Compact FPGA Implementations of the Five SHA-3 Finalists

Stéphanie Kerckhof[1], François Durvaux[1],
Nicolas Veyrat-Charvillon[1], Francesco Regazzoni[1],
Guerric Meurice de Dormale[2], and François-Xavier Standaert[1]

[1] Université catholique de Louvain, UCL Crypto Group,
B-1348 Louvain-la-Neuve, Belgium
{stephanie.kerckhof,francois.durvaux,nicolas.veyrat,
francesco.regazzoni,fstandae}@uclouvain.be
[2] MuElec, Belgium
gm@muelec.com

Abstract. Allowing good performances on different platforms is an important criteria for the selection of the future SHA-3 standard. In this paper, we consider the compact implementations of BLAKE, Grøstl, JH, Keccak and Skein on recent FPGA devices. Our results bring an interesting complement to existing analyzes, as most previous works on FPGA implementations of the SHA-3 candidates were optimized for high throughput applications. Following recent guidelines for the fair comparison of hardware architectures, we put forward clear trends for the selection of the future standard. First, compact FPGA implementations of Keccak are less efficient than their high throughput counterparts. Second, Grøstl shows interesting performances in this setting, in particular in terms of throughput over area ratio. Third, the remaining candidates are comparably suitable for compact FPGA implementations, with some slight contrasts (in area cost and throughput).

Introduction

The SHA-3 competition has been announced by NIST on November 2, 2007. Its goal is to develop a new cryptographic hash algorithm, addressing the concerns raised by recent cryptanalysis results against SHA-1 and SHA-2. As for the AES competition, a number of criteria have been retained for the selection of the final algorithm. Security against cryptanalysis naturally comes in the first place. But good performances on a wide range of platforms is another important condition. In this paper, we consider the hardware performances of the SHA-3 finalists on recent FPGA devices.

In this respect, an important observation is that most previous works on hardware implementations of the SHA-3 candidates were focused on expensive and high throughput architectures, e.g. [17,25]. On the one hand, this is natural as such implementations provide a direct snapshot of the elementary operations' cost for the different algorithms. On the other hand, fully unrolled and pipelined

E. Prouff (Ed.): CARDIS 2011, LNCS 7079, pp. 217–233, 2011.
© IFIP International Federation for Information Processing 2011

architectures may sometimes hide a part of the algorithms' complexity that is better revealed in compact implementations. Namely, when trying to design more serial architectures, the possibility to share resources, the regularity of the algorithms, and the simplicity to address memories, are additional factors that may influence the final performances. In other words, compact implementations do not only depend on the cost of each elementary operation needed in an algorithm, but also on the number of different operations and the way they interact. Besides, investigating such implementations is also interesting from an application point of view, as the resources available for cryptographic functionalities in hardware systems can be very limited. Consequently, the evaluation of this constrained scenario is generally an important step in better understanding the implementation potentialities of an algorithm.

As extensively discussed in the last years, the evaluation of hardware architectures is inherently difficult, in view of the amount of parameters that may influence their performances. The differences can be extreme when changing technologies. For example, ASIC and FPGA implementations have very different ways to deal with memories and registers, that generally imply different design choices [14,16]. In a similar way, comparing FPGA implementations based on different manufacturers can only lead to rough intuitions about their respective efficiency. In fact, even comparing different architectures on the same FPGA is difficult, as carefully discussed in Saar Drimer's PhD dissertation [12]. Obviously, this does not mean that performance comparisons are impossible, but simply that they have to be considered with care. In other words, it is important to go beyond the quantified results obtained by performance tables, and to analyze the different metrics they provide (area cost, clock cycles, register use, throughput, ...) in a comprehensive manner.

Following these observations, the goal of this paper is to compare the five SHA-3 finalists on the basis of their compact FPGA implementation. In order to allow as fair a comparison as possible, we applied the approach described by Gaj *et al.* at CHES 2010 [14]. Namely, the IP cores were designed according to similar architectural choices and identical interface protocols. In particular, our results are based on the a priori decision to rely on a 64-bit datapath (see Section 3 for the details). As for their optimization goals, we targeted implementations in the hundreds of slices (that are the FPGAs' basic resources), additionally aiming for the best throughput over area ratio, in accordance with the usual characteristics of a security IP core. In other words, we did not aim for the lowest cost implementations (e.g. with an 8-bit datapath), and rather investigated how efficiently the different SHA-3 finalists allow sharing resources and addressing memories, under optimization goals that we believe reflective of the application scenarios where reconfigurable computing is useful.

As a result, and to the best of our knowledge, we obtain the first complete study of compact FPGA implementations for the SHA-3 finalists. For some of the algorithms, the obtained results are the only available ones for such optimization goals. For the others, they at least compare to the previously reported ones, sometimes bringing major improvements. For illustration purposes, we additionally provide

the implementation results of an AES implementation based on the same framework. Eventually, we take advantage of our results to discuss and compare the five investigated algorithms. While none of the remaining candidates leads to dramatically poor performances, this discussion allows us to contrast the previous conclusions obtained from high throughput implementations. In particular, we put forward that the clear advantage of Keccak in a high throughput FPGA implementation context vanishes in a low area one. Performance tables also indicate a good behavior for Grøstl in our implementation scenario, in particular when looking at the throughput over area evaluation metric.

1 SHA-3 Finalists

This section provides a quick overview of the five SHA-3 finalists. We refer to the original submissions for the detailed algorithm descriptions.

