrently, we can beat the state-of-the-art implementations of PRESENT and GIFT [JMPS17], in terms of both latency and circuit-size.

Building upon our finding in Section 6, we provide realization of PRESENT and GIFT permutations with 6 swaps that require no additional clock cycles. While encryption/decryption rounds take precisely 64 cycles to complete for each round (add round key and sbox), our permutation layer operates on the state pipeline seamlessly to ensure that each bit leaving the pipeline is already moved to its permuted position. There is no need to freeze the state pipeline or allocate extra clock cycles to the permutation layer either. In comparison, the smallest known implementation from Jean et al. [JMPS17] requires 4 additional cycles each round, leading to a loss of more than a hundred cycles in latency. This is because the additional circuitry that handles the permutation layer requires four cycles to complete the permutation, during which add round key and sbox layers must be stalled. Our implementation of permutation layer, on the other hand, reaches to maximum utilization in a bit-serial implementation architecture, as it brings no additional cycles.

The intuitive idea is to use the core idea of Lemma 3, which, in informal sense, states that (disjoint) cycles can be applied in any order. Since swaps are simply 2-cycles, given $(a, b)$, $(c, d)$ checking whether they are disjoint is straightforward by $a \neq d \wedge a \neq c \wedge b \neq c \wedge b \neq d$. On the contrary, if two dependent swaps are given, e.g. $(a, b)$, $(b, c)$, then we must preserve the order between them. At this point, one needs to be cautious about which exact 2-cycle is run at some given clock cycle, as swp operations on the hardware actually operate at different 2-cycles. For instance, $\mathsf{swp}_{(11,14)}$ performs to the set of 2-cycles $\{(11 + i \bmod 64, 14 + i \bmod 64)\}$ for the clock cycles $i$ in which it is active. Hence, if we expand the operation sequences that leads to two rounds from Tables 9, 10 into a series of actual 2-cycles applications by replacing each $\mathsf{swp}_{(x,y)}$ at active clock cycle $i$ with 2-cycle $(x + i \bmod 64, y + i \bmod 64)$, the following question arises:

*Can the expanded sequence of 2-cycles (which takes 128 according to Section 6) clock cycles be squeezed into fewer number of clock cycles (close to 64) so that we can complete*

**Table 11:** Realization of GIFT and PRESENT permutations in 64 cycles.

| | mode | swap | active cycles (round $i$) | active cycles (round $i+1$) |
|---|---|---|---|---|
| PRESENT | ENC | (20, 5) | 22, 26, 30, 34, 39, 43, 47, 51, 56, 60 | 0, 4 |
| | | (34, 4) | 37, 41, 45, 49, 54, 58, 62 | 2 |
| | | (48, 3) | 52, 56, 60 | 0 |
| | | (60, 57) | 62 | 3, 8, 14, 19, 24, 30, 35, 40, 46, 51, 56 |
| | | (61, 55) | | 0, 5, 16, 21, 32, 37, 48, 53 |
| | | (62, 53) | | 2, 18, 34, 50 |
| PRESENT | DEC | (20, 5) | 33, 37, 41, 45, 50, 54, 58, 62 | 3, 7, 11, 15 |
| | | (34, 4) | 48, 52, 56, 60 | 1, 5, 9, 13 |
| | | (48, 3) | 63 | 3, 7, 11 |
| | | (60, 57) | 9, 14, 19, 25, 30, 35, 41, 46, 51, 57, 62 | 3 |
| | | (61, 55) | 11, 16, 27, 32, 43, 48, 59 | 0 |
| | | (62, 53) | 13, 29, 45, 61 | |
| GIFT | ENC | (24, 12) | 29, 30, 31, 32, 49, 50, 51, 52 | 5, 6, 7, 8 |
| | | (37, 13) | 46, 47, 48, 49 | 2, 3, 4, 5 |
| | | (50, 14) | 63 | 0, 1, 2 |
| | | (61, 45) | | 0, 4, 8, 12, 14, 18, 22, 26, 32, 36, 40, 44 |
| | | (62, 30) | | 2, 6, 10, 14, 16, 20, 24, 28 |
| | | (63, 15) | | 0, 4, 8, 12 |
| GIFT | DEC | (56, 44) | | 0, 1, 2, 3, 20, 21, 22, 23, 40, 41, 42, 43 |
| | | (55, 31) | | 3, 4, 5, 6, 23, 24, 25, 26 |
| | | (50, 14) | | 2, 3, 4, 5 |
| | | (61, 45) | 49, 53, 57, 61 | |
| | | (62, 30) | 37, 41, 45, 49, 51, 55, 59, 63 | |
| | | (63, 15) | 21, 25, 29, 33, 35, 39, 43, 47, 53, 57, 61 | 1 |

*the permutation layer in one pass (single round), with the help of Lemma 3?*

Fortunately, the answer to this question is affirmative. Furthermore, if we make our choices for the initial swap operations wisely, we can even use the exact same swap operations in a combined ENC/DEC circuit for PRESENT permutation. For GIFT permutation, we need to add two more swaps for the combined circuit. The fully worked-out schedule and the carefully chosen six swaps are given in Table 11, for both encryption and decryption circuits. In summary, we achieve the following permutation layer implementations:

1. PRESENT ENC only: 6 swaps, 64 clock cycles per permutation
2. PRESENT ENC/DEC: 6 swaps, 64 clock cycles per permutation
3. GIFT ENC only: 6 swaps, 64 clock cycles per permutation
4. GIFT ENC/DEC: 8 swaps, 64 clock cycles per permutation

## 6.6 Circuit Details

With the improved swap sequences from Table 11, we are able to construct state pipelines that constantly run, both for PRESENT and GIFT. Each round of the state is 64 bits, and the round function consists of adding the key for each bits, passing each nibble through S-box, and finally permuting positions of the bits. This means that the key pipeline needs to keep up with the state pipeline for key addition, and provide the correct bits without skipping a clock cycle. This does not constitute a problem for PRESENT, but for GIFT, we need to improve the way we manage the key pipeline. Therefore, the key scheduling for GIFT that can constantly run is presented in Section 6.7.

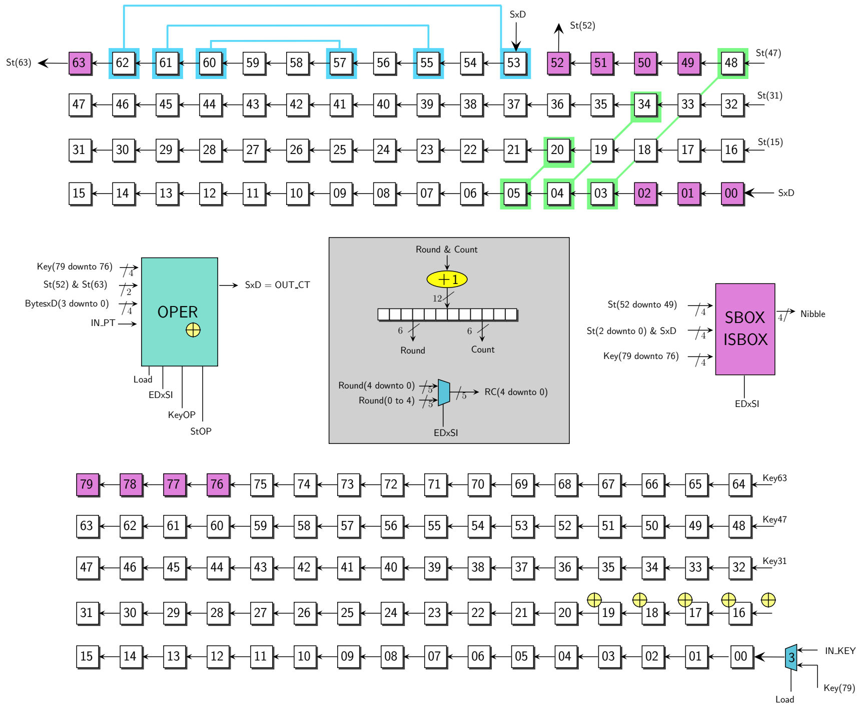


**Figure 8:** Combined ENC+DEC PRESENT circuit using only 6 swaps with no permutation latency

The details for both PRESENT and GIFT circuits for both ENC and combined ENC/DEC variants differ from the previous design presented in Sections 4, 4.2, 5.1 in the following ways:

- For both encryption only and combined circuits, each round takes exactly 64 clock cycles, during which key addition, sbox and permutation layers are always active. In total, we spend 80 clock cycles to load the key and the state simultaneously, and do not perform any operation while loading. Then, we start our round operations. We spend 31 × 64 cycles to complete 31 rounds, and use the final 32nd round to complete the last key addition and output the results simultaneously. In total it takes 2128 clock cycles.

- During encryption, the key and the state are loaded as in the specified order, i.e. $k_{79}$ first and $k_0$ last. During decryption, we reverse the order of the key and the state, so that the direction of the rotation naturally suits both encryption and decryption at the same time without losing any clock cycles. As an example, we load $k_0$ first and $k_{79}$ last during decryption.

- During encryption, each bit enters into the state pipeline at flip flop 00 and exits from flip flop 63 (see Figure 8). However, during decryption each bit enters into the pipeline from flip flop 53, and exits from 52.

- Since the key pipeline needs to always provide one bit at each clock cycle, for PRESENT, we stop its movement at every last 3 clock cycles out of 64 during encryption. We use a combinatorial logic (a combination of muxes) to select the correct key bit, by reaching to the remaining key bits that reside in key flip flops 78, 77, 76. During decryption, we actually skip cycles 60, 61, 62 (but not the last one at 63), which requires us to extend the bit selection logic down to key flip flop 75.

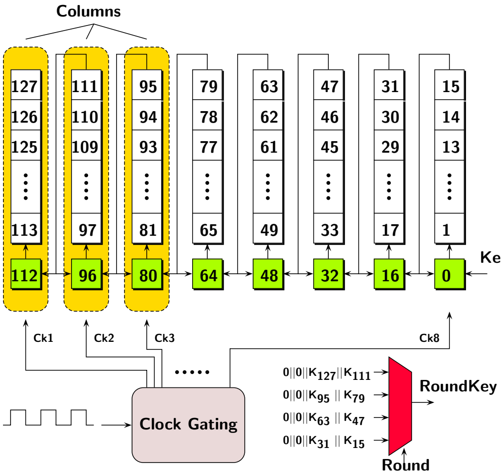


**Figure 9:** GIFT key schedule circuit

- Except cycle 0, the state pipeline actually contains bits from round $i$ and round $i+1$ at the same time. Given that swaps are constantly operating, roughly half of the operations they perform update the state from round $i$, and the remaining operations modify the state $i+1$. In other words, an actual permutation layer operation of the state bits are divided into two rounds. This can be seen more clearly in Table 8.

## 6.7 GIFT key schedule: Reduction of power+latency

The keyschedule of GIFT significantly differs in the sense that it consists of eight 16 bit columns that are shuffled in a 128 bit register in a peculiar way. Recall the key schedule function of GIFT ($L_i$ ($i \in [0,7]$) are the columns):

$$L_7||L_6||\cdots||L_0 \;\leftarrow\; L_1 \ggg 2||L_0 \ggg 12||L_7||\cdots||L_2$$

Each 32 bit block $M_i = L_{2i+1}||L_{2i}$ gets rotated by 32 positions in each cycle, along with internal rotation of 2, 12 bits within each column. This means two things: first there is no mixing of key bits between two different 32 bit blocks and second every four cycles, the individual blocks are back to their starting positions in the registers.

