An update on Keccak performance on ARMv7-M

Alexandre Adomnicăi

alexandre@adomnicai.me

Abstract. This note provides an update on Keccak performance on the ARMv7-M processors. Starting from the XKCP implementation, we have applied architecture-specific optimizations that have yielded a performance gain of up to 21% for the largest permutation instance.

Keywords: Keccak · SHA-3 · SHAKE · ARM

1 Motivation

Keccak is a multipurpose cryptographic primitive which is notably known for being the core of the SHA-3 and SHAKE standards [Dwo15]. While Keccak runs fast on high-end processors thanks to vectorization or dedicated instructions (e.g. ARMv8 SHA3 extension), its largest version does not show outstanding performance on constrained platforms (e.g. microcontrollers) when fully implemented in software. This is mainly due to the lack of general purpose registers to hold the entire 1600-bit internal state, leading to high register pressure and therefore register spilling (i.e. saving and restoring some intermediate variables to and from memory). By way of illustration, on ARM Cortex-M4 processors, the current fastest Keccak implementation spends around half of the running time in memory accesses. On top of being a cryptographic primitive itself, Keccak is also used in other cryptographic algorithms internally, notably many PQC schemes use it for various purposes, from seed expansion to CPA-to-CCA transforms. For instance, the PQC algorithms Dilithium [DKL+18] and Kyber [BDK+18] selected by NIST for standardization rely on Keccak to such an extent that hashing dominates the overall performance up to 85% on Cortex-M4 [GKS20]. In order to reduce this overload, the Keccak designers suggested to use a 12-round variant named TurboSHAKE [BDH+23] instead of the 24-round variant, but NIST stated not being in favor of such a decision[1]. This highlights the needs for improvement on architectures where Keccak does not show outstanding performance.

Our contribution This note details how we managed to improve Keccak performance on ARMv7-M by up to 21% thanks to two kind of optimizations. The first one consists in taking advantage of the inline barrel shifter in order to get rid of explicit rotations in the linear layer. Note that this technique has been already reported in the litterature by Becker and Kannwischer in order to boost Keccak on ARMv8 architectures [BK22] but has not been ported to ARMv7-M so far. The second optimization consists in a more efficient memory access scheduling to avoid pipeline hazards. We provide benchmarks for all the SHA-3, SHAKE and TurboSHAKE instances.

[1] https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/5HveEPBsbxY

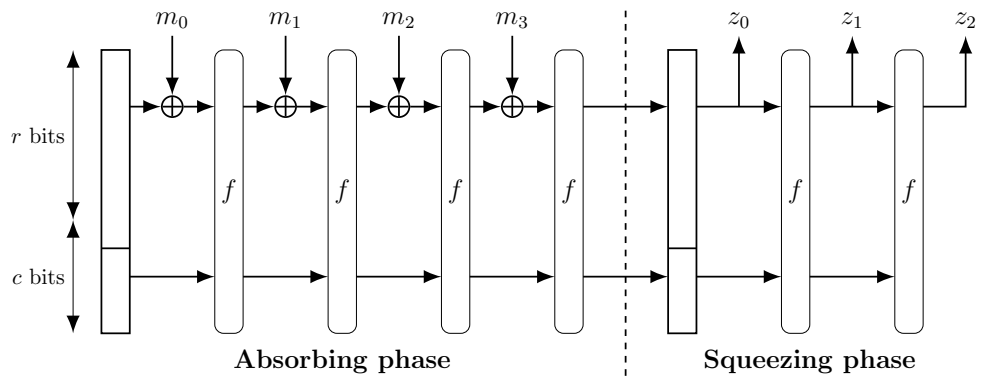

Figure 1: The Sponge construction. m_i refers to the i-th message block after padding, z_i refers to the i-th hash block (before truncation) and r, c refer to the rate and capacity in bits, respectively. Taken from [Jea16].