BLAKE. BLAKE [3] is built on previously studied components, chosen for their complementarity. The iteration mode is HAIFA, an improved version of the Merkle-Damgard paradigm proposed by Biham and Dunkelman [10]. It provides resistance to long-message second preimage attacks, and explicitly handles hashing with a salt and a “number of bits hashed so far” counter. The internal structure is the local wide-pipe, which was already used within the LAKE hash function [4]. The compression algorithm is a modified version of Bernstein’s stream cipher ChaCha [5], which is easily parallelizable. The two main instances of BLAKE are BLAKE-256 and BLAKE-512. They respectively work with 32- and 64-bit words, and produce 256- and 512-bit digests. The compression function of BLAKE relies heavily on the function G, which consists in additions, XOR operations and rotations. It works with four variables : *a, b, c* and *d*. It is called 112 to 128 times respectively for the 32- and 64-bit versions.

Grøstl. Grøstl [15] is an iterated hash function with a compression function built from two fixed, large, distinct but very similar permutations P and Q. These are constructed using the wide-trail design strategy. The hash function is based on a byte-oriented SP-network which borrows components from the AES [11], described by the transforms AddRoundConstant, SubBytes, ShiftBytes and MixBytes. Grøstl is a so-called wide-pipe construction where the size of the internal state (represented by a two 8×16-byte matrices) is significantly larger than the size of the output. The specification was last updated in March of 2011.

JH. JH [26] essentially exploits two techniques : a new compression function structure and a generalized AES design methodology, which provides a simple approach to obtain large block ciphers from small components. The compression function proposed for JH is composed as follows. Half of a 1024-bit hash value $H^{(i-1)}$ is XOR-ed with a 512-bit block message $M^{(i)}$. The result of this operation is passed through a bijective function E8 which is a 42-rounds block cipher with constant key. The output of E8 output is then once again XOR-ed with $M^{(i)}$. This paper considers the round 3 version of the JH specifications submitted to the NIST, in which the number of rounds has been increased from 35.5 to 42.

Keccak. Keccak [6] is a family of sponge functions [7], characterized by two parameters: a bitrate r, and a capacity c. The sponge construction uses $r + c$ bits of state and essentially works in two steps. In a first absorbing phase, r bits are updated by XORing them with message bits and applying the Keccak permutation (called f). Next, during the squeezing phase, r bits are output after each application of the same permutation. The remaining c bits are not directly affected by message bits, nor taken as output. The version of the Keccak function proposed as SHA standard operates on a 1600-bit state, organized in words. The function f is iterated a number of times determined by the size of the state and it is composed of five operations. Theta consists of a parity computation, a rotation of one position, and a bitwise XOR. Rho is a rotation by an offset which depends on the word position. Pi is a permutation. Chi consists of bitwise XOR, NOT and AND gates. Finally, iota is a round constant addition.

Skein. Skein [2] is built out of a tweakable block cipher [23] which allows hashing configuration data along with the input text in every block, and makes every instance of the compression function unique. The underlying primitive of Skein is the Treefish block cipher: it contains no S-box and implements a non-linear layer using a combination of 64-bit rotations, XORs and additions (i.e. operations that are very efficient on 64-bit processors). The Unique Block Iteration (UBI) chaining mode uses Threefish to build a compression function that maps an arbitrary input size to a fixed output size. Skein supports internal state sizes of 256, 512 and 1024 bits, and arbitrary output sizes. The proposition was last updated in October of 2010 (version 1.3) [13].

2 Related Works

Table 1 provides a partial overview of existing low area FPGA implementations for the SHA-3 candidates, as reported in the SHA-3 Zoo [1]. Namely, since our following results were obtained for Virtex-6 and Spartan-6 devices (as will be motivated in the next section), we list only the most relevant implementations on similar FPGAs.

BLAKE has been implemented in two different ways. The first one, designed by Aumasson *et al.* [3], consists in the core functionality (CF) with one G function. This implementation offers a light version of the algorithm but does not really exploit FPGA specificities. On the other hand, the second BLAKE implementation, by Beuchat *et al.* [9], consists in a fully autonomous implementation (FA) and is designed to perfectly fit the Xilinx FPGA architecture : the slice's carry-chain logic is exploited to build a adder/XOR operator within the same slices. The authors also harness the 18-kbit embedded RAM blocks (RAMB) to implement the register file and store the micro-code of the control unit. Table 1 shows Spartan-3 (S3) and Virtex-5 (V5) implementation results.

Jungk *et al.* [20][21] chose to implement the Grøstl algorithm on a Spartan-3 device. They provide a fully autonomous implementation including padding. The similarity between Grøstl and the AES is exploited and AES-specific optimizations

Table 1. Existing compact FPGA implementations of third round SHA-3 candidates (* padding included, ** Altera ALUTs)

	Algorithm	Scope	FPGA	Area [slices]	Reg.	RAMB	Clk cyc.	Freq. [MHz]	Thr. [Mbps]
Aumasson *et al.* [3]	BLAKE-32	CF	V5	390	-	-	-	91	575
Beuchat *et al.* [9]	BLAKE-32	FA	S3	124	-	2	844	190	115
Beuchat *et al.* [9]	BLAKE-32	FA	V5	56	-	2	844	372	225
Aumasson *et al.* [3]	BLAKE-64	CF	V5	939	-	-	-	59	533
Beuchat *et al.* [9]	BLAKE-64	FA	S3	229	-	3	1164	158	138
Beuchat *et al.* [9]	BLAKE-64	FA	V5	108	-	3	1164	358	314
Jungk *et al.* [21]	Grøstl-256	FA*	S3	2486	-	0	-	63	404
Jungk *et al.* [20]	Grøstl-256	FA*	S3	1276	-	0	-	60	192
Jungk *et al.* [20]	Grøstl-512	FA*	S3	2110	-	0	-	63	144
Homsirikamol *et al.* [17]	JH-256	FA	V5	1018	-	-	36	381	5416
Homsirikamol *et al.* [17]	JH-512	FA	V5	1104	-	-	36	395	5610
Bertoni *et al.* [8]	Keccak-256	EM	V5	444	227	-	5160	265	70
Namin *et al.* [24]	Skein-256	CF	AS3	1385**	1858	-	72	574	161

presented in previous works are applied. The table only reports the best and most recent results from [20]. Also, only serial implementations of P and Q are considered, because they better match our low area optimization goal.