In the original design paper of GIFT [BPP$^+$17], the authors had reported a 96 cycle per round bit serial implementation. Since there are no circuit level details provided in the paper, this was probably because not only did the circuit have to produce the appropriate round key bits during the key addition phase, but it also had to prepare the key register for the next round. In the first 64 cycles, the last 2 columns would provide the round key bits to the GIFT state update pipeline (and at the same time undergo the internal column rotation), and the final 32 bit rotation along with the 2 sets of internal rotations would be done in the subsequent 32 cycles. Although this seems to be the most natural way to work the circuit, there are 2 issues to this ideology:

- This requires the key bits to be rotated through the full length of the key register every four rounds. This contributes to waste of power and energy, considering we have not utilized the fact that there is no inter-block mixing in the keybits.
- There seems to be no straightforward method to bring down the number of cycles required per round to less than 96.

In this subsection we look at an alternate keyschedule for the GIFT cipher that solves both the above problems with slight increase in hardware area. In Figure 9, we present the diagrammatic representation of the circuit. Note that each of the key register columns have a scan flip-flop at the bottom (shown in green), and are wired to do both serially push key bits into the register and to do internal rotation in the columns. The first connection is used only during the load stage to insert all the 128 key bits in the register. We describe its functioning briefly as follows:

**Only internal rotation** We find that that it is not necessary to rotate the key bits across the entire length of the register. Instead we limit ourself to only internal rotation in the columns. Note that the block $M_i$ ($i \in [0,3]$) supplies key material for addition in every 4th round (specifically the rounds $4 \cdot t + i$). Since there is no mixing among the key material in the blocks, we could simply extract key bits from each block and use a 4-to-1 multiplexer (controlled by the current round) to filter the appropriate key material in each round.

**Smart use of clock gating** In every 64 clock cycles the key pipeline has to supply 32 key bits to the state pipeline for addition. In round 1, this is usually the keybits $k_{31}||k_{15}$ (of the current key) for the first state nibble, $k_{30}||k_{14}$ for the second state nibble and so on. In a bit serial datapath, it takes the state bits 4 cycles to be appropriately positioned for the addition. Therefore, we can extract the key bits from the to flip-flop of the 2 least significant columns for the first nibble. Every 4 cycles the columns would internally rotate so that the next keybits ($k_{30}||k_{14}$) are moved to the top flip-flops of the columns for the next nibble addition. This when done 16 times over 16 cycles solves the key addition function requirements. Since key material is extracted from different blocks in every round, all the columns must be rotated in this manner in the appropriate round (i.e. once in every 4 clock cycles). This requires some fine grained control over the clock gating circuit.

**Key Update** Rotating every column once in every 4 cycles over 64 cycles, brings about the identity transformation, and so we still have to solve the 2, 12 bit internal rotation required to update the key. The solution to this is simple and requires some more control over the clock gating circuit. Note that once a block $M_i$ is used for key addition, it is not required to supply key bits for another 3 rounds. So if the $M_i$ block supplies key bits at round $j$, the round $j+1$ can be used to rotate the individual columns by 2, 12 bits. So in essence this means that we can easily accommodate keyschedule in 64 cycles.

**Encrypt/Decrypt** This circuit can be easily adopted to perform decryption. The only changes required in decryption is that the key update requires 32 bit rotation in the opposite direction, and the internal rotation be done by 14, 4 bits respectively. While the latter change can be easily accommodated by updating the clock gating circuit, the former change only implies that the order in which the individual blocks/columns supply key material in the successive rounds is reversed. Thus a simple tweak to the logic block that produces select signals for the multiplexer is sufficient for this purpose.

## 6.8 Area and throughput results

Figure 8 shows a block level diagram of PRESENT using 12 scan flip-flops. The encryption only circuits of occupy 694 and 907 GE respectively. These are the lowest reported in the literature so far, and they achieve the maximum utilization by processing exactly 64 bits each cycle. In the combined encryption+decryption architecture, we have implementations of both PRESENT and GIFT occupying 786 and 1055 GE respectively. The combined circuit also attains the same latency. These are also the lowest yet reported in literature.

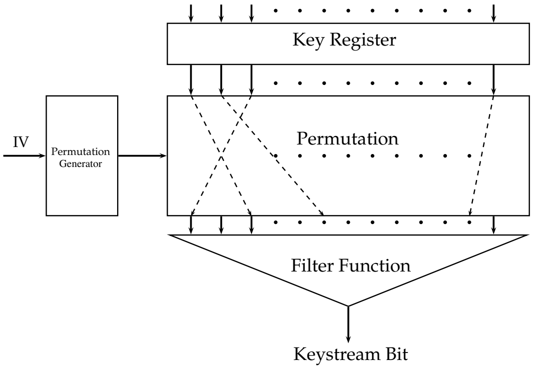


**Figure 10:** The FLIP stream cipher

# 7 Application to FLIP (how to do Knuth shuffles in constrained hardware)

FLIP is a family of stream ciphers proposed by Méaux et al. at Eurocrypt 2016 for FHE based applications. In [MJSC16] the authors suggested a stream cipher based solution to implement the above. The FLIP family stream ciphers have the lowest multiplicative depth compared with previous ciphers. Several versions are provided including 80-bit and 128-bit security instantiations. The main design principle is to filter a constant key register with a time-varying public bit permutation. For 80-bit security the authors suggest the use of the instance FLIP $(42, 128, ^8\Delta^9)$ which uses a 530-bit secret key with hamming weight 265. The internal state $State_i$ of the cipher is a permutation of the original secret key $sk$. The cipher works as follows:

- Let $sk_S \in \{0,1\}^{530}$ with $HW(sk) = 265$.
- For $i = 1 \rightarrow n$ do
  1. Choose a random permutation $P_i$ from the symmetric group $S_{530}$. ($P_i$ may be a function of IV)
  2. Let $State_i = P_i(sk)$.
  3. Compute $z_i = F(State_i)$.

In the above definition, $F$ is a $\{0,1\}^{530} \rightarrow \{0,1\}$ Boolean function of multiplicative depth 4. It consists of a linear function of 42 variables, a quadratic bent function of 128 variables and the remaining 360 variables are used to construct 8 triangular functions of algebraic degree 9 each. For example a degree 3 triangular function is given as $x_1 + x_2x_3 + x_4x_5x_6$ (a degree $n$ function thus has $n(n+1)/2$ variables).

Although the designers stop short of providing detailed design specifications, they do however mention that the permutations are generated by employing a combination of the IV and a PRP (possibly in the counter mode) to generate a sequence of pseudo-random bits, which are then used as random inputs to a Knuth shuffle module which generates the permutation. The question is therefore how to efficiently do a Knuth shuffle in a lightweight setting. In this let us make two observations:

**Observation 1:** If $P_1$ and $P_2$ are random permutations over any symmetric group then $P_1 \circ P_2$ is also a random permutation. This means that we can modify the sequence of operations in FLIP to the following:

- Let $sk \in \{0,1\}^{530}$ with $HW(sk) = 265$.

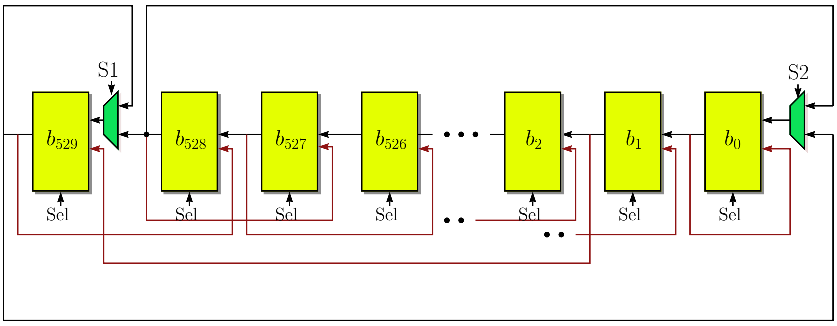


**Figure 11:** 2nd Circuit for Knuth Shuffle

```
• $State_0 = sk$
• for $i = 1 \to n$ do
    1: $P_i \xleftarrow{\$} S_{530}$.
    2: Let $State_i = P_i(State_{i-1})$.
    3: Compute $z_i = F(State_i)$.
```

This makes the $i^{th}$ state $P_i \circ P_{i-1} \circ \cdots \circ P_1(sk)$ in place of $P_i(sk)$ but since $P_i$ and $P_i \circ P_{i-1} \circ \cdots \circ P_1$ are both random permutations, this does not differ from the ideology of FLIP.

**Observation 2:** The above allows us to simply place the secret key in a register of equal length and do state updates by implementing the permutations $P_i$ via some circuit. Since the authors recommend Knuth Shuffle, let us look at the algorithm. Let the state be denoted by the bits $b_{529}, b_{528}, \ldots, b_0$.

```
• for $i$ from 529 downto 1 do
    1: $j \leftarrow$ random integer such that $0 \le j \le i$.
    2: swap $b_j$ and $b_i$
```

Each of these swaps could be implemented with a circuit as shown in Figure 1.

## 7.1 First Attempt

The first idea is therefore to use the circuit in Figure 1 to implement a swap. As explained in Lemma 4, any swap of the form $b_i \leftrightarrow b_j$ $(j < i)$ is implemented by the sequence of functions:

$$[r^i \circ v \circ r^{529-i}] \circ [r^{i-1} \circ v \circ r^{530-i}] \circ \cdots \circ [r^{j+2} \circ v \circ r^{527-j}] \circ [r^{j+1} \circ v^{i-j} \circ r^{529-i}]$$

and by the identity function when $i = j$. This transpositions would take $530(i-j)$ cycles to complete and we could certainly use this circuit to implement one swap. The average value of $i-j$ is around $\frac{530}{4}$ and so implementing 529 swaps one after the other would take around $\frac{530^3}{4} \approx 2^{25}$ cycles which is a high price to pay for one keystream bit.

## 7.2 Second Attempt

We try to investigate if we can affect any speedup by increasing the circuit size. One of the reasons that a swap takes $530(i-j)$ cycles is that data can be transferred in only one direction. This is true because as per Lemma 4, the above sequence of transitions may also be written as

$$\begin{aligned} &r^{1+i} \circ w \circ (r^{529} \circ w)^{i-j-1} \circ (r \circ w)^{i-j-1} \circ r^{529-i} \\ = &(r^{-1})^{530-i} \circ (r^{-1} \circ w')^{i-j-1} \circ (r \circ w)^{i-j} \circ r^{529-i} \end{aligned}$$

In the above equation $w'$ is a permutation that swaps the 1st and 0th bits i.e. $(1,0)$. The above is not difficult to deduce once we use the fact $r^{-1} = r^{529}$ and $r^{-1} \circ w' = r \circ w \circ r^{-2}$. Denote $u = r^{-1} \circ w'$, then the above is written as $(r^{-1})^{530-i} \circ u^{i-j-1} \circ v^{i-j} \circ r^{529-i}$. It can be seen that the circuit of Figure 11, it is possible to realize the functions $r, u, v$ by appropriately adjusting the Sel, S1 and S2 signals. If we assume that in Figure 11, the signal at the top is filtered when the corresponding select signal is 0 and the one at the bottom when select is 1, it is easy to deduce that Sel, S1, S2 = (0,0,0) achieves $v$, Sel, S1, S2 = (0,1,1) achieves $r$ and Sel, S1, S2 = (1,*,*) achieves $u$ (* denotes any signal 0 or 1).

