Efficient Constant-Time Implementation of SM4

with Intel GFNI instruction set extension and Arm NEON coprocessor

Weiji Guo

bilibili, Shanghai, China, guoweiji@bilibili.com

Abstract. The efficiency of constant-time SM4 implementation has been lagging behind that of AES for most internet traffic and applicable data encryption scenarios. The best performance before our works was 3.77 cpb for x86 platform (AESNI + AVX2), and 8.62 cpb for Arm platform (NEON). Meanwhile the state of art constant-time AES implementation could reach 0.63 cpb. Dedicated SM4 instruction set extensions like those optionally available in Armv8.2, could achieve comparable cpb to AES. But they are only available in limited processors, therefore does not impact much to real-world uses. To fill the gap we explored some novel techniques with Intel GFNI instruction set extension and Arm NEON coprocessor. We achieved 1.51 cpb with GFNI + AVX512 and 2.62 cpb with GFNI + AVX2 for Intel processors; we also achieved 6.74 cpb with NEON. In addition, we simplified the algebraic expression of SM4 S-Box. And our technique to exploit L1 cache could also be applied to other applications and hardware platforms if the circumstances apply.

Keywords: Constant-Time · SM4 · S-Box · SIMD · Intel GFNI · Arm NEON · Cache

1 Introduction

SM4 is the block cipher of the Chinese commercial cipher standard (GB/T 32907-2016), and has been standardized internationally (ISO/IEC 18033-3:2010/AMD 1:2021). It also serves as the block cipher for two informal cipher suites of TLS 1.3 [Yan21], which are expected to be widely used in China.

SM4 shares a similar S-Box construct with AES, which is known to be vulnerable to timing attacks [Ber05]. Unlike AES, which has been supported with its own Instruction Set Extension (ISE) in various hardware platforms, options for constant-time SM4 are rather limited.

1.1 Previous works

For **x86 platforms**, Saarinen had proposed the sm4ni method [Saa19] to use the AESNI ISE to achieve both constant-timeness and efficiency for SM4 by leveraging the isomorphism between the $GF(2^8)$ field of SM4 and that of AES. It had been implemented in libgcrypt by Kivilinna [Kiv20] and ported to a few other projects. Our tests suggested 3.77 cycles per byte (cpb) with AVX2 in Intel i5-1038NG7 in ECB mode. This is several times better than variable time software implementations. However, it still costs at least 14 instructions to compute the S-Box [1].

For **Arm platforms**, there is a solution to lookup S-Box within NEON for AES [Bie17]. It loads the S-Box into 16 continuous NEON registers, then looks up with 1 **tbl** instruction

[1]there are totally 14 computation steps in the original project [Saa19]. Most of the instructions have latency of 1. The latency of the aesenclast instruction varies with microarchitectures, and is mostly 3 cycles or more.

call followed by 3 **tbx** instruction calls. 2x or 4x interleaving could be adopted to hide the latency from **tbl** and **tbx** instructions. Porting the solution to SM4 is straightforward for up to 2x interleaving. Kwon etc. even achieved 8.62 cpb by managing 3x interleaving [KKE+21]. A naive expansion to 4x interleaving is infeasible due to the lack of sufficient NEON registers, as we will show in Subsection 4.1.

As comparison, constant-time AES could reach 0.63 cpb with AESNI [BLT16] [2].

1.2 Our contributions

To deliver highly efficient and constant-time SM4 implementations for mainstream server, desktop and mobile platforms so that most SM4 traffic and usage could be secured, we strived further optimizations. This is part of our efforts to develop an efficient constant-time library for Chinese commercial ciphers.

Our contributions are:

First, we achieved 1.51 cpb in Intel platform with GFNI + AVX512, and 6.74 cpb in Arm platform with NEON. For the latter, the optional SM4 ISE can perform much better, for example [Hu22] reported 0.62 cpb [3]. However this ISE is not widely available, especially not in desktop and mobile platforms where our technique could be deployed.

Second, we exploited the L1 cache to boost performance in case there are insufficient registers to hide the latency from certain instructions. We believe we are the first to develop such a technique. Despite the costs of cache, we achieved 61% performance improvements with twice the parallel blocks. Further, this technique could also be applied to other data intensive computations, within or outside of cryptology.

Third, we gave by-far the simplest algebraic expression for SM4 S-Box. The expression initially provided by Liu etc. [LJH+07] uses matrix left-multiply and the values calculated don't equal to the lookup results. The fixing to that in [EDC10] correctly adopts matrix right-multiply, however with four parameters while we only need two.

1.3 Overview of this paper

The rest of this paper is organized as follows: Section 2 briefly introduces SM4. Section 3 gives background information and then detailed descriptions of our works with GFNI. Section 4 details our works on Arm NEON coprocessors *without* the SM4 ISE. Section 5 concludes this paper.

2 Preliminary

This section briefly introduces the SM4 algorithm following notation of [SCA12]. SM4 has 32 rounds of iteration with both block size and key size in 128 bits. Accordingly the key is expanded to 32 round keys, each of which has 32 bits: $(rk^0, rk^1, ..., rk^{31}), rk^i \in Z_2^{32}$.