No low area implementation of JH has been proposed up to now. In order to have a comparison, the implementation proposed by Homsirikamol *et al.* [17] may be mentioned. It is the high speed FPGA implementation that has the lowest area cost reported in the literature.

A low area implementation of the Keccak algorithm is given by Bertoni *et al.* [8]. In this implementation, the hash function is implemented as a small area coprocessor using system (external) memory (EM). In the best case, with a 64-bit memory, the function takes approximately 5000 clock cycles to compute. With a 32-bit memory, this number increases up to 9000 clock cycles.

Finally, Namin *et al.* [24] presented a low area implementation of Skein. It provides the core functionality and is evaluated on an Altera Stratix-III (AS3) FPGA.

3 Methodology

As suggested by the previous section, there are only a few existing low area FPGA implementations of the SHA-3 candidates up to now. Furthermore, those implementations often lack of similar specifications which make them difficult to compare. We therefore propose to design compact hardware cores of the five third-round candidates using a common methodology, which allows a fair comparison of the performances. The methodology we used mainly follows the one described by Gaj *et al.* [14], which suggests to use uniform interface and architecture, and defines some performance metrics.

First of all, we tried to keep the number of slices in the same range for all the implementations (typically between 150 and 300), with the throughput over area ratio as optimization target. This is a relevant choice for hardware cores, as they often need to be as efficient as possible with a limited resources usage. We then decided to primarily focus on the SHA-3 candidate variants with the 512-bit digest output size, as they correspond to the most challenging scenario for compact implementations - and may be the most informative for comparison purposes. For completeness, we also report the implementation results of the 256-bit versions in appendix, that are based on essentially similar architectures. Next, since we are implementing low area designs, we limited the internal datapath to 64-bit bus widths. This is a natural choice, as most presented algorithms are designed to operate well on 64-bit processors. Therefore, trying to decrease the bus size tends to be cumbersome and provides a limited area improvement at the expense of a significantly decreased throughput. In addition, we specified a common interface for all our designs, in which we chose to have an input message width of 64 bits, as this is the most convenient size to use with our 64-bit internal datapath. It also corresponds to our typical scenario, in which the hash IP cores have to be inserted in larger FPGA designs. Smaller or bigger message sizes would, most of the time, require additional logic in order to reorganise the message in 64-bit words. This is resources consuming and can be added externaly if needed by the user. All our cores have been designed to be fully autonomous, which will help us in the comparison of the total resources needed by each candidate.

Drimer presented in [12] that implementation results are subject to great variations, depending on the implementation options. Furthermore, comparing different implementations with each others can be irrelevant if not made with careful considerations. We therefore specified fixed implementation options and architecture choices for all our implementations. We choose to work on a Virtex-6 and Spartan-6 FPGAs, specifically a XC6VLX75T with speed grade -1 and a XC6SLX9 with speed grade -2, which are the most constraining FPGAs in their respective families, in terms of number of available logic elements. Note that the selection of a high-performance device is not in contradiction with compact implementations, as we typically envision applications in which the hash functionality can only consume a small fraction of the FPGA resources. Also, we believe it is interesting to detail implementation results exploiting the latest FPGA structures, as these advanced structures will typically be available in future low cost FPGAs too. In other words, we expect this choice to better reflect the evolution of reconfigurable hardware devices. Besides, and as will be illustrated by the implementation tables in Section 5, the results for Virtex-6 and Spartan-6 devices do not significantly modify our conclusions regarding the suitability of the SHA-3 finalists for compact FPGA implementations.

We did not use any dedicated FPGA resources such as block RAMs or DSPs. It is indeed easier to compare implementations when they are all represented in terms of slices rather than in a combination of several factors. Additionally, the use of block RAMs is often not optimal as they are too big for our actual needs. All the implementations took advantage of the particular LUT capabilities of

the Virtex-6 and Spartan-6, and use shift registers and/or distributed RAMs (or ROMs). The different modules are however always inferred so that portability to other devices is possible, even if not optimal. The design was implemented using ISE 12.1 and for two different sets of parameters. Those two sets are predefined sets available in ISE Design Goals and Strategies project options and are specified as "Area Reduction with Physical Synthesis" and "Timing Performance without IOB Packing". This choice was mainly motivated by the willing to illustrate the impact of synthesis options on the final performance figures.

We have made the assumption that padding is performed outside of our cores for the same reasons as in [14]. The padding functions are very similar from one hash function to another and will mainly result in the same absolute area overhead. Additionally, complexity of the padding function will depend on the granularity of the message (bit, byte, words,...) considered in each application.

Finally, the performance metrics we used in this text is always the throughput for long message (as defined in [14]). We did not specify the throughput for short message, but the information needed to compute it is present in the result tables of Section 5.

4 Architectures

This section presents the different compact architectures we developed. Because of space constraints, we mainly focus on the description of their block diagrams.