The algorithm for the Knuth shuffle basically consists of applying the following transpositions

$$\pi_K = (1, j_1) \circ (2, j_2) \circ (3, j_3) \circ \cdots \circ (528, j_{528}) \circ (529, j_{529}), \text{ with } (j_i \leq i, \ \forall\ i)$$

Denote $\Delta_i = i - j_i$, we have

$$(i, j_i) = \begin{cases} (r^{-1})^{530-i} \circ u^{\Delta_i - 1} \circ v^{\Delta_i} \circ r^{529-i}, & \text{if } \Delta_i > 0, \\ (r^{-1})^{529-i} \circ r^{529-i}, & \text{if } \Delta_i = 0. \end{cases}$$

This results in the following expression for $\pi_K$:

$$\pi_k = r \circ [u^{\Delta_1 - 1} \circ v^{\Delta_1}] \circ [u^{\Delta_2 - 1} \circ v^{\Delta_2}] \circ \cdots \circ \underset{\text{when } \Delta_i = 0}{[r]} \circ \cdots \circ [u^{\Delta_{529} - 1} \circ v^{\Delta_{529}}]$$

The above expression is solely in terms of $r, u, v$ and thus can be executed on the circuit in figure 11. Unless $\Delta_i = 0$ which requires a single rotation, each of the expressions in the square braces takes $2\Delta_i - 1$ cycles. Since $\Delta_i$ has an average value of $\frac{530}{4}$, each shuffle takes around $530 \cdot (2 \cdot \frac{530}{4} - 1) \approx 2^{17}$ cycles. A synthesis of the above circuit using the standard cell library of the STM 90nm logic process, yielded a circuit of 3581 GE. The circuit is certainly an improvement on the previous circuit but still takes a lot of cycles to produce one keystream bit.

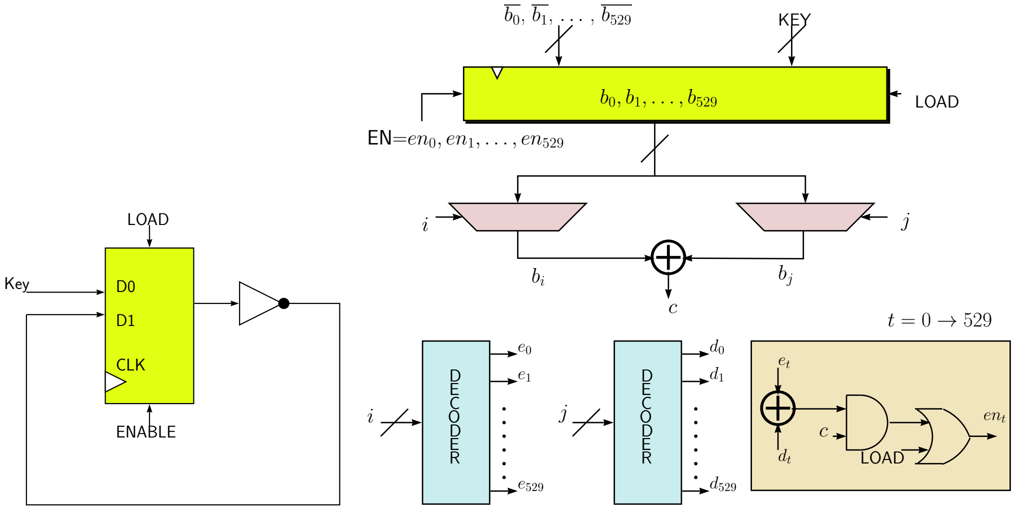
## 7.3 Third Attempt

The previous circuits took time proportional to $N^3$ and $N^2$ clock cycles respectively to produce one keystream bit where $N$ is the size of the key. In this part, we will try to construct a circuit in which the number of clock cycles taken to produce a keystream bit is at most linear in $N$. In order to achieve this, let us look at a few facts:

**1:** In order to achieve a shuffle in linear time, each individual swap has to be executed in constant time.

**2:** Observe that logically, a swap $b_i \leftrightarrow b_j$ needs to be executed only when $b_i$ and $b_j$ are opposite values. No swap operation is really necessary if $b_i$ and $b_j$ are logically equal. Furthermore, when $b_i$ and $b_j$ are logically unequal, a swap is essentially executed by toggling the values of both $b_i$ and $b_j$, i.e. $\textbf{swap}(b_i, b_j)$ is same as $b_i \leftarrow \textbf{not } b_i$ and $b_j \leftarrow \textbf{not } b_j$.

**3:** In order to design this circuit, we note that a memory element must be able to accommodate **a)** the secret key during the initial loading cycle, **b)** hold the current



**(a)** Memory Element

**(b)** Circuit Details

**Figure 12:** 3rd Circuit for FLIP

value stored in the flip-flop for the next cycle, if no swap is required and finally **c)** toggle the current logic state if a swap is required at the particular location. In order to do this we use a scan flip-flop with an additional ENABLE pin that allows transitions at the positive clock edge only if it is HIGH, as shown in Figure 12a.

Thus we propose the circuit in Figure 12b. The circuit comprises of the following elements:

**Multiplexer:** We employ two banks of multiplexers to filter out the bits $b_i$ and $b_j$ from the current state. We compute $c = b_i \oplus b_j$ to determine the difference in the logic values of $b_i$ and $b_j$.

**Decoder:** We employ 2 decoder circuits that convert the 10-bit values $i$ and $j$ into a corresponding set of 530-bit signals $e_t, d_t$ (for $t = 0 \to 529$), such that $e_t = 1$ iff $t = i$, and $d_t = 1$ iff $t = j$, and all signals are 0 otherwise. These signals are employed to feed the the ENABLE ports of the register bank. We argue that the logic value $en_t$ driving the $t - th$ flip-flop is given as $[(e_t \oplus d_t) \cdot c]$ OR LOAD. The logic behind this is as follows. The signal LOAD is high only in the first cycle when in loads the key on to the register and is low thereafter. Thus $en_t$ is forced to be high when LOAD is high. In the subsequent cycles $en_t$ evaluates as $(e_t \oplus d_t) \cdot c$. If $c = 0$, i.e. $b_i$ and $b_j$ are of same parity then $en_t$ evaluates to 0 which means that the flip-flop holds its previous value as no swap is required. If $c = 1$, then $en_t = e_t \oplus d_t$. Now $e_t = d_t = 0$ implies that no swap is scheduled at location $t$ in that particular clock cycle and in this event $en_t$ is 0. Now $e_t = d_t = 1$ occurs when $i = j$ in some iteration of the Knuth Shuffle. Here too, no swap operation is required and $en_t$ evaluates to 0. When $e_t \neq d_t$, $en_t$ evaluates to 1, and it is then that $b_i$ and $b_j$ are both toggled to effect a swap.

From the above description of the circuit elements and operational details, it is clear that each swap can be performed in one clock cycle, and so the shuffle takes exactly $529 + 1 = 530$ (1 extra cycle for key loading) clock cycles to execute a shuffle and hence produce one keystream bit. This circuit when synthesized with standard cell library of the STM 90nm logic process occupies around 8605 GE.

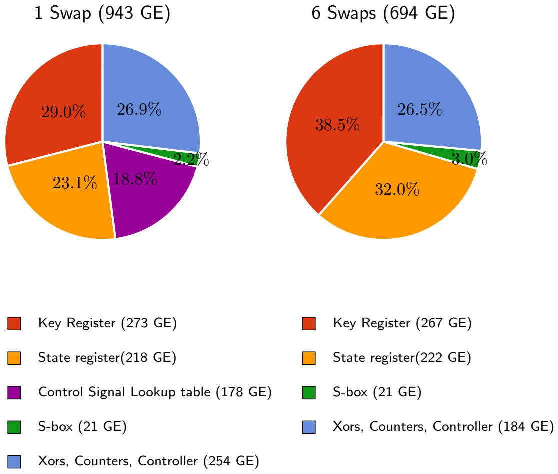


**Figure 13:** Breakdown of the area requirements of the individual components of PRESENT in the 1 and 2 swap architectures

# 8 Results and Conclusion

In this paper we looked at a few circuit constructions aimed at achieving minimalism in block cipher and stream cipher circuits. The final results are presented in Table 1. More specifically, we tried to answer the question if bit-permutations like the one used in the linear layers of block ciphers PRESENT and GIFT can be executed in a flip-flop array using only two scan flip-flops. While it was already known [Con] that the answer to the above question was yes, a straightforward application of the ideas [Con] would take a lot of clock cycles, and thus affect the throughput of the resulting circuit drastically. Much of the paper is then dedicated to reducing the number of operations required to execute the bit permutation in this setting. As an outcome, we construct extremely lightweight implementations of the PRESENT and GIFT circuits for both encryption (E) and combined encryption+decryption (ED) functionalities. In the 2 scan flip-flop setup, the circuits of both PRESENT and GIFT are, for both the (E) and (ED) variants, way too large and have poor throughput.

## 8.1 Increasing scan flip-flops

We tried to see the effect on latency if we added more and more scan flip-flops to the design and finally achieved 64 cycle per round implementations of both PRESENT and GIFT circuits at 694 and 907 GE respectively. These are lowest reported in literature so far. Figure 13 shows a breakdown of the area requirements of the individual components of PRESENT in the 1 and 6 swap architectures. It can be clearly seen the additional control logic used in the 1 swap circuit proves counterproductive in terms of circuit area. One of the reasons that the 6 swap circuit taking 64 cycles/round consumes much less hardware area is the reduced control circuit. A 64 cycle per round finite state machine would require only a 6 bit register to implement and the associated logic blocks used to produce control signals are also much smaller. This area, of course, increases as the number of cycles per round increases. In the (ED) architecture, we have implementations of both PRESENT and GIFT occupying 786 and 1055 GE respectively. These are also the lowest yet reported in literature.

We extend the above ideas to construct a circuit for the stream cipher FLIP. The first circuit we investigate is due to a straightforward application of the results [Con], but takes around $2^{25}$ cycles to produce one keystream bit. This is deemed too impractical to be of any use. The second circuit we construct takes time quadratic in the size of the secret key to produce a keystream bit and occupies only 3581 GE. We then observe that a third circuit that uses slightly different ideas for bit swapping can achieve the FLIP functionality in linear time but occupies around 8605 GE. These are the first reported hardware implementations of FLIP.

## 8.2 Final words

Although one of the goals of this paper is to achieve the smallest area implementation of block ciphers with bit permutations as linear layers, we have also tried to investigate the theory to implement bit permutations in a serialized manner. In this exercise we learnt many interesting things:

- Although implementing block ciphers with only 2 scan flip-flops is a challenging and interesting task, it proves counterproductive, because the control circuit required to operate such a design would be large and thus negate any minimalism achieved due to less number of scan flip-flops in the design.
- On the other hand, increasing the number of scan flip-flops gradually, not only reduces the circuit latency, but it also reduces the total area of the the circuit. The principal reason for this is that with reduced latency, the size of control circuit required to operate the design can be constructed in a much more compact manner. In fact the best implementation of both PRESENT and GIFT circuits, are those which allow slight increase in number of scan flip-flops to bring per round latency to 64 cycles. This reduces the size of the control circuit more than the corresponding increase due to increase in number of scan flip-flops.
- We hope that our findings can help derive design strategies for future cryptosystems.

# Acknowledgments

We thank Christina Boura for helping to improve the quality of the paper. Subhadeep Banik and Fatih Balli are supported by the Ambizione Grant no. PZ00P2_179921, awarded by the Swiss National Science Foundation (SNSF). Francesco Regazzoni received support from the European Union Horizon 2020 research and innovation program under CERBERO project (grant agreement number 732105).

# References

[BBR15] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Exploring energy efficiency of lightweight block ciphers. In *Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers*, pages 178–194, 2015.

[BBR16] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core. In *Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 173–190, 2016.

[BBR17a] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Compact circuits for combined AES encryption/decryption. *Journal of Cryptographic Engineering*, pages 1–15, 2017.

[BBR17b] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Efficient configurations for block ciphers with unified ENC/DEC paths. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, pages 41–46, 2017.

[BJK$^+$16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 123–153, 2016.

[BKL$^+$07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BPP$^+$17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 321–345, 2017.

[BSS$^+$] Ray Beaulieu, Douglas Shors, Jason Smith, Treatman-Clark Stefan, Bryan Weeks, and Louis Wingers. Simon and Speck: Block Ciphers for the Internet of Things. Available at https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf.

[CDK09] Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 272–288, 2009.

