1001 Ways To Implement Keccak

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2],
Gilles Van Assche[1], and Ronny Van Keer[1]

[1] STMicroelectronics
[2] NXP Semiconductors

The structure of Keccak allows a fair amount of diversity in the way it can be implemented. However, it is often not trivial to select the optimal options in given circumstances, and sometimes one may even not be aware of all of Keccak's implementation techniques. We here briefly present different techniques, referring to external documents for the details.

Keccak defines a family of sponge functions with seven different permutations and most implementation techniques work on all of them. For clarity, however, we take as example Keccak-$f[1600]$, the permutation used in the proposed SHA-3 hash function candidates [3].

1 How to cut a state

A hardware circuit can be built to compute the round function within a clock cycle, hence processing the state as a whole. This is what several hardware implementations do, including our high-speed core [6, Section 4.2]. However, this simple solution is not possible in software on standard CPUs and is not always affordable in hardware implementations. The state of Keccak-$f[1600]$ is organized as a three-dimensional array, which suggests several ways to partition the bits. So we here describe several ways to serialize the round function that exploit this structure. We assume that the reader is familiar with the naming conventions (column, lane, plane, slice) depicted in Figure 1.

How to cut a lane: bit interleaving The bit interleaving technique is fairly general, can be combined with most other ideas, applies to both software and hardware, and provides an area-speed trade-off in some classes of implementations.

The state of Keccak-$f[1600]$ can be expressed as 25 lanes of 64 bits each. In software, this calls for an implementation using 64-bit words. While this is an optimal choice on software platforms actually offering 64-bit operations, the bit interleaving technique allows efficient implementations on systems with smaller word sizes and can also be used to target compact hardware circuits.

In its simplest form, namely factor-2 interleaving, it splits the odd and even bits of each lane. The state of Keccak-$f[1600]$ is then represented as 50 words of 32 bits. The rotations in θ and ρ are performed as cyclic shifts on 32-bit words, making them efficient on a 32-bit processor. There is a cost associated to the conversion of the input message into this representation, but this cost remains small compared to the evaluation of the permutation itself. Note that the use of modular addition would have prevented the bit interleaving technique.

In general an interleaving factor of s maps each lane to s words of $\frac{64}{s}$ bits. For instance, factor-8 interleaving expresses the round function of Keccak-$f[1600]$ in terms of operations on bytes. Further details and examples can be found in [6, Section 2.1].

Processing planes A plane is a set of 5 lanes that can be combined in χ. So doing plane-per-plane processing nicely fits in χ. The dispersion step π just before χ can be implemented implicitly by fetching the lanes from appropriate locations, and the rotations ρ can be done individually on each lane together with π. The step θ can be done on the fly (see Section 3). Detailed scheduling of the operations can be found in [6, Section 2.4].

Bit interleaving can also be used to process fractions of planes. For an interleaving factor of s, 5 words of $64/s$ bits are processed together. Currently, the fastest software implementations are organized to process each plane at a time. This includes both implementations optimized for 64-bit platforms ($s = 1$, no interleaving) and those for 32-bit ones ($s = 2$).

Processing consecutive slices While the state of Keccak-$f[1600]$ can be seen as an array of 25 lanes of 64 bits, the transposed view is to see it as an array of 64 slices of 25 bits each. The function of ρ is to disperse bits across different slices, but all the other operations work in a slice-oriented way. More precisely, π and χ work in each slice independently, and for θ the output slice z depends on the input slices $z-1$ and z. Hence, a way to cut the state is to make groups of n slices with consecutive z coordinates, where n divides 64.

The idea of processing consecutive slices in a hardware circuit comes from Jungk and Apfelbeck [7]. They implemented Keccak-$f[1600]$ using $n = 8$ consecutive slices, with extra registers to manage the fact that θ makes the last slice of a group interact with the first slice of the next group. Inter-slice dispersion ρ is implemented in part by an appropriate addressing of RAM and in part by extra registers. This resulted in a compact implementation on a Virtex-5 FPGA using 393 slices with a throughput of 864 MBit/sec for a rate of $r = 1088$ [7].