2 Background

2.1 Keccak

Keccak is built upon the sponge construction, depicted in Figure 1, together with a cryptographic permutation Keccak-p$[b, n_r]$ where b and n_r refer to the bitwidth of the permutation and the number of rounds, respectively. In this note, we focus exclusively on instances where $b = 1600$, such as the variant Keccak-p[1600,24] used in NIST standards. The state A is represented as an array of 5×5 lanes, each composed of $w = b/25$ bits. $A[x, y]$ refers to the lane at position (x, y) and $A[x, y, z]$ refers to the z-th bit of the lane. Keccak-p is an iterated permutation where each round consists of five consecutive operations θ, ρ, π, χ and ι, where χ is the only non-linear operation. See the Listing 1 for a brief description of Keccak-p using pseudo-code.

```
# b refers to the permutation width while nr refers to the number of rounds
keccak-p[b,nr](A):
  A = roundperm(A,RC[i])                                    for i in 0..nr-1
  return A

# r[x,y] refer to rotation offsets while RC refers to the round constant
roundperm(A,RC):
  # theta step
  C[x]   = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]  for x in 0..4
  D[x]   = C[x-1] xor rot(C[x+1],1)                            for x in 0..4
  A[x,y] = A[x,y] xor D[x]                              for (x,y) in (0..4,0..4)
  # rho and pi step
  B[y,2*x+3*y] = rot(A[x,y], r[x,y])                    for (x,y) in (0..4,0..4)
  # chi step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]) for (x,y) in (0..4,0..4)
  # iota step
  A[0,0] = A[0,0] xor RC
  return A
```

Listing 1: Pseudo-code of the Keccak-p cryptographic permutation.

2.2 ARMv7-M processors

ARMv7-M refers to the microcontroller profile of the ARMv7 architecture. It comes with sixteen 32-bit registers (`r0-r15`), out of which one is used as stack pointer (`r13`), one is used as link register (`r14`), and one for the program counter (`r15`). It supports the Thumb-2 technology, which means that 16-bit and 32-bit encoding of instructions can be freely mixed. Thanks to the barrel shifter, flexible second operands can be shifted or rotated as part of the instruction without any impact on performance (unless the amount to be shifted or rotated is specified by a register, in which case the instruction will take an extra cycle to complete). In this note, we consider the ARM Cortex-M3, M4 and M7 processors. All of them are based on the ARMv7-M architecture, but the M4 and M7 also support additional instructions that are part of the DSP extension (i.e. the ARMv7E-M architecture). However, we do not use this extension for our Keccak implementations.

Cortex-M3 and M4 The M3 and M4 processors have a 3-stage pipeline. Most instructions take a single cycle, except branches and memory accesses that may take more cycles. While load (`ldr`) and store (`str`) instructions typically require 2 and 1 cycles, respectively, `ldr` instructions can take up to 3 cycles to complete in case of dependency with the previous instruction. In the absence of such dependency, n `ldr` can be pipelined together to be executed in $n+1$ cycles, and `str` following `ldr` takes 0 cycle. Use of the inline barrel shifter has no impact on performance, unless the amount to be shifted or rotated is specified by a register, in which case the instruction will take an extra cycle to complete.

Cortex-M7 The M7 is a more powerful processor which embeds a 6-stage pipeline with dual-instruction issue, enabling it to execute up to two instructions per clock cycle. It also comes with a branch predictor in addition to instruction and data caches whose sizes may vary from 4 to 64 KiB. Unfortunately, unlike the M3 and M4 cores, ARM did not make information about instructions timing on the M7 publicly available. This makes writing optimized assembly code very difficult given the complexity of the core. Thankfully, some independent analyses have been conducted by individuals in an attempt to address this lack of information [Owe22, jnk].

2.3 Keccak-p on ARMv7-M

2.3.1 Optimization techniques

The designers of Keccak have summarized many possible optimizations for software and hardware implementations in a dedicated document [BDH+12]. The two most useful ones on ARMv7-M are described below.

Bit-interleaving The intuitive way to implement Keccak-p in software is to follow a lane-wise architecture (i.e. a register contains one or multiple lanes), as described in Listing 1. On platforms where registers are not large enough to handle an entire lane, the designers recommend to use the bit-interleaving technique. For $b = 1600$ on 32-bit architectures, it consists in storing bits at odd positions in one register, and bits at even positions in another register. This way, 64-bit rotations can be easily handled by seperate 32-bit rotations without additional operations. Note that this requires some extra calculations for rearranging the lanes at the beginning and at the end of the permutation. However it is calculated at a higher level (i.e. when adding and extracting data to and from the state), and therefore it is not taken into account when benchmarking the permutation itself.