[CLM16] Victor Cauchois, Pierre Loidreau, and Nabil Merkiche. Direct construction of quasi-involutory recursive-like MDS matrices from 2-cyclic codes. *IACR Trans. Symmetric Cryptol.*, 2016(2):80–98, 2016.

[Con] Keith Conrad. Generating Sets. Available at http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/genset.pdf.

[CP08] Christophe De Cannière and Bart Preneel. Trivium. In Matthew J. B. Robshaw and Olivier Billet, editors, *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 244–266. Springer, 2008.

[DL18] Sébastien Duval and Gaëtan Leurent. MDS matrices with lightweight circuits. *IACR Trans. Symmetric Cryptol.*, 2018(2):48–78, 2018.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer Verlag, Berlin, Heidelberg, New York, 2002.

[HJMM08] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. *The Grain Family of Stream Ciphers*, pages 179–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[JMPS17] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 687–707, 2017.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. Shorter linear straight-line programs for MDS matrices. *IACR Trans. Symmetric Cryptol.*, 2017(4):188–211, 2017.

[LSL$^+$19] Shun Li, Siwei Sun, Chaoyun Li, Zihao Wei, and Lei Hu. Constructing low-latency involutory MDS matrices with lightweight circuits. *IACR Trans. Symmetric Cryptol.*, 2019(1):84–117, 2019.

[LW17] Chaoyun Li and Qingju Wang. Design of lightweight linear diffusion layers from near-mds matrices. *IACR Trans. Symmetric Cryptol.*, 2017(1):129–155, 2017.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, pages 311–343, 2016.

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[nis19] Nist lightweight cryptography project. https://csrc.nist.gov/projects/lightweight-cryptography, 2019.

[RPLP08] Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents. In *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, pages 89–103, 2008.

[SS16] Sumanta Sarkar and Habeeb Syed. Lightweight diffusion layer: Importance of Toeplitz matrices. *IACR Trans. Symmetric Cryptol.*, 2016(1):95–113, 2016.

# Appendices

## A Proof of Lemma 3

*Proof.* We start with **A** as it is not difficult to prove. Note that $c_j$'s are themselves disjoint decompositions of $\pi$. Thus it is easy to verify that any $s_{j_1}(a)$ and $s_{j_2}(b)$ will be disjoint for any $j_1 \neq j_2$ and any $a, b$. In particular, they are of course disjoint when $a = b$. This proves that all transpositions in any given $\chi_k$ are disjoint. Since disjoint cycles commute, composing the elements of $\chi_k$ in any order, gives the same permutation. This proves that $\theta_k$ is invariant with respect to ordering.

Denote by $\mu_j[x \to y] = s_j(x) \circ s_j(x-1) \circ \cdots \circ s_j(y)$ (for $x \geq y$). Naturally we have $\mu_j[i_j - 1 \to 0] = c_j$. Although $c_j$ is a cycle of order $i_j$, for the completeness of the proof, let us define $s_j(i_j), s_j(i_j+1), \ldots, s_j(i_{m-1}-1)$ to be the identity permutation with $\mathbb{A}_{s_j(i_j)}, \mathbb{A}_{s_j(i_j+1)}, \ldots, \mathbb{A}_{s_j(i_{m-1}-1)}$ equal to $\varnothing$. With this definition we also have $\mu_j[i_{m-1}-1 \to 0] = c_j$. Now to prove **B**, consider the following composition $\theta_2 \circ \theta_1$.

$$\begin{aligned}
\theta_2 \circ \theta_1 &= s_{m-1}(2) \circ s_{m-2}(2) \circ \cdots \circ s_0(2) \circ s_{m-1}(1) \circ s_{m-2}(1) \circ \cdots \circ s_0(1) && (1)\\
&= s_{m-2}(2) \circ \cdots \circ s_0(2) \circ (s_{m-1}(2) \circ s_{m-1}(1)) \circ s_{m-2}(1) \circ \cdots \circ s_0(1) && (2)\\
&= s_{m-2}(2) \circ \cdots \circ s_0(2) \circ \mu_{m-1}[2 \to 1] \circ s_{m-2}(1) \circ \cdots \circ s_0(1) && (3)\\
&= \mu_{m-1}[2 \to 1] \circ s_{m-2}(2) \circ \cdots \circ s_0(2) \circ s_{m-2}(1) \circ \cdots \circ s_0(1) && (4)\\
&= \mu_{m-1}[2 \to 1] \circ \mu_{m-2}[2 \to 1] \circ \cdots \circ \mu_0[2 \to 1] && (5)
\end{aligned}$$

(1) → (2) is true because all $\theta_k$'s are invariant to internal ordering of transpositions as proven in **A**. (2) → (3) follows from the definition of $\mu_j[x \to y]$. To prove (3) → (4), we start with the fact that $s_{j_1}(a)$ and $s_{j_2}(b)$ are disjoint for any $j_1 \neq j_2$ and any $a, b$, which is to say

$$\mathbb{A}_{s_{j_1}(a)} \cap \mathbb{A}_{s_{j_2}(b)} = \varnothing, \ \forall j_1 \neq j_2, \forall\ a, b$$

Therefore, we have, for all $j \in [0, m-2]$, the following relation:

$$\begin{aligned}
\mathbb{A}_{\mu_{m-1}[2\to1]} \cap \mathbb{A}_{s_j(2)} &= (\mathbb{A}_{s_{m-1}(2)} \cup \mathbb{A}_{s_{m-1}(1)}) \cap \mathbb{A}_{s_j(2)}\\
&= (\mathbb{A}_{s_{m-1}(2)} \cap \mathbb{A}_{s_j(2)}) \cup (\mathbb{A}_{s_{m-1}(1)} \cap \mathbb{A}_{s_j(2)})\\
&= \varnothing \cup \varnothing = \varnothing
\end{aligned}$$

This proves that $\mu_{m-1}[2 \to 1]$ is disjoint with all of $s_{m-2}(2), s_{m-3}(2) \ldots, s_0(2)$ and so (3) → (4) follows. (4) → (5) is just a generalization of steps $(2), (3), (4)$ for the indices $m-2, m-3, \ldots, 0$. Proceeding as in mathematical induction, we can follow exactly the steps above to prove that $\theta_3 \circ \theta_2 \circ \theta_1 = \mu_{m-1}[3 \to 1] \circ \mu_{m-2}[3 \to 1] \circ \cdots \circ \mu_0[3 \to 1]$ and ultimately the fact that

$$\begin{aligned}
\theta_{i_{m-1}-1} \circ \theta_{i_{m-2}-1} \circ \cdots \circ \theta_1 &= \mu_{m-1}[i_{m-1}-1 \to 1] \circ \mu_{m-2}[i_{m-1}-1 \to 1] \circ \cdots \circ \mu_0[i_{m-1}-1 \to 1]\\
&= c_{m-1} \circ c_{m-2} \circ \cdots \circ c_0 = \pi
\end{aligned}$$

□

## B Proof of Lemma 5

*Proof.* The only thing we need to show is that any transposition $(x, y)$ with $x > y$ and $x \equiv y \bmod \kappa$, can be generated using $w_\kappa$ and $r$. Let $z = \frac{x-y}{\kappa}$. We have

$$\begin{aligned}
(x, y) &= (x, x-\kappa) \circ (x-\kappa, y) \circ (x, x-\kappa)\\
&= (x, x-\kappa) \circ (x-\kappa, x-2\kappa) \circ (x-2\kappa, y) \circ (x-\kappa, x-2\kappa) \circ (x, x-\kappa)\\
&= (x, x-\kappa) \circ (x-\kappa, x-2\kappa) \circ \cdots \circ (y+\kappa, y) \circ \cdots \circ (x-\kappa, x-2\kappa) \circ (x, x-\kappa)\\
&= (r^{-\overline{x}} \circ w_\kappa \circ r^{\overline{x}}) \circ (r^{-\kappa-\overline{x}} \circ w_\kappa \circ r^{\kappa+\overline{x}}) \circ \cdots \circ (r^{\kappa-\overline{y}} \circ w_\kappa \circ r^{\overline{y}-\kappa}) \circ \cdots \circ\\
&\quad (r^{-\kappa-\overline{x}} \circ w_\kappa \circ r^{\kappa+\overline{x}}) \circ (r^{-\overline{x}} \circ w_\kappa \circ r^{\overline{x}})\\
&= r^{-\overline{x}} \circ w_\kappa \circ (r^{-\kappa} \circ w_\kappa)^{z-1} \circ (r^{\kappa} \circ w_\kappa)^{z-1} \circ r^{\overline{x}}\\
&= r^{64-\overline{x}} \circ w_\kappa \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ (r^{\kappa} \circ w_\kappa)^{z-1} \circ r^{\overline{x}}\\
&= r^{1+x} \circ w_\kappa \circ (r^{64-\kappa} \circ w_\kappa)^{z-1} \circ (r^{\kappa} \circ w_\kappa)^{z-1} \circ r^{63-x}
\end{aligned}$$

□

## C Proof of Lemma 7

*Proof.* To begin with we have $p_1$ and $p_2$ disjoint, as $\mathbb{A}_{p_1} \cap \mathbb{A}_{p_2} = \varnothing$. Note that this implies $\mathbb{B}_{p_1} \cap \mathbb{B}_{p_2} = \varnothing$ (although the converse may not always be true). This means that the 1's in the $\overrightarrow{\mathsf{Sel}}_{p_1}$ and $\overrightarrow{\mathsf{Sel}}_{p_2}$ vectors are not aligned. Which is to say $\overrightarrow{\mathsf{Sel}}_{p_1} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{p_2}$ has 1's in all the locations in which either $\overrightarrow{\mathsf{Sel}}_{p_1}$ or $\overrightarrow{\mathsf{Sel}}_{p_2}$ has 1. Let $\overrightarrow{\mathsf{Sel}}_{p} = \overrightarrow{\mathsf{Sel}}_{p_1} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{p_2}$. We already know that $\mathbb{B}_p$ would contain all elements of $\mathbb{B}_{p_1}$ and $\mathbb{B}_{p_2}$. Thus the arithmetic sequence structures of both $\mathbb{B}_{p_1}$ and $\mathbb{B}_{p_2}$ are preserved in $\mathbb{B}_p$. Furthermore, $\mathbb{A}_{p_1} \cap \mathbb{A}_{p_2} = \varnothing$ ensures that no new arithmetic sequence of common difference $\kappa$ is created $\mathbb{B}_p$ that are already not present in $\mathbb{B}_{p_1}$ or $\mathbb{B}_{p_2}$. We will prove this by contradiction: if possible let $\exists b_1 \in \mathbb{B}_{p_1},\ b_2 \in \mathbb{B}_{p_2}$ such that $b_2 = b_1 + \kappa$. Then by definition $63 - b_1, 63 - b_1 - \kappa \in \mathbb{A}_{p_1}$ and $63 - b_2, 63 - b_2 - \kappa \in \mathbb{A}_{p_2}$. But $63 - b_1 - \kappa = 63 - b_2$, and so this contradicts the fact that $\mathbb{A}_{p_1} \cap \mathbb{A}_{p_2} = \varnothing$. Since the arithmetic structures are preserved, $p$ essentially executes $p_1$ and $p_2$ concurrently: we have $\forall \alpha \in \mathbb{A}_{p_1}$, $p(\alpha) = p_1(\alpha)$ and $\forall \alpha \in \mathbb{A}_{p_2}$, $p(\alpha) = p_2(\alpha)$. Also $p(\alpha) = \alpha$ for all $\alpha \notin \mathbb{A}_{p_1} \cup \mathbb{A}_{p_2}$. Thus we have $p = p_1 \circ p_2$. □