For i^{th} round, as illustrated in Figure 1, let input data be: $(X_0^i, X_1^i, X_2^i, X_3^i) \in (Z_2^{32})^4$, and round key be $rk^i \in Z_2^{32}$, then the round function is:

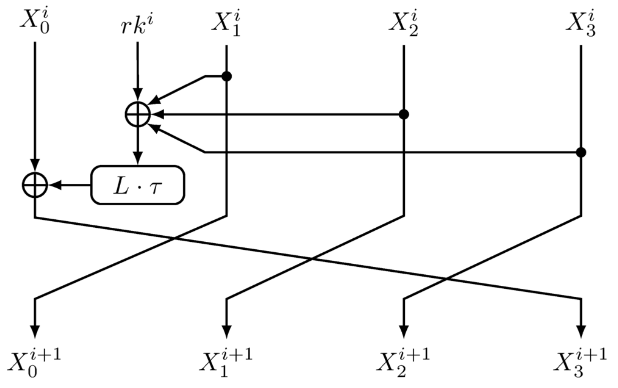
$$F(X_0^i, X_1^i, X_2^i, X_3^i) = X_0^i \oplus L(\tau(X_1^i \oplus X_2^i \oplus X_3^i \oplus rk^i))$$

$\oplus$ is addition over $GF(2)$, equivalent to bit-wise exclusive-or. The non-linear transformation τ is where the S-Box lookup is performed:

$$\tau(a_0, a_1, a_2, a_3) = (Sbox(a_0), Sbox(a_1), Sbox(a_2), Sbox(a_3)), a_i \in Z_2^8$$

Full SM4 S-Box is provided in Table 4 of Appendix A.

[2]data excerpted from Table 1, with fixed data length of 2KB, in ECB mode
[3]translated from 4459181.40kBps@2.75 GHz for ECB mode and block size of 8KB.



$X_0^i, X_1^i, X_2^i, X_3^i \in Z_2^{32}$ are inputs to i^{th} round, $rk^i \in Z_2^{32}$ is i^{th} round key, τ and L are SM4-defined transformations.

Figure 1: SM4 round function

The linear transformation L simply adds together several left rotations of the input data, which is the output of the τ transformation. Let $B \in Z_2^{32}$, and $<<<$ be left rotation, then:

$$L(B) = B \oplus (B <<< 2) \oplus (B <<< 10) \oplus (B <<< 18) \oplus (B <<< 24)$$

We note that although the block size and key size for SM4 are both 128 bits, the calculations are usually carried out on 32 bits data for each round. Therefore a SIMD instruction of 128-bit data width could calculate 4 data blocks in parallel, and 256-bit for 8, 512-bit for 16.

For simplicity we omitted the key expansion, which also involves S-Box lookup and should be subject to similar vulnerabilities. We note that our implementation does calculate the key expansion in constant-time as well.

3 Intel GFNI implementation of SM4 S-Box

This section shows how Intel GFNI ISE could be leveraged to implement SM4 S-Box. We first introduce the algebraic expression of the S-Box, and then the expressions for the two GFNI instructions to be used. Next we combine these expressions together and simplify them by merging adjacent matrixes. Finally we give our implementation results with further discussion.

3.1 Algebraic expression of SM4 S-Box

The algebraic expression of SM4 S-Box had been discovered in [LJH+07], in the form of row vector left-multiply matrix as affine transformation. However it does not derive values consistent with table lookup, as shown in Appendix A. We fixed the expression by rewriting it in matrix right-multiply column vector as in (1). The multiplication $\cdot$ and addition $\oplus$ are over $GF(2)$. $(\cdot)^{-1}$ is the multiplicative inverse over F_S, the SM4 field of $GF(2^8)$ as defined in (2).

$$Sbox(X) = A \cdot (A \cdot X \oplus C)^{-1} \oplus C \tag{1}$$

$$F_S = GF(2)[x]/(x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1) \tag{2}$$

The parameters, 8×8 matrix A and column vector C over $GF(2)$ are given in (3). We follow the little-endian encoding scheme, with the least significant bit in the top left.

Therefore C encodes $(1 + x + x^4 + x^6 + x^7)$, and the circulant matrix A encodes 64 bits with the least significant byte on the top and the least significant bit of each byte on the left.

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{3}$$

3.2 the Intel GFNI instruction set extension

The Intel Galois Field New Instructions or GFNI for short, was introduced in the 3rd Gen Intel Xeon Scalable processors. GFNI instructions are designed for computing over Galois fields. For our purposes here we need two instructions from GFNI, namely **vgf2p8affineqb** for affine transformation as (4), and **vgf2p8affineinvqb** for affine-inverse transformation as (5), which is byte inversion over F_A and then fused affine transformation. The caller-provided parameter M should be a 8×8 matrix and B should be a column vector, both over $GF(2)$. F_A as the AES field of $GF(2^8)$, is shown in (6).

$$\texttt{affine}\ (X, M, B) = M \cdot X \oplus B \tag{4}$$

$$\texttt{affineInv}\ (X, M, B) = M \cdot X^{-1} \oplus B \tag{5}$$

$$F_A = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1) \tag{6}$$

GFNI instructions are capable of computing affine or affine-inverse transformation for multiple inputs, up to 64 input bytes with AVX512, or 32 with AVX2, 16 with AVX. For example, X could be some AVX512 register containing 64 bytes, while M and B remain the same for all these input bytes.

3.3 Combining expressions together

Let $T_{S2A} := F_S \to F_A$ be one of the isomorphic mappings from F_S to F_A, represented by matrix S_A, and $T_{A2S} = T_{S2A}^{-1}$ its inverse mapping, represented by matrix A_S. (7) gives one such pair among 8 possibilities.

$$S_A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \qquad A_S = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{7}$$

Since (5) accepts inputs from F_A, elements of F_S must be transformed to F_A before calling **vgf2p8affineinvqb**, and then back to F_S for further affine transformation as required by (1). Let $X \in F_S$ and $Y \in F_A$, rewrite the SM4 S-Box expression (1) as two formulas: (8) followed by (9) in a successive call, with Y as output of (8) and then input to (9).