BLAKE. BLAKE algorithm is implemented as a narrow-pipelined-datapath design. The architecture of BLAKE is illustrated in Figure 1. The overall organization is similar to the implementation proposed by Beuchat *et al.*.

BLAKE has a large 16-word state matrix V but each operation works with only two elements of it. Hence, the datapath does not need to be larger than 64 bits. The operations are quite simple, they consist in additions, XOR and rotations. This allows us to design a small ALU embedding all the required operators in parallel, followed by a multiplexer. The way the ALU is build allows computing XOR-rotation and XOR-addition operations in one clock cycle.

Our BLAKE implementation uses distributed RAM memory to store intermediate values, message blocks and C constants. Using this kind of memory offers some advantages. Beyond effective slices occupation, the controller must be able to access randomly to different values. Indeed, message blocks and C constants are chosen according to elements of a permutation matrix. Furthermore, elements of the inner state matrix are selected in different orders during column and diagonal steps.

The 4-input multiplexer in front of the RAM memory is used to load message blocks (M), salt (S) and counter (T) through the *Message* input, to load the initialization vector (IV), to write the ALU results thanks to the feedback loop, and to set automatically the salt to zero if the user does not specify any value. Loading salt or initializing it to zero takes 4 clock cycles. Loading initialization vector takes 8 clock cycles. These two first steps are made once per message.

The two following steps, which are loading the counter and message block, take 18 clock cycles and are carried out at each new message block.

The scheduling is made so that, for each call of the round function G (as described in Section 1), the variable a is computed in two clock cycles, because it needs two additions between three different inputs. The three other variables (b, c, and d) are computed in one clock cycle thanks to the feedback loop on the ALU. As a result, one call of the G function needs 10 clock cycles to be executed. To avoid pipeline bubbles between column and diagonal steps, the ordering of G functions during diagonal step is changed to G_4, G_7, G_6 and G_5. The BLAKE-64 version needs 16 (rounds) $\times$ 8 (G calls) $\times$ 10 $=$ 1280 clock cycles to process one block through the round function, and 4 more ones to empty the pipeline. The initialization and the finalization steps need each 20 clock cycles. So, complete hashing one message block takes $18 + 1284 + 40 = 1342$ clock cycles. Finally, the hash value is directly read on the output of the RAM and takes 8 clock cycles to be entirely read.

As expected, these results are very close to those announced by Beuchat *et al.* [9] after adjustement (they considered 14 rounds for the BLAKE-64 version rather than 16), since the overall architectures are very similar.

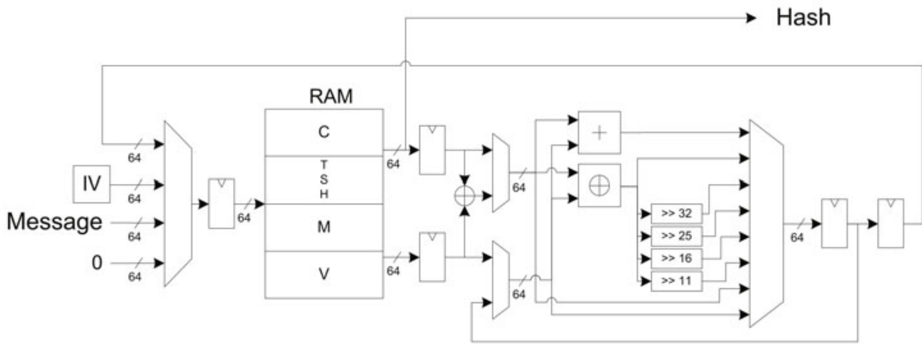

Fig. 1. BLAKE Architecture

Grøstl. The 64-bit architecture of Grøstl algorithm is depicted in Figure 2. This pipelined datapath implements the P and Q permutation rounds in an interleaved fashion (to avoid data dependency problems). The last round function (Ω) is implemented with the same datapath and only resorts to P. The difference between P and Q lies in slightly different AddRound constants and ShiftBytes shift pattern. Besides the main AES-like functions, there are several circuits. A layer of multiplexers and bitwise XORs is required at the beginning and at the end of the datapath. They implement algorithm initialization, additions necessary at beginning and end of each round, and internal and external data loading. Two distinct RAMs are used to store the P and Q state matrices and input message m_i (RAM *qpm*) and the hash result (RAM h). RAM *qpm* is a 64 $\times$ 64-bit dual port RAM. One RAM slot is used to store message m_i and three other slots are used to store current and next P and Q states (slots are used as a circular buffer).

RAM h is a 32×64-bit dual port RAM that stores current and next H (as well as final result).

The four main operations of each P or Q rounds are implemented in the following way. The ShiftBytes operation comes first. It is implemented by accessing bytes of different columns instead of a single column (as if RAM qpm was a collection of eight 8-bit RAMs), to save a memory in the middle of the datapath. Different memory access patterns (meaning different initialization of address counters) are required to implement P and Q ShiftBytes as well as no shift (for post-addition with h and hash unloading). Constants of AddRoundConstant are computed thanks to a few small-size integer counters (corresponding to the row and round numbers) and all-zero or all-one constants. Addition of those constants with data is a simple bitwise XOR operation. The eight S-boxes of SubBytes are simply implemented as eight 8×8-bit ROMs (efficiently implemented in 6-input look-up tables-based FPGAs). Finally, the MixBytes operation is similar to the AES MixColumn, except that 8×6 different 8-bit $\mathbb{F}_2$ multiplications by small constants are required, and that eight 64-bit partial products have to be added together. We implemented it as a large XOR tree, with multipliers hardcoded as 8-bit XORs and partial products XORed together.