## D Proof of Lemma 8

*Proof.* Since $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} = \varnothing$, from the result of the previous lemma, we can certainly use $\overrightarrow{\mathsf{Sel}}_{\pi_0} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{\theta_0}$ to get $\pi_0 \circ \theta_0$. Since all $\mathbb{A}_{\pi_i}$'s and $\mathbb{A}_{\theta_i}$'s are subsets of $\mathbb{A}_{\pi_0}$ and $\mathbb{A}_{\theta_0}$ respectively, we also have $\mathbb{A}_{\pi_i} \cap \mathbb{A}_{\theta_i} = \varnothing$ for all $0 \le i \le z_1 - 1$. We can then use $\overrightarrow{\mathsf{Sel}}_{\pi_i} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{\theta_i}$ to get $\pi_i \circ \theta_i$ for all $0 \le i \le z_1 - 1$. Thus if $\overrightarrow{\mathsf{Sel}}_{p} = \overrightarrow{\mathsf{Sel}}_{\sigma_1} \hat{|}\, \overrightarrow{\mathsf{Sel}}_{\sigma_2}$, we naturally have

$$p = (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\pi_1 \circ \theta_1) \circ (\pi_0 \circ \theta_0)$$

Denote by $\pi[i \to j] = \pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_0$ and $\theta[i \to j] = \theta_i \circ \theta_{i-1} \circ \cdots \circ \theta_0$. Note that $\mathbb{A}_{\pi[i_1 \to j_1]} \cap \mathbb{A}_{\theta[i_2 \to j_2]} = \varnothing$, since the parent sets $\mathbb{A}_{\pi_0}$ and $\mathbb{A}_{\theta_0}$ are themselves disjoint. So we have

$$\begin{aligned}
p &= (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\pi_1 \circ \theta_1 \circ \pi_0 \circ \theta_0) && (6)\\
&= (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\theta_1 \circ \pi_1 \circ \pi_0 \circ \theta_0) && (7)\\
&= (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\theta_1 \circ \pi[1 \to 0] \circ \theta_0) && (8)\\
&= (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\theta_1 \circ \theta_0 \circ \pi[1 \to 0]) && (9)\\
&= (\pi_{z_1-1} \circ \theta_{z_1-1}) \circ (\pi_{z_1-2} \circ \theta_{z_1-2}) \circ \cdots \circ (\theta[1 \to 0] \circ \pi[1 \to 0]) && (10)
\end{aligned}$$

$(6 \to 7)$ follows because $\mathbb{A}_{\pi_1} \cap \mathbb{A}_{\theta_1} = \varnothing$. $(8 \to 9)$ follows because $\mathbb{A}_{\pi[1 \to 0]} \cap \mathbb{A}_{\theta_0} = \varnothing$. The remaining statements follow from definition. The steps in the above equations can be repeated for $i = 2$ to $z_1 - 1$ to get $p = \pi[z_1 - 1 \to 0] \circ \theta[z_1 - 1 \to 0] = \sigma_1 \circ \sigma_2$. □

## E Proof of Lemma 9

*Proof.* First of all, let us clarify what we are trying to do. We want to implement

$$\begin{aligned}
\sigma_1 \circ \sigma_2 &= \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_2 \circ \pi_1 \circ \pi_0 \circ (x_2, y_2)\\
&= \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_{i+1} \circ \pi[i \to 0] \circ (x_2, y_2)\\
&= \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_{i+1} \circ (\pi[i \to 0](x_2), \pi[i \to 0](y_2)) \circ \pi[i \to 0]\\
&= \pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_{i+1} \circ p \circ \pi[i \to 0]
\end{aligned}$$

We are therefore trying to implement $\pi_{z_1-1} \circ \pi_{z_1-2} \circ \cdots \circ \pi_{i+1} = \pi[z_1 - 1 \to i + 1]$ and $p$ concurrently after implementing $\pi[i \to 0]$. Now $\mathbb{B}_{\pi_{i+1}}, \mathbb{B}_{\pi_{i+2}}, \ldots$ are singleton sets and so are $\mathbb{B}_{\gamma_1}, \mathbb{B}_{\gamma_2}, \ldots$. If $\mathbb{B}_{\gamma_0} = \{g_1, g_1 + \kappa, g_1 + 2\kappa, \ldots, h_1\}$ and $\mathbb{B}_{\pi_{i+1}} = \{g_2\}$ are disjoint (note we have taken $h_1 > g_1$), then we have

$$\begin{gathered}\mathbb{B}_{\gamma_1} = \{h_1 - \kappa\},\ \ \mathbb{B}_{\pi_{i+2}} = \{g_2 - \kappa\} \\ \mathbb{B}_{\gamma_2} = \{h_1 - 2\kappa\},\ \ \mathbb{B}_{\pi_{i+3}} = \{g_2 - 2\kappa\} \\ \vdots\end{gathered}$$

Thus $\mathbb{B}_{\gamma_j}$ and $\mathbb{B}_{\pi_{i+j+1}}$ are not only disjoint (for $j \geq 1$), but the distance between the single elements in the sets equals $g_2 - h_1$ which is a non-zero constant. Note that we have $g_1 \equiv h_1 \equiv g_2 \bmod \kappa$, since $\mathbb{A}_{\pi_0} \cap \mathbb{A}_{\theta_0} \neq \varnothing$. Since $\mathbb{B}_{\gamma_0} \cap \mathbb{B}_{\pi_{i+1}} = \varnothing$, we must have either $g_2 \geq h_1 + \kappa$ or $g_2 < g_1 - \kappa$. $g_2 = g_1 - \kappa$ is not possible as it leads to a contradiction: if $g_2 = g_1 - \kappa$, then the largest element in $\mathbb{B}_{\pi_0}$ is $g_1 + i\kappa$, and so $\sigma_1 = (u, 63 - g_1 - (i+1)\kappa)$, for some $u$. We have $p = (63 - h_1, 63 - g_1 - \kappa) = (63 - h_1, \pi[i \to 0](63 - g_1 - (i+1)\kappa))$, which means $\sigma_2 = (v, 63 - g_1 - (i+1)\kappa)$, for some $v$. This contradicts the fact that $\sigma_1$ and $\sigma_2$ are disjoint. Denote $z_3 = \pi[i \to 0](x_2) - \pi[i \to 0](y_2))$. We have

$$\begin{aligned}\mathbb{A}_{\gamma_0} = \{63 - g_1, 63 - g_1 - \kappa, \ldots, 63 - h_1, 63 - h_1 - \kappa\},\ \ & \mathbb{A}_{\pi_{i+1}} = \{63 - g_2, 63 - g_2 - \kappa\} \\ \mathbb{A}_{\gamma_1} = \{63 - h_1 + \kappa, 63 - h_1\},\ \ & \mathbb{A}_{\pi_{i+2}} = \{63 - g_2 + \kappa, 63 - g_2\} \\ \mathbb{A}_{\gamma_2} = \{63 - h_1 + 2\kappa, 63 - h_1 + \kappa\},\ \ & \mathbb{A}_{\pi_{i+3}} = \{63 - g_2 + 2\kappa, 63 - g_2 + \kappa\} \\ & \vdots \\ \mathbb{A}_{\gamma_{q-1}} = \{63 - g_1, 63 - g_1 - \kappa\}. &\end{aligned}$$

Thus $\mathbb{A}_{\gamma_j}$ and $\mathbb{A}_{\pi_{i+j+1}}$ are non-disjoint (for $j \geq 0$) only if $g_2 = h_1 + \kappa$ or $g_2 = g_1 - \kappa$. Also note that

$$\mathbb{A}_{\pi[i+j+1 \to i+1]} = \{63 - g_2 - \kappa, 63 - g_2, 63 - g_2 + \kappa, \ldots, 63 - g_2 + j\kappa\}$$

We want to find $\mathbb{A}_{\pi[i+j+1 \to i+1]} \cap \mathbb{A}_{\gamma_j}$. The numerical maximum of $\mathbb{A}_{\pi[i+j+1 \to i+1]}$ is $63 - g_2 + j\kappa$ and numerical minimum of $\mathbb{A}_{\gamma_j}$ is $63 - h_1 + (j-1)\kappa$. The min - max difference comes out to be $g_2 - h_1 - \kappa$. If $g_2 > h_1 + \kappa$, this is always greater than 0 and so the sets are disjoint. If $g_2 < g_1 - \kappa$, then the minimal element of $\mathbb{A}_{\pi[i+j+1 \to i+1]}$, i.e. $63 - g_2 - \kappa > 63 - g_1$ which is the maximal element in the $\mathbb{A}_{\gamma_j}$'s. Here too the sets are disjoint. So we have three cases to analyze (A) $g_2 > h_1 + \kappa$ or $g_2 < g_1 - \kappa$, (B) $g_2 = h_1 + \kappa$. So let us split the analysis into two cases:

**A: $\boldsymbol{g_2 > h_1 + \kappa}$ or $\boldsymbol{g_2 < g_1 - \kappa}$:** We have $\mathbb{A}_{\gamma_j} \cap \mathbb{A}_{\pi_{i+j+1}} = \mathbb{B}_{\gamma_j} \cap \mathbb{B}_{\pi_{i+j+1}} = \varnothing$ for all $j \geq 0$. We also have $\mathbb{A}_{\pi[i+j+1 \to i+1]} \cap \mathbb{A}_{\gamma_j} = \varnothing$. This means that $\overrightarrow{\mathsf{Sel}}_1 \hat{\lceil} \overrightarrow{\mathsf{Sel}}_2$ has 1's in locations where either $\overrightarrow{\mathsf{Sel}}_1$ or $\overrightarrow{\mathsf{Sel}}_2$ is 1. Let $\mathsf{z}$ be the final length of $\overrightarrow{\mathsf{Sel}}_1, \overrightarrow{\mathsf{Sel}}_2$ after padding. By Lemma 7, if $\overrightarrow{\mathsf{Sel}}_\Pi = \overrightarrow{\mathsf{Sel}}_1 \hat{\lceil} \overrightarrow{\mathsf{Sel}}_2$, then

$$\begin{aligned}\Pi &= (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\pi_{i+2} \circ \gamma_1) \circ (\pi_{i+1} \circ \gamma_0) && (11) \\ &= (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\gamma_1 \circ \pi_{i+2}) \circ (\pi_{i+1} \circ \gamma_0) && (12) \\ &= (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\gamma_1 \circ \pi[i+2 \to i+1] \circ \gamma_0) && (13) \\ &= (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\pi[i+2 \to i+1] \circ \gamma_1 \circ \gamma_0) && (14) \\ &= (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\pi[i+2 \to i+1] \circ \gamma[1 \to 0]) && (15) \\ &= \pi[\mathsf{z}+i \to i+1] \circ \gamma[\mathsf{z}-1 \to 0] = \pi[\mathsf{z}+i \to i+1] \circ p && (16)\end{aligned}$$

(11 → 12) follows because $\mathbb{A}_{\gamma_j} \cap \mathbb{A}_{\pi_{i+j+1}} = \varnothing$ for all $j$. (13 → 14) follows because $\mathbb{A}_{\pi[i+j+1 \to i+1]} \cap \mathbb{A}_{\gamma_j} = \varnothing$ for all $j$. (15 → 16) follows after repeating (11 → 15) for $j = 0, 1, 2 \ldots$ etc. The remaining statements follow by definition.