$$Y = S_A \cdot (A \cdot X \oplus C) \tag{8}$$

$$Sbox(X) = A \cdot (A_S \cdot Y^{-1}) \oplus C \tag{9}$$

Further, let $A_1 = S_A \cdot A$, $C_1 = S_A \cdot C$, $A_2 = A \cdot A_S$, rewrite (8) as (10) and (9) as (11), where values of A_1, C_1, A_2 are listed in (12).

$$Y = A_1 \cdot X \oplus C_1 \tag{10}$$

$$Sbox(X) = A_2 \cdot Y^{-1} \oplus C \tag{11}$$

$$A_1 = \begin{bmatrix} 0&0&1&1&0&0&1&0\\ 0&0&0&1&0&1&0&0\\ 1&0&1&1&1&1&1&0\\ 1&0&0&1&1&1&0&1\\ 0&1&0&1&1&0&0&0\\ 0&1&0&0&0&1&0&0\\ 0&0&0&0&1&0&1&0\\ 1&0&1&1&1&0&1&0 \end{bmatrix} \quad C_1 = \begin{bmatrix} 0\\1\\1\\1\\1\\1\\0\\0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1&1&0&0&1&1&1&1\\ 1&1&0&1&0&1&0&1\\ 0&0&1&0&1&1&0&0\\ 1&0&0&1&0&1&0&1\\ 0&0&1&0&1&1&1&0\\ 0&1&1&0&0&1&0&1\\ 1&0&1&0&1&1&0&1\\ 1&0&0&1&0&0&0&1 \end{bmatrix} \tag{12}$$

It then follows that the affine transformation of (10) could be computed with the **vgf2p8affineqb** instruction:

$$Y = \texttt{affine}\,(X, A_1, C_1)$$

and the affine-inverse transformation of (11) could be computed with the **vgf2p8affineinvqb** instruction:

$$Sbox(X) = \texttt{affineInv}\,(Y, A_2, C)$$

3.4 Implementation results and discussion

We implemented SM4 S-Box with these two GFNI instructions. With 16 parallel data blocks we achieved 1.51 cpb in macOS 12.5 running over Intel Core i5-1038NG7 CPU (Ice Lake).

For comparable settings, we also ported the libgcrypt implementation [Kiv20] of the sm4ni method [Saa19] to Go Assembly, then tested and compared the performance on the same computer. Implementation of both methods only differ in S-Box handling, and the sm4ni method needing to load a few more parameters. Table 1 lists the results.

Table 1: GFNI vs sm4ni

	AVX	AVX2	AVX512
sm4ni (baseline)	6.99	3.77	1.92
GFNI (this work)	5.28	2.62	1.51
cpb gain	1.71	1.15	0.41
speedup	1.32×	1.44×	1.27×

Benchmarking results with Intel Core i5-1038NG (Ice Lake). All data were measured with minimal parallel blocks in order *not* to amortize the setup costs. Numbers are in cpb except for the speedup.

According to Intel Intrinsics Guide [Int22], the latency for both GFNI instructions is 3 and the reciprocal throughput is 0.5 to 1 with Ice Lake. Therefore the S-Box computation finishes in 6 clock cycles. Further optimization could calculate 32 or more parallel data blocks and interleave the GFNI instructions to hide the latency, and potentially to leverage the multi-issuing of GFNI instructions.

We note that we only benchmarked ECB mode with minimal data size (256 bytes for AVX512, 128 bytes for AVX2 and 64 for AVX). This reflects the actual performance benefits of our method. Handling larger or arbitrary data size could amortize some setup costs and result in better numbers. We plan to do so for production release.

We also note that GFNI requires newer processors and is not available in AMD processors by the time we wrote this paper.

4 Arm NEON implementation

This section first briefly introduces how to lookup S-Box within NEON, then shows the difficulties applying it to SM4 fully, and how to resolve these issues. Finally we give our implementation details and results, with further discussion.

4.1 Table look-up within NEON

Biesheuvel implemented S-Box lookup for AES with the **tbl** and **tbx** instructions of the Arm NEON vector coprocessor [Bie17]. This technique works as the following: 1) load the 256 elements of S-Box into 16 continuous NEON registers (V_{16-31}) once; 2) for each round of iteration, use **tbl** once then **tbx** three times to lookup the table, each time querying from 64 elements in 4 continuous registers.

A **tbx** instruction with 4 registers to lookup from comes with a high latency, which is 6 for Arm Cortex-X1 [Arm21], and 8 in some other implementations including both icestorm and firestorm of Apple M1 [Joh21]. **tbl** is slightly better but only used one third of times of **tbx**.

[Bie17] hides the latency by interleaving 4 different AES states. However for SM4, 4x naive interleaving does not work. Figure 2 shows how 4x interleaving could have worked in an altered round function should there be sufficient NEON registers. Each of the inputs $(U_j, W_j, X_j, Y_j \in (Z_2^{32})^4, j = 0, 1, 2, 3)$ holds four 32-bit data sub-blocks, so 16 NEON registers are required for the states of 16 data blocks in a would-be 4x interleaving. On the other hand we show that the 4x interleaved lookup per se needs 29 NEON registers in Appendix B. The total requirement is therefore 45 registers, obviously exceeding 32.

We solved this insufficiency by exploiting L1 cache to hold the states during lookup.