Efficient in-place processing When updating the internal state during the round function, it is possible to store all processed data back into the same memory location it was loaded

from. This way only a single instance of the state (instead of two) must be preserved. Because the π operation moves lanes within the state, it requires to define a mapping between the lane coordinates and the memory location depending on the round number. The Keccak designers propose a linear map using a matrix of order 4, which means the state will return to its initial memory location after 4 rounds.

2.3.2 Performance

On 32-bit architectures, each round of Keccak-p$[1600, \cdot]$ consists of 152 XORs, 50 ANDs, 50 NOTs and 58 rotations thanks to the bit-interleaving technique. On ARM, it is possible to merge 1 AND and 1 NOT into a single `bic` instruction and the rotations during the θ step can be easily combined with an XOR thanks to the inline barrel shifter. This leads to 250 instructions per round overall: 152 `eor`, 50 `bic` and 48 `ror`. Assuming logical instructions take 1 cycle, the raw cost of Keccak-p$[1600, 24]$ on this platform should theoretically be $250 \times 24 = 6\,000$ clock cycles. However, given the fact that the state is 1600-bit long and that ARMv7-M only offers 14 32-bit general purpose registers to work with, it implies that performance will inevitably bear the cost of many loads and stores on the stack. As reference, we consider the ARMv7-M implementation from the eXtended Keccak Code Package (XKCP) [BDH+]. Because it contains optimized, free and open-source implementations for various platforms and architectures, it is often used as a third-party software component. According to previous research, the assembly implementation of Keccak-p$[1600, 24]$ from XKCP requires 12 969 clock cycles on the Cortex-M4 [Sto19], meaning that around 54% of the cycles are spent in memory accesses. However, there is still room for improvement as explained in the next section.

3 Architecture specific optimizations

3.1 Pipelining memory accesses

The ARMv7-M assembly implementation provided by XKCP at the time of writing[2] works as follows. At the beginning of each round, all the parity lanes (namely `D[x]` on line 10 of Listing 1) are precomputed. To compute half a parity lane, the code relies on a macro `xor5` which consists of 5 `ldr` and 4 exlcusive-OR (`eor`) instructions, as detailed in Listing 2.

```
1  .macro    xor5        result,b,g,k,m,s
2      ldr         \result, [r0, #\b]
3      ldr         r1, [r0, #\g]
4      eors        \result, \result, r1
5      ldr         r1, [r0, #\k]
6      eors        \result, \result, r1
7      ldr         r1, [r0, #\m]
8      eors        \result, \result, r1
9      ldr         r1, [r0, #\s]
10     eors        \result, \result, r1
11 .endm
```

Listing 2: ARMv7-M assembly code to compute half a parity lane. Note that memory accesses are interleaved with logic operations.

Once this precomputing phase is complete, it executes the θ, ρ, π, χ and ι steps by group of 5 half lanes at a time, as detailed in Listing 3. Note that round constants for the ι step are precomputed (i.e. hardcoded in ROM) instead of being computed on-the-fly.

[2] https://github.com/XKCP/XKCP/commit/7fa59c0ec4b5802b7c269ddd9ef0ef35999b4f0f

```
1  .macro    KeccakThetaRhoPiChi   aB1, aA1, aDax, rot1, \
2                                  aB2, aA2, aDex, rot2, \
3                                  aB3, aA3, aDix, rot3, \
4                                  aB4, aA4, aDox, rot4, \
5                                  aB5, aA5, aDux, rot5
6      ldr     \aB1, [r0, #\aA1]
7      ldr     \aB2, [r0, #\aA2]
8      ldr     \aB3, [r0, #\aA3]
9      ldr     \aB4, [r0, #\aA4]
10     ldr     \aB5, [r0, #\aA5]
11     eors    \aB1, \aB1, \aDax
12     eors    \aB3, \aB3, \aDix
13     eors    \aB2, \aB2, \aDex
14     eors    \aB4, \aB4, \aDox
15     eors    \aB5, \aB5, \aDux
16     rors    \aB1, #32-\rot1
17 .if  \rot2 > 0
18     rors    \aB2, #32-\rot2
19 .endif
20     rors    \aB3, #32-\rot3
21     rors    \aB4, #32-\rot4
22     rors    \aB5, #32-\rot5
23     xandnot \aA1, r3, r4, r5
24     xandnot \aA2, r4, r5, r6
25     xandnot \aA3, r5, r6, r7
26     xandnot \aA4, r6, r7, r3
27     xandnot \aA5, r7, r3, r4
28 .endm
```

Listing 3: ARMv7-M assembly implementation of the Keccak round function w/o the ι step. Note that memory accesses are grouped together.