**B: $g_2 = h_1 + \kappa$:** Before we analyze this case, let us restate a result in permutation theory

$$(x_{n_1}, x_{n_2}, \ldots, x_{n_l}) \circ (x_{n_l}, x_{n_{l+1}}, \ldots, x_{n_k}) = (x_{n_1}, x_{n_2}, \ldots, x_{n_l}, \ldots, x_{n_k}). \tag{17}$$

Since $g_2 = h_1 + \kappa$, the following is easy to verify (denote $\overline{h}_1 = 63 - h_1, \overline{g}_1 = 63 - g_1$)

$$\begin{aligned} \pi_{i+j+1} = &\quad (\overline{h}_1 + (j-2)\kappa,\ \overline{h}_1 + (j-1)\kappa),\ \text{and} \\ \gamma_j = &\begin{cases} (\overline{h}_1 - \kappa,\ \overline{h}_1,\ \overline{h}_1 + \kappa,\ \ldots,\ \overline{g}_1), & \text{if } j = 0, \\ (\overline{h}_1 + (j-1)\kappa,\ \overline{h}_1 + j\kappa), & \text{otherwise.} \end{cases} \end{aligned}$$

By directly applying equation (17), we can obtain the following

$$\pi_{i+j+1} \circ \gamma_j = \begin{cases} (\overline{h}_1 - 2\kappa,\ \overline{h}_1 - \kappa,\ \overline{h}_1,\ \overline{h}_1 + \kappa,\ \ldots,\ \overline{g}_1), & \text{if } j = 0, \\ (\overline{h}_1 + (j-2)\kappa,\ \overline{h}_1 + (j-1)\kappa,\ \overline{h}_1 + j\kappa), & \text{otherwise.} \end{cases} \tag{18}$$

$$\pi[i+j+1 \to i+1] = (\overline{h}_1 + (j-1)\kappa,\ \ldots,\ \overline{h}_1,\ \overline{h}_1 - \kappa,\ \overline{h}_1 - 2\kappa)$$

By induction it is easy to deduce that

$$\gamma[j \to 0] = (\overline{h}_1 - \kappa, \quad \overline{h}_1 + j\kappa,\ \overline{h}_1 + (j+1)\kappa, \ldots,\ \overline{g}_1)$$

From the above two equations we can deduce that

$$\pi[i+j+1 \to i+1] \circ \gamma[j \to 0] = (\overline{h}_1 - 2\kappa,\ \overline{h}_1 + (j-1)\kappa, \ldots, \overline{h}_1, \overline{h}_1 - \kappa, \overline{h}_1 + j\kappa, \ldots, \overline{g}_1) \tag{19}$$

Note that if we denote $\mathbb{B}_{q_j} = \mathbb{B}_{\gamma_j} \cup \mathbb{B}_{\pi_{i+j+1}}$, then $\mathbb{B}_{q_0} = \{g_1, g_1 + \kappa, \ldots, h_1, h_1 + \kappa\}$ and $\mathbb{B}_{q_j} = \{h_1 - (j-1)\kappa,\ h_1 + j\kappa\}$ for $j > 0$. From this it is easy to deduce that $q_j = \pi_{i+j+1} \circ \gamma_j$ for all $j$, (only that this time $\pi_{i+j+1}$ and $\gamma_j$ do not commute). Thus as per the analysis of case (A) we again have

$$\Pi = (\pi_{\mathsf{z}+i} \circ \gamma_{\mathsf{z}-1}) \circ (\pi_{\mathsf{z}+i-1} \circ \gamma_{\mathsf{z}-2}) \circ \cdots \circ (\pi_{i+2} \circ \gamma_1) \circ (\pi_{i+1} \circ \gamma_0)$$

where $\Pi$ is such that $\overrightarrow{\mathsf{Sel}}_{\Pi} = \overrightarrow{\mathsf{Sel}}_1 \hat{\upharpoonleft} \overrightarrow{\mathsf{Sel}}_2$. In spite of the fact that $\pi_{i+j+1}$ and $\gamma_j$ do not commute, we intend to prove that

$$(\pi_{i+j+1} \circ \gamma_j) \circ \cdots \circ (\pi_{i+2} \circ \gamma_1) \circ (\pi_{i+1} \circ \gamma_0) = \pi[i+j+1 \to i+1] \circ \gamma[j \to 0],\ \forall\, j$$

which would prove equation (16) for this case too. We proceed by mathematical induction: for $j = 1$, from equation (19), we have

$$\pi[i+2 \to i+1] \circ \gamma[1 \to 0] = (\overline{h}_1 - 2\kappa,\ \overline{h}_1,\ \overline{h}_1 - \kappa, \overline{h}_1 + \kappa, \ldots, \overline{g}_1)$$

Also $(\pi_{i+2} \circ \gamma_1) \circ (\pi_{i+1} \circ \gamma_0)$ can be calculated from equation (18) as:

$$\begin{aligned} &(\overline{h}_1 - \kappa,\ \overline{h}_1,\ \overline{h}_1 + \kappa) \circ (\overline{h}_1 - 2\kappa,\ \overline{h}_1 - \kappa,\ \overline{h}_1,\ \overline{h}_1 + \kappa,\ \ldots,\ \overline{g}_1) \\ &= (\overline{h}_1 - 2\kappa,\ \overline{h}_1,\ \overline{h}_1 - \kappa, \overline{h}_1 + \kappa, \ldots, \overline{g}_1) = \pi[i+2 \to i+1] \circ \gamma[1 \to 0]. \end{aligned}$$

We will now prove instance $j+1$ assuming all instances from $1 \to j$ are correct. From equations (18), (19) we can calculate $(\pi_{i+j+2} \circ \gamma_{j+1}) \circ \pi[i+j+1 \to i+1] \circ \gamma[j \to 0]$ as follows:

$$\begin{aligned} &(\overline{h}_1 + (j-1)\kappa,\ \overline{h}_1 + j\kappa,\ \overline{h}_1 + (j+1)\kappa)\ \circ\ (\overline{h}_1 - 2\kappa,\ \overline{h}_1 + (j-1)\kappa, \ldots, \overline{h}_1, \\ &\overline{h}_1 - \kappa, \overline{h}_1 + j\kappa, \ldots, \overline{g}_1) \\ &= (\overline{h}_1 - 2\kappa, \overline{h}_1 + j\kappa,\ \overline{h}_1 + (j-1)\kappa, \ldots, \overline{h}_1, \overline{h}_1 - \kappa, \overline{h}_1 + (j+1)\kappa, \ldots, \overline{g}_1) \\ &= \pi[i+j+2 \to i+1] \circ \gamma[j+1 \to 0]. \end{aligned}$$

This concludes proof for case (B).

□

# F PRESENT and GIFT control tables

**Table 12:** Constructed $\mathbb{B}$ sets for the $t_i$'s and $s_i$'s in the PRESENT permutation

| Group | mod3 | $j$ | $\mathbb{B}_{\eta_j}$ | #Cycles |
|---|---|---|---|---|
| $s_i$ | 1 | 0 | $\{29, 32, 35, 38, 41, 44, 47, 50\}$ | 512 |
| | | 1 | $\{8, 11, 14, 17, 20, 23, 26, 47\}$ | |
| | | 2 | $\{5, 8, 11, 23, 44\}$ | |
| | | 3 | $\{8, 20, 26, 29, 32, 41\}$ | |
| | | 4 | $\{5, 17, 29, 38, 47, 50, 53, 56\}$ | |
| | | 5 | $\{14, 26, 35, 44, 53\}$ | |
| | | 6 | $\{11, 32, 50\}$ | |
| | | 7 | $\{8, 29, 47\}$ | |
| | 2 | 0 | $\{31, 34, 37, 40, 43, 46, 49, 52\}$ | 512 |
| | | 1 | $\{10, 13, 16, 19, 22, 25, 28, 49\}$ | |
| | | 2 | $\{25, 28, 31, 34, 37, 40, 43, 46\}$ | |
| | | 3 | $\{4, 7, 10, 22, 40, 43\}$ | |
| | | 4 | $\{7, 19, 25, 28, 37, 40, 46, 49, 52, 55\}$ | |
| | | 5 | $\{4, 10, 13, 16, 25, 34, 37, 52\}$ | |
| | | 6 | $\{10, 13, 31, 34, 49\}$ | |
| | | 7 | $\{10, 28, 31, 46\}$ | |
| $t_i$ | 0 | 0 | $\{18, 21, 24, 27, 30, 33, 39, 42, 45, 48, 51, 54\}$ | 576 |
| | | 1 | $\{6, 9, 12, 15, 18, 30, 51\}$ | |
| | | 2 | $\{15, 27, 48\}$ | |
| | | 3 | $\{12, 24, 45\}$ | |
| | | 4 | $\{9, 21, 27, 30, 33, 36, 39, 42\}$ | |
| | | 5 | $\{6, 18, 36, 39\}$ | |
| | | 6 | $\{3, 6, 9, 33, 51, 54, 57\}$ | |
| | | 7 | $\{6, 30, 54\}$ | |
| | | 8 | $\{3, 27, 51\}$ | |
| | 1 | 0 | $\{11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41\}$ | 704 |
| | | 1 | $\{38\}$ | |
| | | 2 | $\{35\}$ | |
| | | 3 | $\{32\}$ | |
| | | 4 | $\{29, 35, 38, 41, 44, 47, 50, 53\}$ | |
| | | 5 | $\{2, 5, 20, 23, 26, 50\}$ | |
| | | 6 | $\{2, 14, 20, 23, 35, 59\}$ | |
| | | 7 | $\{20, 44\}$ | |
| | | 8 | $\{17, 41\}$ | |
| | | 9 | $\{14, 38\}$ | |
| | | 10 | $\{11, 35\}$ | |
| | 2 | 0 | $\{19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49\}$ | 704 |
| | | 1 | $\{7, 10, 13, 16, 19, 22, 46\}$ | |
| | | 2 | $\{19, 31, 34, 43, 55, 58\}$ | |
| | | 3 | $\{1, 16, 31, 40, 55\}$ | |
| | | 4 | $\{13, 19, 37, 43\}$ | |
| | | 5 | $\{10, 34\}$ | |
| | | 6 | $\{7, 31\}$ | |
| | | 7 | $\{28\}$ | |
| | | 8 | $\{25\}$ | |
| | | 9 | $\{22\}$ | |
| | | 10 | $\{19\}$ | |