The number n of consecutive slices can serve as a parameter for speed-area trade-offs. We have explored the effects on area and throughput when n takes the values 2, 4, 8, 16 and 32. The round function is computed in $\frac{64}{n} + 1$ clock cycles, while the complete permutation requires $25 * \frac{64}{n} + 24$ clock cycles. The most convenient configurations, in term of area-throughput ratio, are obtained with n equal to 16 or 32. Implementations in VHDL are available in [5].

Processing interleaved slices An alternative to the idea of consecutive slices is to group slices in an interleaved way. For a given interleaving factor s, the slices are grouped with constant coordinate z modulo s. Here again π and χ work in each slice independently. However, compared to the case of consecutive slices, ρ can be partly done within a group of slices. And if some extra memory is dedicated to the parity (see Section 3), θ can be implemented without interdependencies between groups of slices.

This idea allows similar speed-area trade-offs in hardware implementations, as the number of slices to be grouped can be chosen. However, an actual implementation is needed to determine which one of consecutive or interleaved slices works best on which platforms.

2 Minimizing the memory footprint

In terms of memory usage, Keccak has no feedforward loop and the state can serve as a message queue without the need for additional memory dedicated specifically to that purpose. Hence, the memory footprint of Keccak is determined solely by that of the underlying permutation. We here describe a technique to minimize the memory needed by Keccak-$f[1600]$ without sacrificing speed.

Efficient in-place processing A typical speed-optimized software implementation reserves two memory areas, each with the size of the state (200 bytes). The computation of a round takes the state in one area and stores the result in the other, alternatively. In [6, Section 2.5], we propose a way to store processed data back into the same memory location it was taken from. Hence a single instance of the state must be reserved, plus some extra memory to store the parity and/or the θ-effect (see Section 3). As π moves lanes to different coordinates, this requires to define a mapping between the lane coordinates (x, y) and the memory location that depends on the round number. The mapping has a cycle of 4 rounds, so after the 24 nominal rounds the memory area returns to its original configuration.

This technique can be combined with bit interleaving. In that case, the mapping between the lane coordinates and memory location must be adapted. E.g., with factor-2 interleaving the mapping still has a cycle of 4 rounds. For instance, the currently fastest implementation on the 32-bit processor ARM

Cortex-M3 makes use of the in-place processing with 4 rounds unrolled and requires only 272 bytes on the stack [6, Section 3.2.1].

Detailed algorithms using efficient in-place processing, with and without bit interleaving, can be found in [6, Section 2.5].

3 Additional techniques

In this section, we give optimization techniques for the evaluation of χ and of θ.

Lane complementing The mapping χ of Keccak-f[1600] consists in 5 XOR, 5 AND and 5 NOT operations. Some platforms support instructions that combine a AND and a NOT, but not all do. In the latter case, the lane complementing technique aims at removing 4 out of 5 NOT operations by representing some of the lanes by their complement. This makes simple use of the De Morgan laws, replacing some logical ANDs by ORs. We explain how this can be done in [6, Section 2.2].

Extending the state to compute θ on the fly The θ operation consists in XORing a pattern in the entire state that depends only on the parity of the columns before θ. The pattern to XOR is called the θ-effect and is constant over each column. If the implementation can afford some extra memory, one can use 5 lanes:

– to accumulate the parity of the columns as the output of χ in the previous round is being computed, and/or
– to store the θ-effect to be able to XOR it as the current round is being processed.

Further details and examples can be found in [6, Section 2.3] and in [6, Section 2.4.1].

4 Protecting against side-channel attacks

When the input of Keccak contains a secret key, e.g., to compute a MAC or to do (authenticated) encryption, protection against side-channel attacks may be appropriate. Regarding timing attacks, all the implementations described here make use of a fixed sequence of operations without the need for look-up tables. To help protect against differential power analysis and its variants, we provide techniques in [6, Chapter 5] and in [2].