4.2 Exploiting L1 cache

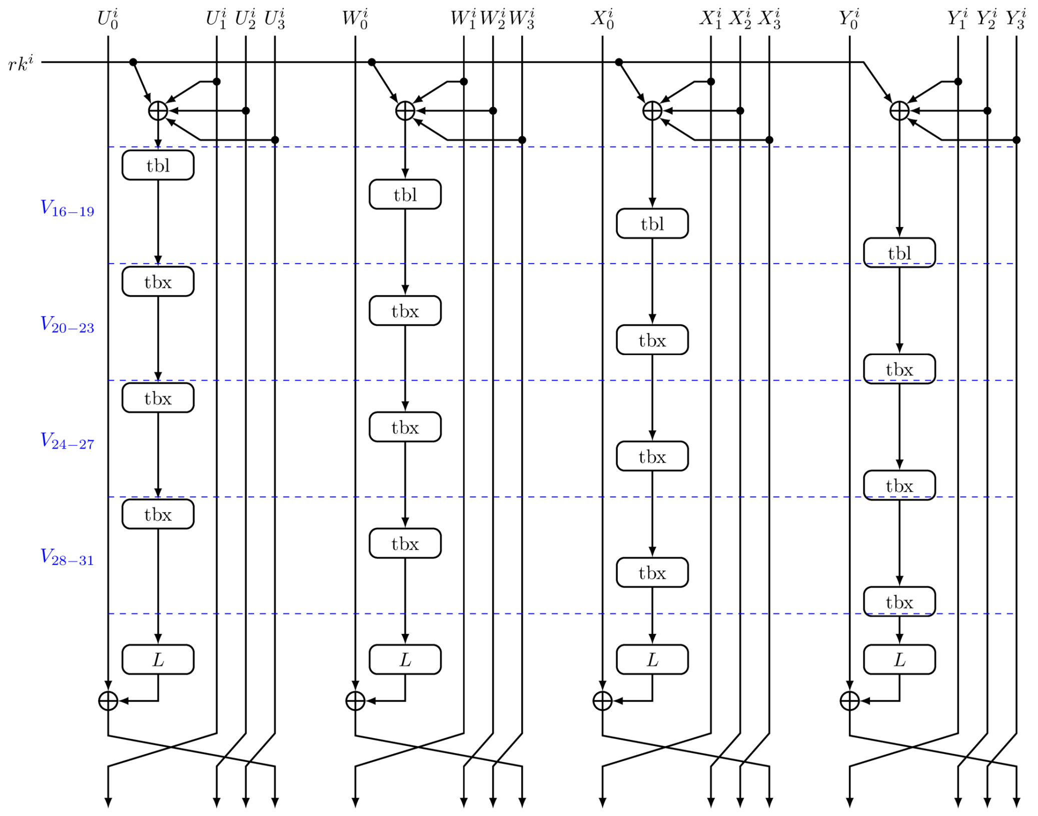
Lookup happens in the τ transformation. A close examination to Figure 1 reveals that X_1^i, X_2^i and X_3^i are not needed for lookup after they have been added with the round key rk^i. And X_0^i is read/written only after the L transformation. Therefore after calculating the input to τ transformation from the states, we could save away these states, load another set of states to the earlier occupied registers, and calculate more inputs for parallel τ transformation to hide the latency of **tbl** and **tbx**. By saving away states and carefully orchestrating all the registers, we managed 4x interleaving.

To save away the states, we simply store them into designated memory locations, and load them back later when data is needed again. In 32 rounds of iteration, the same memory locations will be accessed frequently enough so that the data content will be kept in L1 data cache, therefore loading them again could take as quickly as in 3 clock cycles, subject to microarchitecture implementations. Table 2 gives latency data for more processors, covering server, desktop and mobile platforms.

Table 2: L1 data cache latency (hit) of simple access via pointer

Processor	Microarchitecture	Latency
AMD Opteron A1170	Cortex-A57	4
Apple M1	Firestorm	3
Qualcomm Snapdragon 855	Cortex-A76	4

data collected from https://7-cpu.com



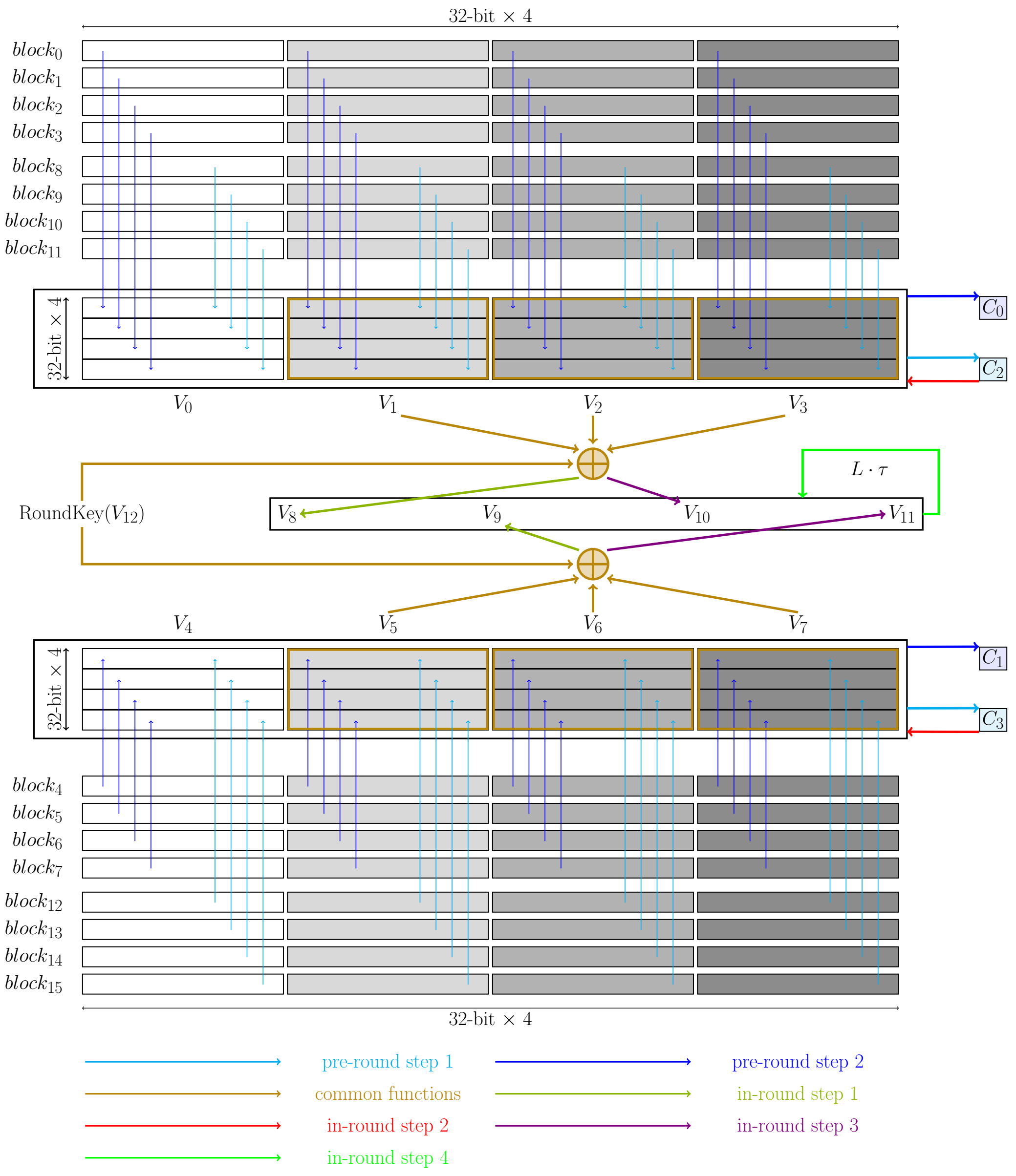
$U_j^i, W_j^i, X_j^i, Y_j^i \in (Z_2^{32})^4, j = 0,1,2,3$ are inputs to i^{th} round, round key rk^i is duplicated to 128 bit, tbl and tbx are NEON table lookup instructions (extension), and L is SM4-defined transformation. Interleaved execution of tbl and tbx are highlighted with dashed blue lines, between which the same instruction looks up data for different indexing & destination registers from the same set of sub-table as indicated in the left. In our implementation, we load the S-Box into NEON's upper 16 SIMD registers, from V_{16} to V_{31}. Then the four calls to tbl instruction lookup from V_{16-19} each time with different indexing and destination registers, first four calls to tbx instruction lookup from V_{20-23}, then from V_{24-27}, and finally from V_{28-31}.