Hashing a 1024-bit chunk of a message takes around 450 cycles: 16 (loading of m_i) + 14 (rounds) × 2 (interleaved P and textscq) × 16 (columns of state matrix) + 8 (ending). The last operation Ω requires around 350 cycles: 14 (rounds) × (16 (columns) + 6 (pipeline flush)) + 8 (ending) + 8 (hash output).

Roughly speaking, the most consuming parts of the architecture are MixBytes (accounting for 30 % of the final cost), the S-boxes (25 %) and the control of the dual port RAMs (25 %). Note that most pipeline registers are taken from already consumed slices, hence do not increase the slice count of the implementation.

JH. The JH architecture is illustrated in Figure 3 and is composed as follows. Two 16×32-bit single port distributed RAMs (HASH RAM) are used to store the intermediate hash values. Those RAMs are first initialized in 16 clock cycles with IV values coming from a 16×64-bit distributed ROM[1] and are afterwards updated with the output of R8 or the XOR operation output. R8 performs the round functions and is composed of sixteen 5×4 S-boxes, eight linear functions and a permutation. As the permutation layer always puts in correspondence two consecutive nibbles with a nibble from the first half and another from the second half of the permuted state, the output of R8 can be split into two 32-bits words, one coming from the first half and the other from the second half of the intermediate hash value. An address controller (ADDR CONTR), composed of two 16×4-bit dual-port distributed RAMs is then used to reach the wanted location in each HASH RAM, at each cycle. Rotations before and after R8 are needed to organize correctly the hash intermediate values in the two HASH RAMs.

A similar path is designed for constants generation. Two 16×8-bit single-port distributed RAM (CST RAMs) are used to store the constants intermediate values.

[1] IV ROM contains $H^{(0)}$ initial value and not $H^{(-1)}$ as defined in JH specifications. That way, we save 688 cycles of initialization and only loose a couple of slices

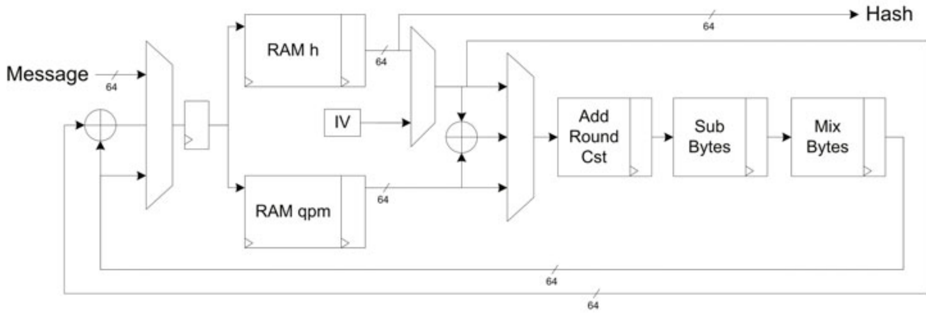

Fig. 2. Grøstl Architecture

The function R6 performs a round function on 16 bits of the constant state. The same address controller as for HASH RAMs is used for CST RAMs.

Finally, a GROUP/DE-GROUP block is used to re-organize the input message. As JH has been designed to achieve efficient bit-slice software implementations, a grouping of bits into 4-bit elements has been defined as the first step of the JH bijective function E8. Similarly, a de-grouping is performed in the last step of E8. When those grouping and de-grouping phases have no impact on high speed hardware implementations (as they result only in routing), this in not the case anymore for low area architectures. Indeed, those steps requires 16 additional clock cycles per message block, as well as more complex controls to access the single port RAMs. To avoid this, we chose to always work on a grouped hash and therefore to perform the data organization on the message with the GROUP/DE-GROUP block. The same component is also used to re-organize the hash final value before sending it to the user.

Our implementation of JH needs 16×42 clock cycles to compute the 42 rounds and 16 additional ones to perform the final XOR operation. In total, 688 clock cycles are required to process a 512-bit message block, 16 for RAM initialization and 20 additional clock cycles are used for the finalization step (4 to empty the pipeline and 16 to output hash from the GROUP/DE-GROUP component).

Keccak. Our architecture, depicted in Figure 4, implements the Keccak version proposed as SHA-3 standard. It works on state of 1600 bits organized into 25 words of 64 bits each. The whole algorithm does not use complex operations, but only XORs, rotations, negations and additions. The basic operations are performed on the 64-bit words, thus our implementation has a 64-bit internal datapath.

We maintained the same organization of Bertoni *et al.*, where the computation was split into three main steps: the first which does part of the theta transformation by calculating of the parity of the columns, the second which completes the theta transformation and performs the rho and pi transformations, and the third which computes the chi and iota steps. This structure requires a memory of 50 words of 64 bits, which are needed to store the state and the intermediate values at the end of the pi transformation.

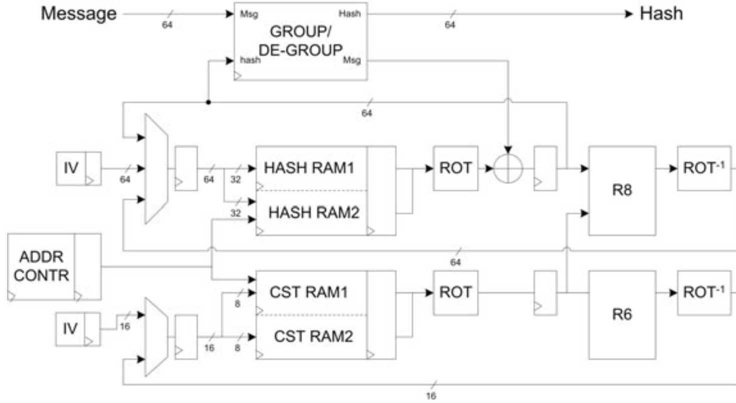

Fig. 3. JH Architecture

To allow parallel read/write operations and to simplify the access to the state, we organized the whole memory into two distinct asynchronous read single port RAM of 32×64-bit (RAM A and RAM B), and we reserved RAM B to store the output of the pi transformation.