Software: two-share masking To decorrelate the data being processed to the native value of the state, a simple technique consists in working on two randomized shares a and b chosen such that their XOR $a \oplus b$ is the native value. The linear operations of the round function θ, ρ and π can be performed on each share independently. Details on how to do implement χ with little overhead can be found in [6, Section 5.3].

Hardware: three-share masking In hardware, an additional source of side-channel information for an attacker is the presence of glitches. In this case, we turn ourselves to three-share implementations, i.e., with three randomized shares a, b and c chosen such that their XOR $a \oplus b \oplus c$ is the native value [8]. The operations on the shares are such that any logical block never processes more than two shares together. Hence, glitches cannot be correlated to the native value. More details can be found in [6, Section 5.4].

5 Taking best advantage of high-end platforms

In the evaluation of hardware performances, a popular measure is the throughput-to-area ratio. The reason is that it says more about efficiency than speed or area alone.

In software, high-end platforms provide an ever increasing computational bandwidth, not only by embedding more cores, but also by widening single-instruction multiple-data (SIMD) registers and associated operations. For instance, the new AVX and upcoming AVX2 instruction sets from Intel provide 256-bit SIMD operations, in addition to the 128-bit ones from the previous architectures. A software equivalent to the throughput-to-area ratio would be the *throughput-per-core* ratio or equivalently the number of cycles per byte per core. Faster hash functions exploiting this parallelism can be built in a scalable way using tree hashing techniques and in particular with leaf interleaving [1].

Throughput per core on Sandy Bridge Processors based on the Sandy Bridge architecture from Intel provide a fast processing of 128-bit SIMD instructions. Evaluating two instances of KECCAK-$f[1600]$ in parallel using these instructions takes about the same number of cycles as evaluating a single instance using regular 64-bit instructions. Hence, a tree hashing mode with two parallel branches on top of KECCAK can provide a hash function provably as secure as KECCAK itself [1], only twice as fast. We measured the two parallel evaluations at 1660 cycles, hence providing an estimated 6.5 cycles/byte/core with default $r = 1024$ and leaf interleaving $(G = \text{LI}, H = 1, D = 2, B = 64, C = c = 576)$ [1].

Extending the instruction set We can imagine several ways to extend future processors with instructions enabling faster implementations of KECCAK. A very simple way seems to be the introduction of cyclic-shift instructions. The AVX and SSE instruction sets provide shifts over 64-bit words but not cyclic shifts. Hence, in the exercise above we implemented the rotations of θ and ρ using two shifts and a bitwise OR. With a cyclic shift instruction, we estimate[1] the two parallel evaluations would take about 1150 cycles, or 4.5 cycles/byte/core with default $r = 1024$. And further assuming that the future processors are as fast with 256-bit SIMD as with 128-bit SIMD, four parallel evaluations would yield about 2.25 cycles/byte/core.

6 More information

For more information, the central documentation on implementation techniques is [6]. For software implementations, one can find many examples of optimized code in [5]. In addition, KECCAKTOOLS [4] provides functions to generate optimized code based on most of the techniques mentioned here.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210, 2009, `http://eprint.iacr.org/`.
2. ———, *Building power analysis resistant implementations of* KECCAK, Second SHA-3 candidate conference, August 2010.
3. ———, *The* KECCAK *SHA-3 submission*, January 2011, `http://keccak.noekeon.org/`.
4. ———, KECCAKTOOLS *software*, April 2012, `http://keccak.noekeon.org/`.
5. ———, *Reference and optimized implementations of* KECCAK, 2012, `http://keccak.noekeon.org/`.
6. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, KECCAK *implementation overview*, May 2012, `http://keccak.noekeon.org/`.
7. B. Jungk and J. Apfelbeck, *Area-efficient FPGA implementations of the SHA-3 finalists*, International Conference on ReConFigurable Computing and FPGAs (ReConfig), 2011, to appear.
8. S. Nikova, V. Rijmen, and M. Schläffer, *Secure hardware implementation of nonlinear functions in the presence of glitches*, ICISC (P. J. Lee and J. H. Cheon, eds.), Lecture Notes in Computer Science, vol. 5461, Springer, 2008, pp. 218–234.