3.1.1 Cortex-M3 and M4

On the Cortex-M3 and M4, it is clear that the `xor5` macro suffers pipeline stalls since only 2 out of the 5 `ldr` instructions are consecutive. While grouping all these loads together would allow to save 3 cycles per macro call, this requires to change the way variables are assigned to registers since `r1` is used as the only destination of `ldr` instructions in order to preserve the other registers. Nevertheless, we managed to relax the register pressure thanks to another register allocation, allowing us to pipeline all the 5 `ldr` together within the `xor5` macro. To further improve memory access pipelining, we also reordered some other instructions. Notably, we moved `str` instructions after multiple `ldr` as much as possible.

3.1.2 Cortex-M7

Due to discrepancies in the pipeline architecture, our optimizations discussed above are not relevant on the Cortex-M7. While the M3 and M4 benefit from grouping all memory accesses together, the M7 benefits from interleaving memory accesses with arithmetic or logic operations to maximize dual-issue pipeline capabilities [Lor16]. Indeed, according to independent benchmarks, loads from the same memory interface cannot be dual issued [Owe22, jnk]. Therefore, to avoid pipeline stalls due to consecutive `ldr` instructions at the beginning of the `KeccakThetaRhoPiChi` macro (i.e. lines 6 to 10 of Listing 3), we interleaved them with the subsequent logic operations in a similar way to which it is done in Listing 2.

3.2 Lazy rotations

The XKCP implementation makes use of explicit rotations for the ρ step through `ror` instructions. While it makes the code easy to follow, it requires 47 such instructions per round. As recently proposed by Becker and Kannwischer on AArch64 [BK22], one can omit those explicit rotations by means of *lazy rotations* (i.e. rotating the second operands thanks to the inline barrel shifter) during subsequent operations. They recommend to defer the explicit rotations until the θ step in the next round: once all the (unrotated) parity lanes have been calculated, then they are rotated explicitly. By proceeding this way, the (unrotated) state lanes are lazily rotated when treated as second operands during the XOR with the (rotated) parity lanes, so that the internal state is back to the classical representation and the process can be reiterated thereafter. While it would be also possible to keep deferring rotations, it would require to fully unroll the permutation code, resulting in substantial impacts on code size.

On AArch64, it leads to 3 explicit rotations instead of 5 since 2 deferred rotation values are in fact 0. While it should theoretically result to 6 explicit rotations on ARMv7-M because of its 32-bit architecture, it is possible to stick to 3 rotations overall as described below. Thanks to the bit-interleaving technique, computing a 1-bit rotation on a 64-bit word can be achieved using a single 32-bit rotation only. Therefore, when computing the parity lanes, only 5 `eor` instructions need to rotate their second operand, leaving the barrel shifter available for deferred rotations during the remaining 5 `eor`. In the end, only 3 explicit rotations remain per round instead of 47.

We implemented this technique on ARMv7-M along with the in-place processing optimization in two different ways. While the most efficient approach is to use lazy rotations for all rounds, it requires to have specific routines for the first and last rounds to deal with input and output misalignements since the internal state is expected to be properly aligned at function entry and exit. When considering in-place processing, and therefore a quadruple round routine, it results in a code size increase by half since the first and last rounds should both come in two variants. In order to boost the performance while limiting the impact on code size, we also propose a variant where lazy rotations are applied for three-quarters of the rounds only. This way, explicit rotations are used every 4 rounds to ensure the internal state is correctly aligned when entering the quadruple round. Still, a potential drawback of deferring rotations is that it affects the code readability, which may make the integration of side-channel countermeasures (e.g. masking) more cumbersome.