Figure 2: SM4 would-be round function 4x.

Figure 3 and Figure 4 further detail how the registers are allocated, reused, saved to and loaded from cache. The SM4 encryption or decryption could be split into 3 phases: pre-round, in-round and post-round. For pre-round phase, step 1 and step 2 stride loads data from memory into registers and also manage to hit L1 cache by saving loaded data to designated memory locations. Stride loading is accomplished with the **ld4** instruction. For the in-round phase, there are 8 steps, which are shown in Figure 3 and then in Figure 4, separated due to the complexity of the data paths. In post-round phase encrypted or decrypted data are stride stored from register to memory with the **st4** instruction.

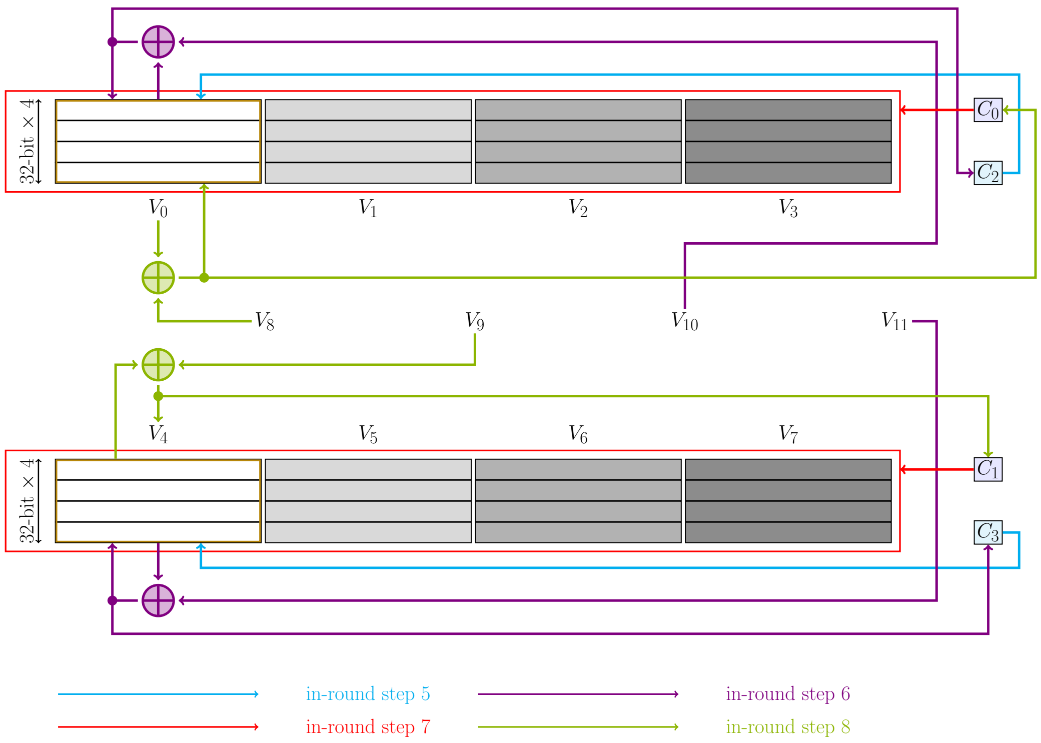
4.3 Implementation details and results

We implemented both the NEON table lookup and cache exploiting techniques. And we compared the performance of 4x interleaving with L1 cache exploiting against that of plain 2x with no cache exploiting. The results are shown in Table 3, which also includes performance data for non-interleaving version (4 parallel blocks) and optimized software implementation. The software implementation looks up S-Box (blended with L