Internally, our architecture has 5 registers of 64 bits, connected in order to create a word oriented rotator. During the theta transformation, the registers store the results of the computed parities. The rotator allows to quickly position the correct word for computing the second part of theta, as well as for computing the chi transformation.

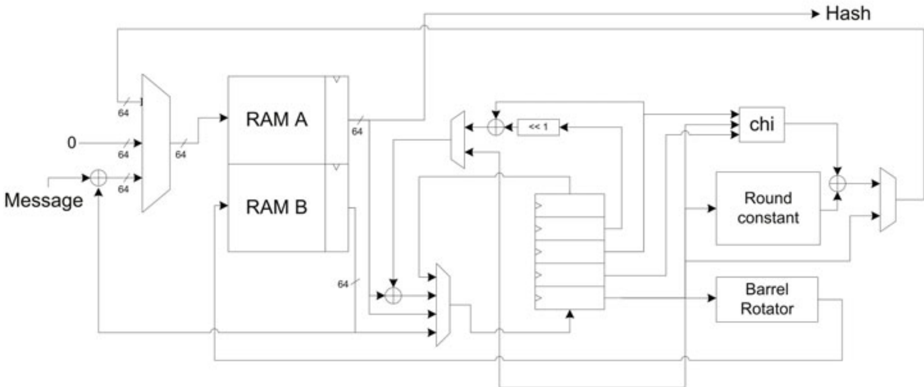

Fig. 4. Keccak Architecture

The most crucial part of Keccak is the rho transformation, which consist of rotation of words with an offset which depends from the specific index. We implemented this step efficiently in FPGA by using a 64-bit barrel rotator and by storing the rotation offsets into a dedicated look up table. The explicit

implementation of a barrel rotator allows to significantly reduce the area requirements in comparison to the use of a basic multiplexer. Furthermore, even if the 25 words of the state need to be processed successively by the same component, the use of a single barrel rotator reaches the trade off between the reached performances and the overall area cost which is more suitable for the scope of this paper.

Our implementation of Keccak requires 88 clock cycles to compute a single round. Since Keccak-1600 has 24 rounds, the total number of cycles required to hash a message is 2112, to which is should be added the initial XOR with the current state (25 cycles repeated for each block), the load of the message (9 cycles), and the offloading of the final result (8 cycles).

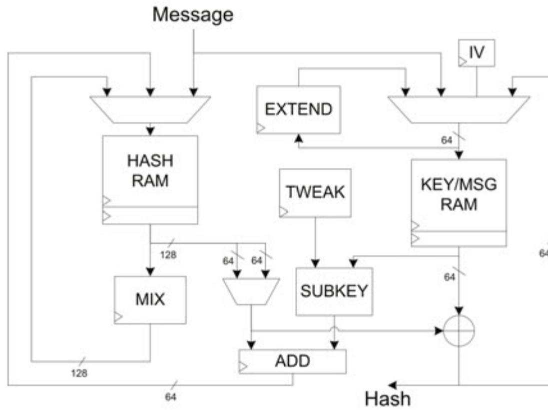

Fig. 5. Skein Architecture

Skein. Our implementation only contains a minimal set of operations necessary to the realization of round computations. In order to provide acceptable performances and memory requirements, the operations are not broken up all the way down to the basic addition, exclusive OR and rotate operations, but rather realize the MIX and subkey addition steps. The architecture is illustrated in figure 5. the initial UBI value is obtained through an 8×64-bit rom (IV) which avoids hashing the first configuration block. Key extension is performed on-the-fly using some simple arithmetic and a 64-bit register (EXTEND). One 17×64-bit RAM memory (KEY/MSG RAM) is used to store both the message block in view of the next UBI chaining, and the keys used for the current block. The hash variables can be memorized in two different 4×64-bit RAMs (HASH RAM), since the permute layer never swaps even and odd words. The permute operation itself is implicitly computed using arithmetic on memory addresses. The MIX operations take two 64-bit values (MIX), and require 4 cycles per round. The subkey addition acts on 64-bit values (ADD), requiring 8 cycles every 4 rounds. Subkeys are computed just before addition, with the help of the tweak registers (SUBKEY and TWEAK). Finally, a 64-bit XOR is used for UBI chaining. After the completion of round

operations, the hash digest is read from the key register. Given the variable management in this architecture, only single-port RAMs are needed, rather than the more expensive dual-port RAMs. All these are used asynchronously. When hashing a message, the operator first has to load the initialization vector, taking 9 cycles, followed by 457 cycles per 512-bit message block. Finally, one last block has to be processed before the hash value is output, leading to an overhead of 466 additional cycles.

5 Implementation Results and Discussion

The complete implementation results for our different architectures are given in Tables 2 and 3 for Virtex-6 and Spartan-6 devices, respectively. As expected, one can notice the strong impact of the two sets of options we considered (i.e. area and timing). Still, a number of important intuitions can be extracted.