4 Results summary

4.1 Benchmark settings

For our benchmarks, we used the three development boards described below. All implementations were compiled using `arm-none-eabi-gcc 10.3.1` along with the `-O3` optimization flag, and clock cycles were measured using the `DWT_CYCCNT` cycle counter register. These results were obtained by simply measuring the execution of a single function call. Our implementations are publicly available at https://github.com/aadomn/keccak_armv7m.

STM32L100C. It features a Cortex-M3 running up to 32 MHz, 256 KB of flash memory and 16 KB of RAM. The core was clocked at 16 MHz to execute code from flash with zero-wait state.

STM32F407VG. It features a Cortex-M4 running up to 168 MHz, 1024 KB of flash memory and 192 KB of RAM divided into three blocks: two contiguous blocks of SRAM

Table 1: Keccak-p[1600, 24] benchmark on ARMv7-M processors.

Ref.	Implementation characteristics*			Speed (clock cycles)			Code size	RAM
	`ldr`/`str`	lazy `ror`	in-place	M3	M4	M7	(bytes)	(bytes)
XKCP[3]	mostly grouped	✗	✓	11 707	11 749	6 162	5 576	**264**
Ours[4]	interleaved	✗	✗	12 409	12 445	**5 578**	**2 904**	464
	grouped	✗	✓	10 339	10 375	6 571	5 772	**264**
	grouped	✓ (3/4)	✓	9 403	9 439	6 694	6 556	**264**
	grouped	✓ (4/4)	✓	**9 206**	**9 242**	6 725	9 536	**264**

*All listed implementations take advantage of the bit-interleaving technique.

connected to the bus matrix with different interconnects, and a core coupled memory (CCM) block which is connected directly to the core. We exclusively used the 64 KB CCM to achieve the best performance with a clock core speed of 24 MHz to execute code from flash with zero-wait state.

STM32F746NG. It features a Cortex-M7 running up to 216 MHz with 4 KB data and 4 KB instruction caches, 1024 KB of flash memory and 320 KB of RAM divided into three blocks: two contiguous blocks of SRAM connected to the bus matrix with different interconnects, and a data tightly-coupled memory (DTCM) block which is connected directly to the core. To achieve the best performance, we exclusively used the 64 KB DTCM and accessed the flash memory via the ITCM bus with a core clock speed of 24 MHz for zero-wait state accesses.

4.2 Keccak permutation

Table 1 provides a benchmark of Keccak-p[1600, 24] for different implementation characteristics. The first one, denoted by `ldr/str`, refers to the memory access pipelining strategy. In this context, the term "mostly grouped" signifies that while the majority of memory accesses are packed together, this is not the case during the parity lanes precomputation. The other two characteristics indicate, respectively, the utilization and degree of lazy rotations, as well as the implementation of in-place processing. In our scenario, the performance impact of in-place processing is negligible. Therefore, the primary reason for deactivating it is to emphasize its influence on memory usage. It is important to note that the situation might vary when accessing the flash memory through the AXI master interface, as it can trigger the operation of the instruction cache, if present. In this case, if the instruction cache is limited to only 4 KB, optimizing the code size to ensure it fits entirely within the cache can lead to improved performance.

4.2.1 Cortex-M3 and M4

On the M3 and M4, the pipeline optimizations introduced thanks to our new register allocation resulted in a performance boost of 11.6 %. Note that it comes at the cost of a slight code size increase due to the fact that some `ldr` instructions are now using hi registers (i.e. `r8` to `r14`) as destination, leading to a 32-bit encoding (versus 16-bit when using lo registers). When combined with the use of lazy rotations, we managed to achieve up to a 21.4 % performance boost. Nevertheless, the majority of the remaining instructions that were initially encoded on 16 bits have now been 32-bit encoded. This is because the

[3] https://github.com/XKCP/XKCP/commit/7fa59c0ec4b5802b7c269ddd9ef0ef35999b4f0f
[4] https://github.com/aadomn/keccak_armv7m/commit/0bf17a342569774c844f3151b9b0dfc0f0b20324

Table 2: SHA-3 and SHAKE benchmark on ARMv7-M processors.