Steps of the *first* (0^{th}) round of the 4x interleaving for 16 parallel data blocks, which are placed continuously from $block_0$ to $block_{15}$, and in units of 32-bit. pre-round step 1: Stride load from $block_{8-11}$ to V_{0-3} and from $block_{12-15}$ to V_{4-7}, in unit of 32-bit. Then save V_{0-3} to C_2, and V_{4-7} to C_3. C_i are designated memory addresses. pre-round step 2: Stride load from $block_{0-3}$ to V_{0-3}, and from $block_{4-7}$ to V_{4-7}. Then save V_{0-3} to C_0, and V_{4-7} to C_1. common functions and in-round step 1: $V_8 = V_1 \oplus V_2 \oplus V_3 \oplus V_{12}$, $V_9 = V_5 \oplus V_6 \oplus V_7 \oplus V_{12}$. in-round step 2: Load C_2 to V_{0-3} and C_3 to V_{4-7}. common functions and in-round step 3: $V_{10} = V_1 \oplus V_2 \oplus V_3 \oplus V_{12}$, $V_{11} = V_5 \oplus V_6 \oplus V_7 \oplus V_{12}$. in-round step 4: Run in-place τ and L transformations over V_{8-11}.

Figure 3: register allocation and execution plan (part 1)



(continued) in-round step 5: Load V_0 from C_2 and V_4 from C_3. in-round step 6: $V_0 = V_0 \oplus V_{10}$, $V_4 = V_4 \oplus V_{11}$. Save V_0 back to C_2, and save V_4 back to C_3. in-round step 7: Load C_0 to V_{0-3} and C_1 to V_{4-7}. in-round step 8: $V_0 = V_0 \oplus V_8$, $V_4 = V_4 \oplus V_9$. Save V_0 back to C_0, and save V_4 back to C_1.

Figure 4: register allocation and execution plan (part 2)

transformation) in memory, thus variable time. The tests were run on same Mac Book Pro with Apple M1 processor @3.2GHz. The exploiting of L1 cache contributes 61% more performance on top of the 2x interleaving. We did not implemented or benchmarked the 3x interleaving, but even with the reported performance data from [KKE+21], our cache exploiting technique still gain 28% more cpb.

Table 3: SM4 performance with Apple M1

	cache exploiting, 4x	2x interleaving	non-interleaving	variable time software
parallel blocks	16	8	4	1
cpb	6.74	10.96	19.92	20.87
speedup	1.61×	1.00×	0.55×	0.53×

1) cache exploiting: our contribution to enable 4x interleaving for SM4; 2) 2x interleaving: the baseline with NEON lookup; 3) non-interleaving: lookup without interleaving tbl/tbx instructions; 4) variable time software: optimized pure-software implementation with lookup blended with L transformation, using 4 tables of 256 elements of 32 bits

In our implementation, we could manage 4x interleaving with 30 NEON registers. But for performance reasons, all 32 are used during pre-lookup. For details, see Appendix B.

We note that the 4x interleaving table lookup per se only needs 29 NEON registers. So if we take care to leave V_0 and V_4 intact during lookup, we could skip in-round step 5. However we did not take this course for two reasons: 1) our implementation is based on

Go Assembly, and it does not support **tbx** instruction directly. We had to code machine words and hard coded register numbers. It is still possible but then the source code will be much longer; 2) benchmarking showed barely any gain. Also, given the effects of cache line, it does not matter to load V_0 / V_4 individually or V_{0-3} / V_{4-7} together in step 5.

We believe this is the first time cache is purposefully exploited to hide latency. And this technique could be used for other applications and hardware platforms when the number of register is the constraining factor for hiding latency for data intensive computations.

4.4 Discussion

A practical consideration is whether this cache exploiting technique is still effective under heavy load and high contention. The performance will certainly be impacted when cached data corresponding to C_i (Figure 3, Figure 4) are evicted from L1 or even from L2 or L3. However, we argue without testing data that the costs is negligible if the application is not frequently interrupted or preempted, or otherwise still acceptable.

Modern non-realtime operating systems use time quantum of 10 to 100 milliseconds, and context switching usually costs about 10 microseconds [Bel13]. Typical main memory reference costs about 100 nanoseconds [4], just a small portion of context switching costs, and even smaller portion of the time quantum. In case the application is not frequently interrupted, this means at most 0.1 to 1 thousandth of total allocated time will be wasted on cache miss.

Considering interruption in non-realtime operating systems or preemption in RTOS, a key concern is for how long can this application keep running without being interrupted or preempted. In practice, 6.74 cpb means the computation of 16 parallel block encryption or decryption could finish within 1 microsecond for most modern processors. Assuming an extreme case where the SM4 computation is interrupted or preempted once every microsecond. The computation then suffers cache miss upon resumption. Without cache miss the performance gain is 61% over 2x or 28% over 3x, then around 100 nanoseconds of penalty or 10% to 20% loss in worst case could still be offset by the gain. The actual situation could be much better due to: 1) interruption or preemption happen at a much lower frequency, far below 1MHz; 2) or they could run on another core; 3) interrupt routines finish their jobs pretty briefly and quickly without evicting L2 or L3.

To summarize, we suggest to enable the cache exploiting technique by default, unless found optimal the other way in some corner cases.

5 Conclusion

We developed these techniques while developing a software package for constant time implementation of the Chinese commercial ciphers. There are still ways for further improvements, for example to stitch SM4 encryption with GCM mode and to encrypt or decrypt many instead of minimal blocks in a function call. We plan further enhancements and to open-source the package for public merits.