**Table 13:** Constructed $\mathbb{B}$ sets for the $u_i$'s, $t_i$'s and $s_i$'s in the GIFT permutation

| Group | mod4 | $j$ | $\mathbb{B}_{\eta_j}$ | mod4 | $j$ | $\mathbb{B}_{\eta_j}$ |
|---|---|---|---|---|---|---|
| $u_i$ | 0 | 0 | $\{19, 23, 27, 31, 35\}$ | 1 | 0 | $\{26, 30, 34, 38, 42, 46, 50\}$ |
| | | 1 | $\{31, 47, 51, 55\}$ | | 1 | $\{26, 30, 34, 38, 42, 46\}$ |
| | | 2 | $\{27, 35, 39, 51\}$ | | 2 | $\{18, 38, 42\}$ |
| | | 3 | $\{23, 35, 47\}$ | | 3 | $\{34, 38, 58\}$ |
| | | 4 | $\{19\}$ | | 4 | $\{30, 34\}$ |
| | | | | | 5 | $\{26, 30\}$ |
| | | | | | 6 | $\{26\}$ |
| | 2 | 0 | $\{13, 17, 21, 25, 29, 33, 37, 41, 45\}$ | 3 | 0 | $\{16, 20, 24, 28, 32, 36, 40, 44, 48\}$ |
| | | 1 | $\{41, 53, 57\}$ | | 1 | $\{28, 32, 36, 40, 44, 48, 52\}$ |
| | | 2 | $\{37, 53\}$ | | 2 | $\{40, 44, 48, 52, 56\}$ |
| | | 3 | $\{33\}$ | | 3 | $\{28, 36, 44, 52\}$ |
| | | 4 | $\{29, 37, 41, 45, 49, 53\}$ | | 4 | $\{32, 40, 48\}$ |
| | | 5 | $\{25, 49\}$ | | 5 | $\{28, 36\}$ |
| | | 6 | $\{21, 45\}$ | | 6 | $\{24, 32\}$ |
| | | 7 | $\{17, 41\}$ | | 7 | $\{20\}$ |
| | | 8 | $\{13, 37\}$ | | 8 | $\{16\}$ |
| $t_i$ | 0 | 0 | $\{15, 19, 23, 27, 31, 35, 39, 43\}$ | 1 | 0 | $\{10, 14, 18, 22, 26, 30, 34\}$ |
| | | 1 | $\{7, 11, 15, 19, 39\}$ | | 1 | $\{10, 14, 18, 30\}$ |
| | | 2 | $\{3, 15, 35\}$ | | 2 | $\{14, 26, 42\}$ |
| | | 3 | $\{11, 31\}$ | | 3 | $\{2, 10, 22\}$ |
| | | 4 | $\{7, 27\}$ | | 4 | $\{18\}$ |
| | | 5 | $\{23\}$ | | 5 | $\{14\}$ |
| | | 6 | $\{19\}$ | | 6 | $\{10\}$ |
| | | 7 | $\{15\}$ | | | |
| | 2 | 0 | $\{25, 29, 33, 37, 41\}$ | 3 | 0 | $\{8, 12, 16, 20, 24, 28, 32, 36, 40\}$ |
| | | 1 | $\{5, 9, 13, 37\}$ | | 1 | $\{4, 8, 12, 16, 20, 36\}$ |
| | | 2 | $\{9, 21, 33\}$ | | 2 | $\{0, 16, 32\}$ |
| | | 3 | $\{5, 29\}$ | | 3 | $\{12, 16, 28\}$ |
| | | 4 | $\{25\}$ | | 4 | $\{8, 24\}$ |
| | | | | | 5 | $\{4, 20\}$ |
| | | | | | 6 | $\{16\}$ |
| | | | | | 7 | $\{12\}$ |
| | | | | | 8 | $\{8\}$ |
| $s_i$ | 0 | 0 | $\{31, 35, 39, 43, 47, 51\}$ | 1 | 0 | $\{6, 10, 14, 38, 42, 46\}$ |
| | | 1 | $\{11, 15, 19, 23, 27, 57\}$ | | 1 | $\{10, 26, 30, 42\}$ |
| | | 2 | $\{15, 19, 23, 43\}$ | | 2 | $\{6, 26, 38, 46, 50, 54\}$ |
| | | 3 | $\{15, 19, 39, 47\}$ | | 3 | $\{50\}$ |
| | | 4 | $\{15, 35\}$ | | 4 | $\{54\}$ |
| | | 5 | $\{11, 31\}$ | | | |
| | 2 | 0 | $\{17, 21, 25, 29, 33, 37, 41, 45\}$ | 3 | 0 | $\{12, 16, 20, 24, 28, 32, 36, 40, 44\}$ |
| | | 1 | $\{9, 13, 17, 21, 25, 41\}$ | | 1 | $\{12, 16, 20, 24, 28, 32, 36, 40\}$ |
| | | 2 | $\{21, 25, 29, 33, 37, 41, 45, 49\}$ | | 2 | $\{16, 20, 24, 32, 36\}$ |
| | | 3 | $\{17, 29, 33, 37, 45\}$ | | 3 | $\{16, 20, 28, 32\}$ |
| | | 4 | $\{13, 29, 33, 41\}$ | | 4 | $\{16, 24, 28\}$ |
| | | 5 | $\{9, 25, 37\}$ | | 5 | $\{20, 24\}$ |
| | | 6 | $\{21, 33\}$ | | 6 | $\{16, 20\}$ |
| | | 7 | $\{17, 29\}$ | | 7 | $\{12, 16\}$ |
| | | | | | 8 | $\{12\}$ |

## G Circuit Details for PRESENT

Note that the sequence of operation in PRESENT are as follows:

PRESENT **Datapath**

```
1. For $i = 1 \to 31$ do
     addRoundkey(STATE,$K_i$)
     sBoxLayer(STATE)
     pLayer(STATE)
2. addRoundkey(STATE,$K_{32}$)
```

PRESENT **Keypath**

```
1. For $i = 1 \to 32$ do
     $K_i = [k_{79}, k_{78}, \ldots, k_{16}]$
     $[k_{79}, k_{78}, \ldots, k_1, k_0] \leftarrow [k_{18}, k_{17}, \ldots, k_{20}, k_{19}]$
     $[k_{79}, k_{78}, k_{77}, k_{76}] \leftarrow S[k_{79}, k_{78}, k_{77}, k_{76}]$
     $[k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \leftarrow [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \oplus i$
```

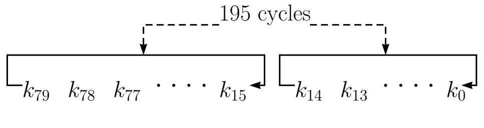
In order to explain the circuit operations, it is most instructive to give a cycle by cycle explanation of the flow of data in the registers.

**First 80 cycles:** In this period the plaintext and key are loaded onto the state and key registers bit by bit. We initiate a register **Cycle** which is reset to zero at the end of the key and plaintext loading.

**Cycle 0 to 63:** This period is used for adding the roundkey to the state bits and then a subsequent S-box operation. Although key addition and the subsequent register updates are done bitwise, it is possible to execute the 4-bit S-box operation by using the idea introduced in [JMPS17]. In Figure 3, we can see that the last 4 flip-flops in the circuit are in fact scan flip-flops which will help in the S-box operation. In the first 3 cycles of every 4-cycle period, the SB signal that controls these flip-flops are kept at zero so that in these 3 cycles the updated value is the addition of the corresponding state and keybits without the S-box operation. In the 4th cycle of this 4-cycle period, the SB signal is changed to 1 so that 4 bit output of the S-box is updated en-masse in this cycle.

**Cycle 64 to 1535:** The next 1472 cycles are used to implement the permutation layer as explained in the previous sub-section. The Sel port that controls the $61^{st}$ flip-flop is fed the signals from the $\overrightarrow{\mathsf{Sel}}$ vector constructed in the previous section. The **Cycle** register is reset to zero at the end of this period.

The above procedure is repeated 31 times. In the $32^{nd}$ iteration the first 64 cycles are used for the final roundkey addition operation and the ciphertext is available at the output of the xor gate that does the key addition. The keypath operations are slightly more involved. We need to perform the key update operations correctly, and at the same time ensure that the correct roundkey bit is available during the roundkey addition operation. The key update operation rotates the 80-bit key towards the left by 61 bits, then applies the s-box to a fixed nibble and then adds the round-constant to another fixed 5 bit chunk. The main concern therefore is to ensure that after the completion of a round, which in this case consists of 1536 cycles, the key is rotated by exactly 61 bits. It may have been possible to achieve this using a gated clock in the key registers that freezes the update operations for certain period of time. But clock gating requires some logic of its own and our intention was to see if we could achieve the required functionality without resorting to gating. Note that if we were to let the key register rotate uninterrupted for 1536 cycles, we would achieve a left key rotation of 1536 mod 80 = 16 bits. However if we rotate the register for $\beta$ cycles such that $\beta \equiv 61 \bmod 80$ and somehow freeze the rotation for the remaining $1536 - \beta$ cycles, we would achieve the required functionality. We chose $\beta = 1341$, which would require freezing the rotation operation for 195 cycles. To achieve this we use a scan flip-flop in the $15^{th}$ location, controlled by a Rtx signal. When the Rtx signal is 1, the key register performs internal rotation between the first 65 and the next 15 bit chunks as shown in the following figure.


195 cycles
$k_{79}$ $k_{78}$ $k_{77}$ · · · · $k_{15}$
$k_{14}$ $k_{13}$ · · · · $k_0$

Since 195 is a multiple of both 15 and 65, such internal rotation when performed for 195 clock cycles, results in the identity function, and so we achieve our end objective of arresting rotation for exactly 195 cycles. For this purpose, one can choose any 195 of the 1536 cycles used in every round, except of course for the first 64 when the key addition is being performed. There are additional control signals KB that like SB in the case of of the state path controls the S-box operation in the key registers. And the AddC signal controls addition with round constants. These signals are set to 1 at appropriate cycles to ensure the respective functionalities.

### Circuit for Encryption+Decryption

There are few additions to the combined circuit, as compared to the encryption-only circuit that are listed below:

- There is an additional circuit for the PRESENT inverse S-box.
- The order of operations in the decryption process is listed as follows:

PRESENT **Datapath**

```
1. addRoundkey(STATE,$K_{32}$)
2. Inv-pLayer(STATE)
3. For $i = 31 \to 2$ do
     Inv-sBoxLayer(STATE)
     addRoundkey(STATE,$K_i$)
     Inv-pLayer(STATE)
4. Inv-sBoxLayer(STATE)
5. addRoundkey(STATE,$K_1$)
```

PRESENT **Keypath**

```
1. $K_{32} = [k_{79}, k_{78}, \ldots, k_{16}]$
2. For $i = 31 \to 1$ do
     $[k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \leftarrow [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \oplus i$
     $[k_{79}, k_{78}, k_{77}, k_{76}] \leftarrow S^{-1}[k_{79}, k_{78}, k_{77}, k_{76}]$
     $[k_{79}, k_{78}, \ldots, k_1, k_0] \leftarrow [k_{60}, k_{59}, \ldots, k_{62}, k_{61}]$
```

The sequence of operations during decryption is slightly different. So let us look at the sequence of operations in each cycle:

**First 80 cycles:** As usual the ciphertext and key are loaded onto the respective registers.

**Cycle 0 to 63:** In the round immediately after ciphertext loading, we perform only bitwise round key addition in this period. However in all the subsequent rounds, we need to do an Inverse s-box operation before roundkey addition. This would require some incremental additions to the circuit. First of all we need a 4-bit xor to do the key addition instead of just a single bit xor in the encryption path. A four bit multiplexer is additionally required to select between the 4-bit updates during encryption and decryption. The logic circuit is explained diagrammatically in Figure 14.

**Cycle 64 to 1535:** The next 1472 cycles are used to implement the inverse permutation layer, with the $\mathcal{S}$ executed ahead of $\mathcal{T}$.

- The keyschedule involves addition by round constant followed by application of inverse s-box on a fixed nibble followed by rotation by 19 bits to the left. As before we need to try to rotate the key register for $\beta \equiv 19 \bmod 80$ cycles and somehow arrest the rotation for the remaining $1536 - \beta$ cycles. We choose $\beta = 1399$, which requires stopping the rotation for 237 cycles. Again, we try to achieve this by breaking up the key into chunks of 79 and 1 bits and doing internal rotation within the key-chunks for 237 cycles. Since 237 is a multiple of 79 and 1, internal rotation for 237 cycles again gives the identity transformation which satisfies our end objective. In terms of hardware, this requires two extra multiplexers to do the internal rotation as shown in Figure 14.