In the first place, and compared to previous works, we see that our implementation results for BLAKE are quite close to the previous ones of Beuchat *et al.* The main differences are our exploitation of distributed memories (reported in the slices count) rather than embedded memory blocks and the fact that they implemented only 14 rounds, as specified in previous BLAKE-64 version, instead of 16. By contrast, for all the other algorithms, our results bring some interesting novelty. In particular, for Keccak, the previous architecture of Bertoni *et al.* was using only three internal registers, because of its compact ASIC-oriented flavor. This was at the cost of a weak performances, in the range of 5000 clock cycles per hash block. We paid a significant attention in taking advantage of the FPGA structure, in particular its distributed RAMs. As a result, we reduced the number of clock cycles by a factor of more than two. As for the three remaining algorithms, no similar results were known to date, which make them interesting, as first milestones.

Next, this table also leads to a number of comments regarding the different algorithms and their compact FPGA implementations. First, one can notice that Grøstl compares favorably with all the other candidates. While it has quite expensive components, interleaving the P and Q functions allows reducing the logic resources. More importantly, this algorithm proceeds blocks of 1024 bits and has a quite limited cycle count, which leads to significantly higher throughput than our other implementations.

BLAKE and JH also achieve reasonable throughput, but do not reach the level of performance of Grøstl in this case study. For BLAKE, the input blocks are still 1024-bit wide, but our implementation requires three times more cycles per block. For JH, it is rather the reduction of input block size that is in cause.

Skein provides interesting performances too. Its most noticeable limitation is a lower clock frequency, that could be improved by better pipelining the additions involved in our design. As a first step, we exploited the carry propagate adders that are efficiently implemented in Xilinx FPGAs. But this is not a theoretical limitation of the algorithm. One could reasonably assume that further optimization efforts would increase the frequency at the level of the other candidates.

Table 2. Implementation results for the 5 SHA-3 candidates on Virtex-6 (512-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	1024	1024	512	576	512	128
	Clock cycles per block	1342	448	688	2137	458	44
	Clock cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	701	912	789	519	770	658
	Number of Registers	371	556	411	429	158	364
	Number of Slices	192	260	240	144	240	205
	Frequency (MHz)	240	280	288	250	160	222
	Throughput (Mbit/s)	183	640	214	68	179	646
	Efficiency (Mbit/s/slice)	0.95	2.46	0.89	0.47	0.75	3.15
Timing	Number of LUTs	810	966	1034	610	1039	845
	Number of Registers	541	571	463	533	506	524
	Number of Slices	215	293	304	188	291	236
	Frequency (MHz)	304	330	299	285	200	250
	Throughput (Mbit/s)	232	754	222	77	223	727
	Efficiency (Mbit/s/slice)	1.08	2.57	0.73	0.41	0.77	3.08

Table 3. Implementation results for the 5 SHA-3 candidates on Spartan-6 (512-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	1024	1024	512	576	512	128
	Clock cycles per block	1342	448	688	2137	458	44
	Clock cycles overhead (pre/post)	12/8	24/354	16/20	9/8	9/466	8/0
Area	Number of LUTs	719	912	737	525	888	685
	Number of Registers	370	574	338	433	249	365
	Number of Slices	230	343	260	193	292	232
	Frequency (MHz)	135	240	113	166	91	125
	Throughput (Mbit/s)	103	548	84	45	102	364
	Efficiency (Mbit/s/slice)	0.47	1.60	0.32	0.23	0.35	1.57
Timing	Number of LUTs	856	766	1106	640	1059	852
	Number of Registers	594	759	646	476	395	529
	Number of Slices	303	281	362	216	351	274
	Frequency (MHz)	150	265	175	166	111	154
	Throughput (Mbit/s)	114	605	130	45	124	448
	Efficiency (Mbit/s/slice)	0.38	2.15	0.36	0.21	0.35	1.64

Finally, Keccak presents the poorest performances for the 512-bit digests. This is an interesting result in view of the excellent behavior of this algorithm in a high throughput implementation context [14]. Further optimizations could be investigated in order to reduce the number of clock cycles. But as long as a similar architecture as in this paper is used, this would probably be at the cost of a larger datapath (hence, higher slice count). Also, even considering an optimistic

50 cycles per round, the throughput of Keccak would remain 6 times smaller than the one of Grøstl. The main reason of this observation relates different rotations used in this algorithm (that come for free in unrolled implementations but may turn out to be expensive in compact ones) and to the large state that needs to be addressed multiple times when hashing a block. We note that our results are in line with the recent evaluations from CHES 2011 [18], where it is stated that Keccak is not straightforwardly suitable for folding[2]. An interesting alternative and scope for further research would be to change the overall architecture in order to better exploit the bit interleaving techniques described in [19].

Unsurprisingly, the main difference between the Virtex-6 and Spartan-6 implementations consists in a slightly larger number of slices, most likely due to the more constraining FPGA, and a reduction in frequency due to the lower performance of the Spartan-6 FPGAs.

In addition to these results, Table 4 in appendix provides the implementation results for the 256-bit digest versions of the hash algorithms, on Virtex-6. In general, these smaller variants do not exhibit significantly different conclusions. One important reason for this observation is that, when using distributed RAM's in an implementation, reducing the size of a state does not directly imply a gain in slices for a compact implementation (as only the depth of the memories are affected in this case). In fact, the move to 256-bit digests only implied a change of architecture for BLAKE (in the 256-bit version, we used a datapath size of 32 bits). Overall, this move towards smaller digests is positive for Keccak, because of a larger bitrate r, which allows this candidate to be more in line with the other finalists. By contrast, for BLAKE, the processing of 512-bit blocks does not come with a sufficient reduction of the number of rounds, hence leading to smaller throughputs. As for Grøstl, the number of rounds is also reduced by less than a factor 2, but the smaller number of columns in the state matrix allows keeping a higher throughput.