Acknowledge

We thank our colleagues Na Xiao, Lei Zhang, Hualin Lu for careful reviews and helpful comments. We'd also like to mention the TikZ for Cryptographers project [Jea16] as we drew all four figures based on it.

[4]for example, check out `https://www.7-cpu.com/cpu/Apple_M1.html` or `http://norvig.com/21-days.html#answers`

References

[Arm21] Arm. Arm cortex-x1 core software optimization guide, issue 4.0. https://developer.arm.com/documentation/102174/latest, 2021. Accessed: 2022-07-15.

[Bel13] John Bell. Operating systems course notes: Chapter 6 CPU scheduling. https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/6_CPU_Scheduling.html, 2013. Accessed: 2022-08-25.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005. Accessed: 2022-07-13.

[Bie17] Ard Biesheuvel. Accelerated AES for arm64 linux kernel. https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/, 2017. Accessed: 2022-07-08.

[BLT16] Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. Comb to pipeline: Fast software encryption revisited. *IACR Cryptol. ePrint Arch.*, page 47, 2016.

[EDC10] Jeremy Erickson, Jintai Ding, and Chris Christensen. Algebraic cryptanalysis of SMS4: Gröbner basis attack and SAT attack compared. In Donghoon Lee and Seokhie Hong, editors, *ICISC 09*, volume 5984 of *LNCS*, pages 73–86. Springer, Heidelberg, December 2010.

[Hu22] Daniel Hu. SM4 optimization for ARM by HW instruction. https://github.com/openssl/openssl/pull/17455/commits/4db4629b35b49eefbb2cb85d873935407d83c58c, 2022. Accessed: 2022-07-08.

[Int22] Intel. Intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2022. Accessed: 2022-07-14.

[Jea16] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[Joh21] Dougall Johnson. Firestorm/IceStrom TBX (four register table, 16b). https://dougallj.github.io/applecpu/measurements/{firestorm,icestorm}/TBX_four_reg_table_16B.html, 2021. Accessed: 2022-07-15.

[Kiv20] Jussi Kivilinna. sm4-aesni-avx2-amd64.s. https://github.com/gpg/libgcrypt/commit/35a78eb248d6bacd2a58477a122a0020d796ce63, 2020. Accessed: 2022-07-08.

[KKE+21] Hyeokdong Kwon, Hyunjun Kim, Siwoo Eum, Minjoo Sim, Hyunji Kim, Wai-Kong Lee, Zhi Hu, and Hwajeong Seo. Optimized implementation of sm4 on avr microcontrollers, risc-v processors, and arm processors. Cryptology ePrint Archive, Paper 2021/667, 2021. https://eprint.iacr.org/2021/667.

[LJH+07] Fen Liu, Wen Ji, Lei Hu, Jintai Ding, Shuwang Lv, Andrei Pyshkin, and Ralf-Philipp Weinmann. Analysis of the SMS4 block cipher. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *ACISP 07*, volume 4586 of *LNCS*, pages 158–170. Springer, Heidelberg, July 2007.

[Saa19] Markku-Juhani O. Saarinen. sm4ni. https://github.com/mjosaarinen/sm4ni, 2019. Accessed: 2022-07-08.

[SCA12] SCA (State Cryptography Administration). GM/T 0002-2012: SM4 block cipher algorithm. Standards Press of China, 2012.

[Yan21] Paul Yang. Shangmi (SM) cipher suites for TLS 1.3. *RFC*, 8998:1–13, 2021.

A SM4 S-Box table and its algebraic expression rewriting

The complete SM4 S-Box is provided in Table 4.

Table 4: SM4 S-Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D6	90	E9	FE	CC	E1	3D	B7	16	B6	14	C2	28	FB	2C	05
1	2B	67	9A	76	2A	BE	04	C3	AA	44	13	26	49	86	06	99
2	9C	42	50	F4	91	EF	98	7A	33	54	0B	43	ED	CF	AC	62
3	E4	B3	1C	A9	C9	08	E8	95	80	DF	94	FA	75	8F	3F	A6
4	47	07	A7	FC	F3	73	17	BA	83	59	3C	19	E6	85	4F	A8
5	68	6B	81	B2	71	64	DA	8B	F8	EB	0F	4B	70	56	9D	35
6	1E	24	0E	5E	63	58	D1	A2	25	22	7C	3B	01	21	78	87
7	D4	00	46	57	9F	D3	27	52	4C	36	02	E7	A0	C4	C8	9E
8	EA	BF	8A	D2	40	C7	38	B5	A3	F7	F2	CE	F9	61	15	A1
9	E0	AE	5D	A4	9B	34	1A	55	AD	93	32	30	F5	8C	B1	E3
A	1D	F6	E2	2E	82	66	CA	60	C0	29	23	AB	0D	53	4E	6F
B	D5	DB	37	45	DE	FD	8E	2F	03	FF	6A	72	6D	6C	5B	51
C	8D	1B	AF	92	BB	DD	BC	7F	11	D9	5C	41	1F	10	5A	D8
D	0A	C1	31	88	A5	CD	7B	BD	2D	74	D0	12	B8	E5	B4	B0
E	89	69	97	4A	0C	96	77	7E	65	B9	F1	09	C5	6E	C6	**84**
F	18	F0	7D	EC	3A	DC	4D	20	79	EE	5F	3E	D7	CB	39	48

Numbers in hex. Looking up 0xEF is to find the intersection of row E and column F, which is 0x84.

Liu etc. originally provided the algebraic expression of SM4 S-Box in [LJH+07]. We first translated it to the notation we used in this paper, as (13).

$$S(X) = (X \cdot A_o \oplus C_o)^{-1} \cdot A_o \oplus C_o \tag{13}$$

In (13), $(\cdot)^{-1}$ is multiplicative inverse over F_S, A_o is 8×8 matrix and $A_o = A$, C_o is a row vector and $C_o = C^T = (1,1,0,0,1,0,1,1)$ (Check out (3) for A and C).