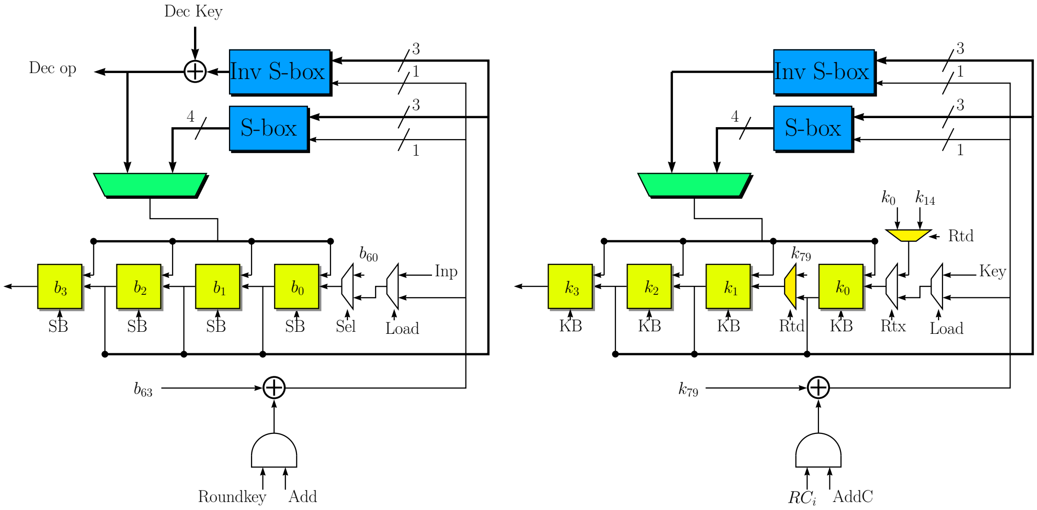


**Figure 14:** Modified logic around the last 4 flip-flops to accommodate decryption

## H Circuit Details for GIFT

The sequence of operations in the data and keypaths are as follows:

GIFT Datapath

```
1. For $i = 1 \rightarrow 28$ do
    sBoxLayer(STATE)
    pLayer(STATE)
    addRoundkey(STATE,$RK_i$)
```

GIFT Keypath

```
1. For $i = 1 \rightarrow 28$ do
    $K_i = [k_{127}, k_{78}, \dots, k_0]$
    For $j = 0 \rightarrow 7$: $L_j \leftarrow [k_{16j+15}, k_{16j+15}, \dots, k_{16j}]$
    $RK_i = L_1 || L_0$
    $L_7 || L_6 || \cdots || L_0 \leftarrow L_1 \ggg 2 || L_0 \ggg 12 || L_7 || \cdots || L_2$
```

So the sequence of operations in the datapath is as follows:

**First 128 cycles:** In this period the key is loaded onto the state and key registers bit by bit. In cycles 64 to 127, the plaintext is loaded onto the state register after performing the s-box operation. Thereafter we have 28 iterations of the following operations.

**Cycle 0 to 1727:** Used to compute the permutation layer.

**Cycle 1728 to 1791:** The next 64 cycles are used to compute add roundkeys and then perform the s-box operation of the next round.

Thus the total number of cycles taken for the encryption routine is $128 + 28 \times 1792 = 50304$. The keyschedule is slightly more complicated: it breaks up the current key into eight 16 bit words $L_7$ to $L_0$. $L_1$ and $L_0$ are internally right rotated by 2, 12 bits respectively and the whole key is then right rotated by 32 bits. In other words this means internal left rotation of $L_1$ and $L_0$ by 14, 4 bits and overall left rotation by 96 bits. So we do the following:

- Let the key register rotate left for 96 cycles. After this $L_1$ and $L_0$ occupy the most significant 32 bits of the register($k_{127}$ to $k_{96}$).

- At this point of time we will partition the key register into chunks of 16 ($k_{127}$ to $k_{112}$), 16 ($k_{111}$ to $k_{96}$) and 96 bits ($k_{95}$ to $k_0$) and do an internal rotation for 288 cycles. Since $1792 - 288 = 1504 \equiv 96 \bmod 128$ this achieves our first objective of 96 bit left rotation.

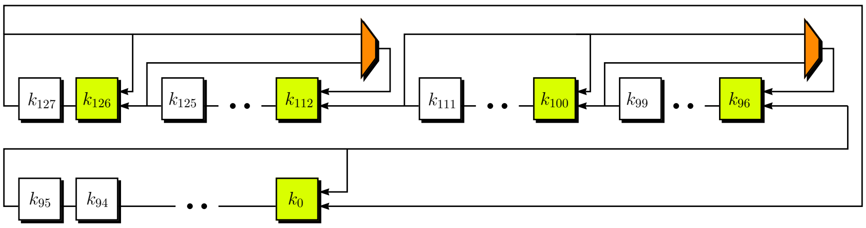


**Figure 15:** The GIFT key register

- To achieve left rotation of $L_1$ by 14 bits, we partition the 1st 16 MSBs into chunks of 2 ($k_{127}$ to $k_{126}$) and 14 ($k_{125}$ to $k_{112}$) bits and do an internal rotation in these 2 groups for 98 cycles, and do a normal rotation over ($k_{127}$ to $k_{112}$) over the remaining 288-98=190 cycles (see figure 15). Since 2 and 14 both divide 98, the rotation results in identity transformation. So the effective rotation is for $190 \equiv 14 \bmod 16$ cycles.
- Similarly rotating $L_0$ by 4 bits, we partition 2nd 16 MSBs into chunks of 12 ($k_{111}$ to $k_{100}$) and 4 ($k_{99}$ to $k_{96}$) bits. Internal rotation is carried out for in these smaller chunks 12 cycles. So effectively we rotate the 2nd chunk by $276 \equiv 4 \bmod 16$ bits.

However the key addition in GIFT is quite complicated: neighboring key bits do not xor with neighboring state bits as in PRESENT. In fact, the designers recommend that $\forall i \in [0, 31]$ the $i^{th}$ bit of $L_1$ be xored with the $(4i+2)^{nd}$ state bit and the the $i^{th}$ bit of $L_0$ be xored with the $(4i+1)^{st}$ state bit. Thus, the circuit also requires a filter to extract the correct roundkey bit in every cycle, which increases the total area slightly.

## I Combined Circuit for encryption and decryption

The GIFT decryption circuit suffers from the same issues as the corresponding PRESENT circuit, and therefore the circuit for the combined decryption is same as the one outlined in Figure 14. The only differences are in the order in which the functions are carried out. The following is the sequence of operations:

```
GIFT Datapath

1. addRoundkey(STATE,$RK_{28}$)
2. inv-pLayer(STATE)
3. For $i = 27 \to 1$ do
     inv-sBoxLayer(STATE)
     addRoundkey(STATE,$RK_i$)
     inv-pLayer(STATE)
4. inv-sBoxLayer(STATE)
```

```
GIFT Keypath

1. For $i = 1 \to 28$ do
     $K_i = [k_{127}, k_{78}, \ldots, k_0]$
     For $j = 0 \to 7$: $L_j \leftarrow [k_{16j+15}, k_{16j+15}, \ldots, k_{16j}]$
     $RK_i = L_1 || L_0$
     $L_7 || L_6 || \cdots || L_0 \leftarrow L_5 || \cdots || L_0 || L_7 \lll 2 || L_6 \lll 12$
```

As expected, the inverse permutation layer is constructed by executing the $s_i$ transpositions first, followed by the $t_i$'s and then the $u_i$'s. The cycle by cycle execution of operations is as follows:

**First 128 cycles:** In this period the key is loaded onto the state and key registers bitwise. In cycles 64 to 127, the plaintext is loaded onto the state register without performing the inverse s-box operation. Thereafter the next operations are executed 28 times.

**Cycle 0 to 63:** Used for executing the inverse s-box operations followed by roundkey addition as shown in Figure 14. As in PRESENT only in the first round, the inverse s-box operation is omitted.

**Cycle 64 to 1791:** Used for executing the inverse p-layer.

After this, the GIFT decryption process requires one more inverse s-box operation. Hence the decryption operation requires an additional 64 cycles to complete. The key schedule for decryption can be carried out using the same circuit as in Figure 15. We need left rotation of $L_7$, $L_6$ by 2, 12 bits followed by a left rotation by 32 bits. At the beginning of the round cycle when $L_7$, $L_6$ still occupy the 32 MSBs in the key register we do internal rotation for 96 cycles. It is easy to see to verify that this will achieve left rotation by 32 bits. In these 96 cycles, we do further internal rotation between the 2 and 14 bit chunks ($k_{127}$ to $k_{126}$ and $k_{125}$ to $k_{112}$) for 14 cycles, and for 36 cycles between the 12 and 4 bits chunks ($k_{111}$ to $k_{100}$ and $k_{99}$ to $k_{96}$) for 36 cycles. This is sufficient to achieve the required functionalities in the inverse keyschedule.

## J Python code for GIFT permutation

We present a simple python3 code that simulates how GIFT permutation layer is operated over the pipeline with the help of six swap operations. The swaps are $(24, 12)$, $(37, 13)$, $(50, 14)$, $(61, 45)$, $(62, 30)$, $(63, 15)$.

In Figure 16, the list S represents the 64 bits stored in the pipeline. Let $A_{63}, A_{62}, \ldots, A_0$ be the sequence of bits that needs to be permuted. These bits arrive to S[0] one at a time fashion in the first round. For $i \in \{0, 1, \ldots, 63\}$, $A_{63-i}$ is loaded into S[0] at the end of the $i$-th cycle (this variable is denoted with count in line 62 of the code). All bits are completely stored in the pipeline S at the end of the 63-th cycle of the round. Hence the permuted bits can be read from S[63] between cycles 0 to 63 of the next round.

In this example, we permute $31 \times 64$ bits (inputbits) with GIFT permutation, in exactly $32 \times 64 = 2048$ cycles. Additional 64 cycles are incurred not because of our permutation, but due to the time it takes to fill and flush the pipeline with state bits. This already happens in a serial implementation regardless of our permutation layer, as bits are fed into the circuit one bit each cycle. In other words, simply filling $31 \times 64$ bits into the pipeline and then waiting them to completely flush out from the pipeline would also take precisely $32 \times 64$ cycles in total. Therefore, our permutation in fact operates seamlessly in parallel without incurring no latency.

```
GIFT = [ 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 ]

cycles1 = [ [29, 30, 31, 32, 49, 50, 51, 52, 5, 6, 7, 8],
            [46, 47, 48, 49, 2, 3, 4, 5],
            [63, 0, 1, 2] ]

cycles2 = [ [0, 4, 8, 12, 14, 18, 22, 26, 32, 36, 40, 44],
            [2, 6, 10, 14, 16, 20, 24, 28],
            [0, 4, 8, 12] ]

swaps1 = [(24, 12), (37, 13), (50, 14)]
swaps2 = [(61, 45), (62, 30), (63, 15)]


def apply_permutation(X, perm):
    Z = [’ ’] * 64
    for i in range(64):
        Z[63-perm[i]] = X[63-i]
    return Z


def print_pipe(S):
    Z = list(reversed(S))
    print(Z[:16])
    print(Z[16:32])
    print(Z[32:48])
    print(Z[48:])
    print()


def executeSwaps(S, round, count):
    disableSwaps1 = (round == 0 and count < 29) or (round == 31 and count > 28)
    disableSwaps2 = round == 0
    for j in range(3):
        if count in cycles1[j] and not disableSwaps1:
            (x, y) = swaps1[j]
            (S[x], S[y]) = (S[y], S[x])
    for j in range(3):
        if count in cycles2[j] and not disableSwaps2:
            (x, y) = swaps2[j]
            (S[x], S[y]) = (S[y], S[x])
    return S


def rotate_pipieline(S):
    return [S[-1]] + S[:-1]

def store_input_bit(S, bit):
    S[-1] = bit
    return S

def read_exit_bit(S):
    return S[-1]

def simulate_permutation_over_pipeline(inputbits):
    K = [[None]*64 for _ in range(32)] # to store output bits from pipeline
    S = [’___’] * 64 # keeps the contents of the pipeline
    for round in range(32):
        for count in range(64):
            print("at the beginning of round " + str(round) + "\tcylce: " + str(count))
            S = executeSwaps(S, round, count)
            print_pipe(S)
            K[round][count] = read_exit_bit(S)
            S = store_input_bit(S, inputbits[round][count])
            S = rotate_pipieline(S)
    return K[1:]

stateletters = [chr(y+65) for y in range(26)] + [chr(y+97) for y in range(6)]
inputbits = [[str(x) + y for x in range(63, -1, -1)] for y in stateletters]
results = [apply_permutation(x, GIFT) for x in inputbits]
results = results[:-1] # the last 64 bits are not used, they are placeholder
K = simulate_permutation_over_pipeline(inputbits)

print("An example output from the first round of the pipeline:")
print_pipe(K[0])
print("Expected result: ")
print_pipe(results[0])


if K == results:
    print(str(len(results)) + "*64 bits are permuted correctly")
```

**Figure 16:** Example code for GIFT permutation using 6 swaps in 64 cycles.