Algorithm	r	**Speed** (cycles per byte)		
		M3	M4	M7
SHA3-224	1 152	72.1	72.4	46.4
SHA3-256	1 088	76.7	77	49.1
SHA3-384	832	97.5	97.9	61.5
SHA3-512	576	138	138.4	85.5
SHAKE128	1 344	62.1	62.3	39.3
SHAKE256	1 088	75.9	76.2	47.5
TurboSHAKE128	1 344	34.1	34.2	22.7
TurboSHAKE256	1 088	41.1	41.2	26.8

second operands undergo systematic rotation using the inline barrel shifter. As discussed in Section 3.2, the variant that utilizes lazy rotations for all rounds incurs a 50% increase in code size for a minor performance gain of less than 3%. While we could have potentially made it more compact, we believe that doing so would have compromised the performance advantage, resulting in no overall benefits compared to the other variant. Therefore, when code size is a critical factor, we suggest to favor the implementation that uses lazy rotations for three-quarters of the rounds only in order to minimize the memory footprint.

4.2.2 Cortex-M7

On the Cortex-M7, our pipeline optimizations did not only consist in interleaving memory accesses with arithmetic/logic operations. Another optimization that was taken into consideration was to ensure that rotations are not manipulating the result of the previous instruction to avoid one-cycle delays [Owe22]. All in all, it allowed us to achieve a 10% performance boost compared to the reference implementation. However, in contrast to the M3 and M4, the use of lazy rotations does not improve performance on this platform but makes it even worse. This can be explained by the fact that, on this platform, instructions with shifted/rotated second operand cannot be dual issued [Owe22, jnk]. Therefore, we did not consider an implementation that combines lazy rotations with our pipeline optimizations on this platform as we do not expect any benefits from it.

4.3 SHA-3 and SHAKE

Table 2 provides benchmark of the Keccak-based functions defined in the FIPS 202 standard, along with the 12-round variants TurboSHAKE128 and TurboSHAKE256. For each processor, we considered the fastest variant of Keccak-p[1600] reported in Table 1. Results were obtained when computing a digest on 8 192 bytes for the hash functions and expanding a 32-byte seed into 8 192 bytes for the extandable-output functions. Performance are expressed in cycles per byte. The careful reader will note a slight performance overhead compared to raw Keccak-p[1600]. This is due to the fact that the r-bit input/output blocks are mapped to/from the bit-interleaving representation when being absorbed/squeezed by the state, outside the permutation itself. This extra cost could be discarded when strict compliance with the standard is not needed, without affecting security.

5 Conclusion

In this note we explained how we managed to enhance Keccak-p[1600] performance up to 21% on ARMv7-M processors. Optimizing this permutation not only brings inherent advantages but also holds substantial value for other cryptographic primitives, such as forthcoming NIST PQC standards. Although we believe that there is limited scope for further enhancements on the Cortex-M3 and M4, there is a possibility that we missed some pipeline optimizations on the Cortex-M7 core due to its more complex architecture and the lack of timing information.

References

[BDH+] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. XKCP: eXtended Keccak Code Package. https://github.com/XKCP/XKCP. commit 7fa59c0.

[BDH+12] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf, 2012. Accessed: 2023-05-26.

[BDH+23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. TurboSHAKE. Cryptology ePrint Archive, Paper 2023/342, 2023. https://eprint.iacr.org/2023/342.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM. In *EuroS&P*, pages 353–367. IEEE, 2018.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 272–293, Cham, 2022. Springer International Publishing. https://eprint.iacr.org/2022/1243.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, Feb. 2018.

[Dwo15] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015-08-04 2015.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, Dec. 2020.

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

[jnk] jnk0le. random. https://github.com/jnk0le/random/tree/master/pipeline%20cycle%20test#cortex-m7. commit 725a014.

[Lor16] Thomas Lorenser. The DSP capabilities of ARM Cortex-M4 and Cortex-M7 Processors. https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf, 2016. Accessed: 2023-05-26.

[Owe22] Mark Owen. Cortex-M7 instruction cycle counts, timings, and dual-issue combinations. https://www.quinapalus.com/cm7cycles.html, 2022. Accessed: 2023-05-26.

[Sto19] Ko Stoffelen. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology – LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2–4, 2019, Proceedings*, page 323–340, Berlin, Heidelberg, 2019. Springer-Verlag.