We then showed that (13) does not lead to S-Box values in accordance with conventional matrix multiplication rules. We simply tested:

$$S(0) \overset{?}{=} \mathit{Lookup}(0)$$

1) We have $\mathit{Lookup}(0) = $ 0xD6 by looking to the intersection of row 0 and column 0 of Table 4.

2) For $S(0)$, we should have $S(0) = C_o^{-1} \cdot A_o \oplus C_o$. According to [LJH+07], a row vector $V = (v_0, v_1, ..., v_7)$ encodes $\sum_{i=0}^{7} v_i x^i$. Therefore C_o encodes $1 + x + x^4 + x^6 + x^7$, and its inverse over F_S should be $C_o^{-1} = (1,1,0,0,1,0,0,1)$, thus $C_o^{-1} \cdot A_o = (1,0,0,0,0,0,1,0)$. Then $S(0) = (0,1,0,0,1,0,0,1) = $ 0x92.

Therefore $S(0) \neq \mathit{Lookup}(0)$. □

We are not the first to expose this issue, but probably the one to give the simplest correct form of the SM4 S-Box expression as in (1). [EDC10] rewrote (13) in matrix right-multiply instead of left-multiply, however, with two more parameters than our expression: two different matrixes and two different column vectors.

B Register allocation details and 4x interleaving codes

Pre-lookup. Among 32 NEON registers, V_{16} to V_{31} are allocated to hold S-Box, thus saved from Figure 3 and Figure 4. V_0 to V_{12} have been allocated (Figure 3). This leaves 3 NEON registers at our disposal ($V_{13} - V_{15}$). We still need to reserve one to hold a constant value needed to calculate indexing value for tbx instruction. This is V_{15} for CONST in below code block. We also allocated the remaining 2 (V_{13} and V_{14}) for performance reasons during the $\oplus$ evaluation, for example in the in-round step 1, calculating V_8 with 2 additional registers could take only 2 clock cycles instead of 3 by leveraging *multi-issuing*:

$V_{13} = V_1 \oplus V_2$ *in parallel with* $V_{14} = V_3 \oplus V_{12}$
$V_8 = V_{13} \oplus V_{14}$

Lookup. For the τ transformation of 4x interleaving, we could reuse V_0 to V_7 to calculate the new indexing value, as shown in below code block of macro definition in Go Assembly. We had renamed V_0 to V_3 as Z_0 to Z_3, V_4 to V_7 as Y_0 to Y_3, V_8 to V_{11} as U_0 to U_3. CONST is V_{15}. The τ transformation of 4x interleaving therefore needs 29 NEON registers including 16 holding the S-Box.

```
#define tableLookupX16() \
    \// 1st
    VSUB   CONST.B16, U0.B16, Z0.B16 \
    VTBL   U0.B16, [V16.B16, V17.B16, V18.B16, V19.B16], U0.B16 \
    VSUB   CONST.B16, U1.B16, Z1.B16 \
    VTBL   U1.B16, [V16.B16, V17.B16, V18.B16, V19.B16], U1.B16 \
    VSUB   CONST.B16, U2.B16, Y0.B16 \
    VTBL   U2.B16, [V16.B16, V17.B16, V18.B16, V19.B16], U2.B16 \
    VSUB   CONST.B16, U3.B16, Y1.B16 \
    VTBL   U3.B16, [V16.B16, V17.B16, V18.B16, V19.B16], U3.B16 \
    \// 2nd
    VSUB   CONST.B16, Z0.B16, Z2.B16 \
    WORD   $0x4E007000 | 0<<16 | 20<<5 | 0x08 \
    VSUB   CONST.B16, Z1.B16, Z3.B16 \
    WORD   $0x4E007000 | 1<<16 | 20<<5 | 0x09 \
    VSUB   CONST.B16, Y0.B16, Y2.B16 \
    WORD   $0x4E007000 | 4<<16 | 20<<5 | 0x0A \
    VSUB   CONST.B16, Y1.B16, Y3.B16 \
    WORD   $0x4E007000 | 5<<16 | 20<<5 | 0x0B \
    \// 3rd
    VSUB   CONST.B16, Z2.B16, Z0.B16 \
    WORD   $0x4E007000 | 2<<16 | 24<<5 | 0x08 \
    VSUB   CONST.B16, Z3.B16, Z1.B16 \
    WORD   $0x4E007000 | 3<<16 | 24<<5 | 0x09 \
    VSUB   CONST.B16, Y2.B16, Y0.B16 \
    WORD   $0x4E007000 | 6<<16 | 24<<5 | 0x0A \
    VSUB   CONST.B16, Y3.B16, Y1.B16 \
    WORD   $0x4E007000 | 7<<16 | 24<<5 | 0x0B \
    \// 4th
    WORD   $0x4E007000 | 0<<16 | 28<<5 | 0x08 \
    WORD   $0x4E007000 | 1<<16 | 28<<5 | 0x09 \
    WORD   $0x4E007000 | 4<<16 | 28<<5 | 0x0A \
    WORD   $0x4E007000 | 5<<16 | 28<<5 | 0x0B \
```

Go Assembly does not directly support the **tbx** instruction so we hard coded a word of 0x4E007000, followed by its parameters from left to right: the indexing register, the registers to lookup from, and the destination register. For example,

```
    WORD  $0x4E007000 | 0<<16 | 20<<5 | 0x08
```

means to lookup values of V_0 from V_{20-23}, and save results to V_8:

```
    VTBX  V0, [V20.B16, V21.B16, V22.B16, V23.B16], V8
```