To conclude this work, we finally reported the performance results for an AES-128 implementation, with "on-the-fly" key scheduling, based on a 32-bit architecture. This implementation is best compared with the 256-bit versions of the SHA-3 candidates (because of a 128-bit key). One can notice that the slice count and throughput also range in the same levels as the ones of Grøstl.

Acknowledgements. This work was funded by the Walloon Region (S@T Skywin, MIPSs and SCEPTIC projects) and by the Belgian Fund for Scientific Research (FNRS-F.R.S. and FRIA funding). It also received support from the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II.

References

1. The sha-3 zoo, `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`
2. The skein hash function family, `http://www.skein-hash.info/`

[2] See also the following work to appear at INDOCRYPT 2011 [22].

3. Aumasson, J.-P., Henzen, L., Meier, W., Phan, R.C.-W.: Sha-3 proposal blake, version 1.4 (2011), http://131002.net/blake/
4. Aumasson, J.-P., Meier, W., Phan, R.C.-W.: The Hash Function Family LAKE. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 36–53. Springer, Heidelberg (2008)
5. Bernstein, D.J.: Chacha, a variant of salsa20. In: Workshop Record of SASC 2008: The State of the Art of Stream Ciphers (2008), http://cr.yp.to/chacha.html#chacha-paper
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The keccak sha-3 submission. Submission to NIST, Round 3 (2011)
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
8. Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: Keccak sponge function family main document, version 1.2. April 23 (2009), http://keccak.noekeon.org/
9. Beuchat, J.-L., Okamoto, E., Yamazaki, T.: Compact implementations of blake-32 and blake-64 on fpga. Cryptology ePrint Archive, Report 2010/173 (2010), http://eprint.iacr.org/
10. Biham, E., Dunkelman, O.: A framework for iterative hash functions - haifa. Cryptology ePrint Archive, Report 2007/278 (2007), http://eprint.iacr.org/
11. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus (2002)
12. Drimer, S.: Security for volatile FPGAs. Technical Report UCAM-CL-TR-763, University of Cambridge, Computer Laboratory (November 2009)
13. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The skein hash function family. Submission to NIST, round 3 (2011)
14. Gaj, K., Homsirikamol, E., Rogawski, M.: Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 264–278. Springer, Heidelberg (2010)
15. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schlffer, M., Thomsen, S.S.: Sha-3 proposal grøstl, version 2.0.1 (2011), http://www.groestl.info/
16. Henzen, L., Gendotti, P., Guillet, P., Pargaetzi, E., Zoller, M., Gürkaynak, F.K.: Developing a Hardware Evaluation Method for SHA-3 Candidates. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 248–263. Springer, Heidelberg (2010)
17. Homsirikamol, E., Rogawski, M., Gaj, K.: Comparing hardware performance of fourteen round two sha-3 candidates using fpgas. Cryptology ePrint Archive, Report 2010/445 (2010), http://eprint.iacr.org/
18. Homsirikamol, E., Rogawski, M., Gaj, K.: Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 491–506. Springer, Heidelberg (2011)
19. http://keccak.noekeon.org/
20. Jungk, B., Reith, S.: On fpga-based implementations of grøstl. Cryptology ePrint Archive, Report 2010/260 (2010), http://eprint.iacr.org/
21. Jungk, B., Reith, S., Apfelbeck, J.: On optimized fpga implementations of the sha-3 candidate grøstl. Cryptology ePrint Archive, Report 2009/206 (2009), http://eprint.iacr.org/

22. Kaps, J.-P., Yalla, P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., Pham, J.: Lightweight implementations of sha-3 candidates on fpgas. To appear in the Proceedings of IndoCrypt (2011)
23. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable Block Ciphers. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 31–46. Springer, Heidelberg (2002)
24. Namin, A.H., Hasan, M.A.: Hardware implementation of the compression function for selected sha-3 candidates. CACR 2009-28 (2009), http://www.vlsi.uwaterloo.ca/~ahasan/hasan_report.html
25. Tillich, S., Feldhofer, M., Kirschbaum, M., Plos, T., Marc-Schmidt, J., Szekely, A.: High-speed hardware implementations of blake, blue midnight wish, cubehash, ECHO, fugue, grøstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein. Cryptology ePrint Archive, Report 2010/445 (2010), http://eprint.iacr.org/
26. Wu, H.: The hash function jh. Submission to NIST, round 3 (2011)

A 256-Bit Digest Implementation Results

Table 4. Implementation results for the 5 SHA-3 candidates on Virtex-6 (256-bit digests)

		BLAKE	Grøstl	JH	Keccak	Skein	AES
Properties	Input block message size	512	512	512	1088	256	128
	Clock cycles per block	1182	176	688	2137	230	44
	Clock cycles overhead (pre/post)	12/8	24/122	16/20	17/16	5/234	8/0
Area	Number of LUTs	417	912	789	519	770	658
	Number of Registers	211	556	411	429	158	364
	Number of Slices	117	260	240	144	240	205
	Frequency (MHz)	274	280	288	250	160	222
	Throughput (Mbit/s)	105	815	214	128	179	646
	Efficiency (Mbit/s/slice)	0.90	3.13	0.89	0.89	0.75	3.15
Timing	Number of LUTs	500	966	1034	610	1039	845
	Number of Registers	284	571	463	533	506	524
	Number of Slices	175	293	304	188	291	236
	Frequency (MHz)	347	330	299	285	200	250
	Throughput (Mbit/s)	132	960	222	145	223	727
	Efficiency (Mbit/s/slice)	0.75	3.27	0.73	0.77	0.77	3.08