[1] This estimation was done by replacing the rotations of θ and ρ by shifts.

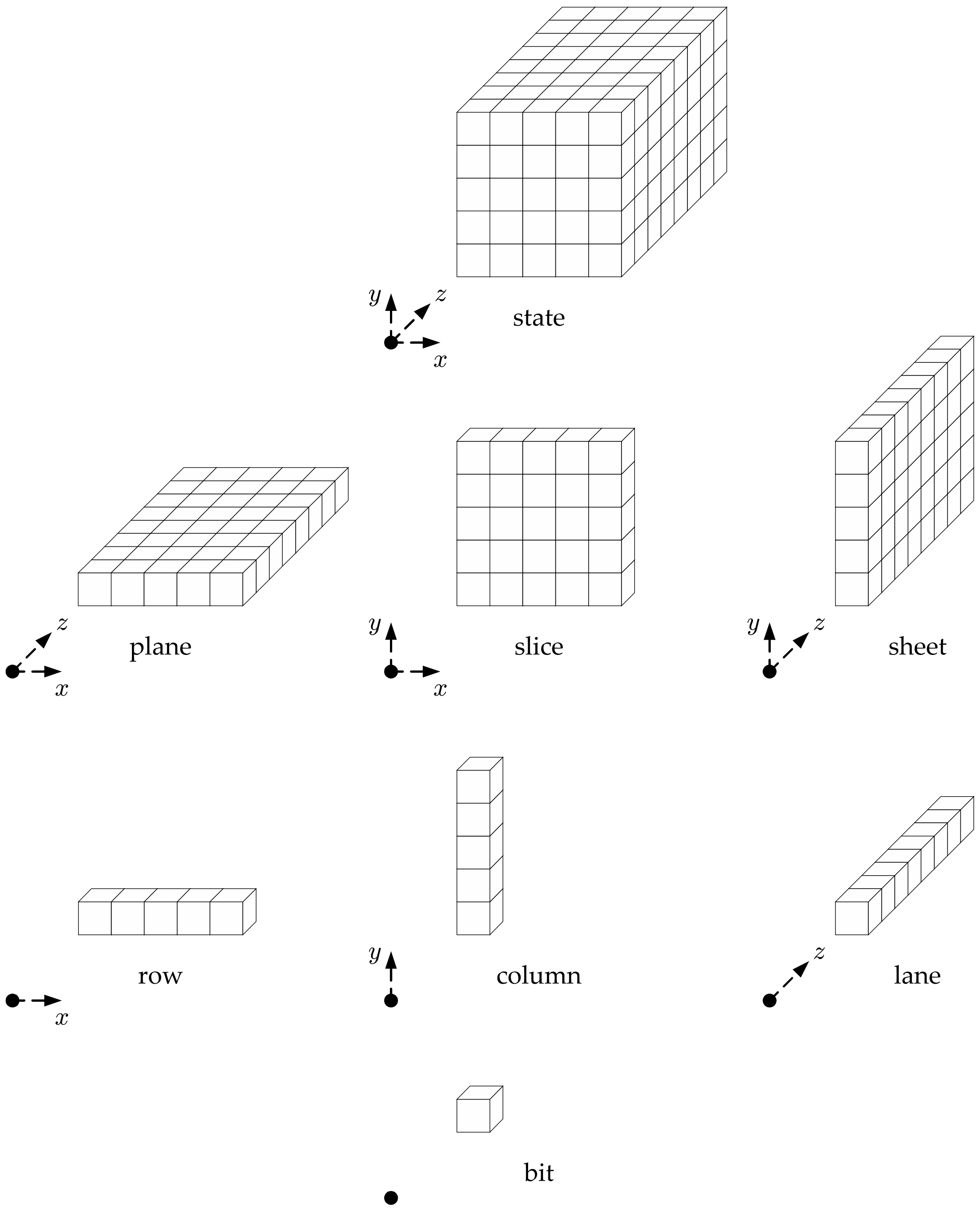

Fig. 1. Naming conventions for parts of the Keccak-f state