Unfolding Method for Shabal on Virtex-5 FPGAs: Concrete Results

Julien Francq, Céline Thuillet
EADS Defence & Security

***Abstract*—Recent cryptanalysis on SHA-1 family has led the NIST to call for a public competition named SHA-3 Contest. Efficient implementations on various platforms are a criterion for ranking performance of all the candidates in this competition. It appears that most of the hardware architectures proposed for SHA-3 candidates are basic. In this paper, we focus on an optimized implementation of the Shabal candidate. We improve the state-of-the-art using the unfolding method. This transformation leads to unroll a part of the Shabal core. More precisely, our design can produce a throughput over 3 Gbps on Virtex-5 FPGAs, with a reasonable area usage.**

***Index Terms*—SHA-3 Contest, Hash functions, Shabal, High-speed, Hardware, FPGA.**

I. Introduction

Hash functions are commonly used in digital signature applications, authentication protocols, key establishment and random number generation. They are designed in order to convert one large message with arbitrary length to one fixed length digest. Due to this feature, generic attacks exist to find collisions, first pre-image or second pre-image. Important properties for hash functions are computational resistance to these attacks. Security bounds of these functions must be as close as possible to generic attacks complexity.

The current standard is defined in FIPS 180-3[1] by the National Institute of Standards and Technology (NIST). Recent cryptanalysis on SHA-1 family has led the NIST to open a public competition called SHA-3 Contest[2], similar to the past one for Advanced Encryption Standard (AES). Its aim is to choose a new standard for hash functions. The submitted algorithms must respect some requirements, such as different sizes of message digest, security bounds, tunable parameters to adjust security and efficiency on various platforms.

In this contest, 64 candidates were submitted. 51 of these candidates were accepted in the first round. The second round begun on July 24, 2009 and the list of the candidates were reduced to 14 submissions[3]: BLAKE, BLUE MIDNIGHT WISH, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD and Skein.

The candidates are submitted to public comments for security aspects. Efficient implementations, both software and hardware, on various platforms are also a criterion for ranking performance of all the candidates of the SHA-3 competition. Some comparative studies of SHA-3 candidates are published on hardware platforms [1], [2], [3]. However, these proposed designs are close to the description given by the submission. Only a few have proposed design optimizations [4].

1) Contributions: Shabal's permutation can be viewed as a NLFSR. This structure allows us to apply the unfolding method, well known for the stream ciphers like Grain [5] and Trivium [6]. The previous works on Shabal do not mention optimizations and point out restricted parallezibility as a limit of performance.

In this paper, we present optimizations for designing highly efficient implementations of Shabal, using intensive unfolding transformation described for example in [7], [8], [9]. This method leads to unroll the Shabal core.

We design our implementations on a FPGA platform rather than in an ASIC because of its low cost, its greater flexibility, and its easier validation process. FPGAs are re-programmable platforms. So they allow to propose quickly different designs in order to compare their efficiency. To ease comparisons with the state-of-the-art, we choose to give our implementation results of Shabal on the same platform, which is Virtex-5 FPGA [10].

Our goal is to present a high-efficient implementation of Shabal, i.e. with a high throughput (TP) with a minimum area usage defined as follows.

The throughput for long messages can be calculated with the following equation:

$$TP = \frac{\#\text{Message block length} \times \text{Frequency}}{\#\text{Cycles per message block}}. \qquad (1)$$

2) Outline: First of all, we will give a short description of the Shabal specifications. Then, our different implementations and results will be provided as well as a comparison with the state-of-the-art. Finally, our results will be put back in the wider context of the SHA-3 competition.

[1]This publication is available electronically from the NIST web site: http://csrc.nist.gov/publications/PubsFIPS.html.

[2]All details are available on the website: http://csrc.nist.gov/groups/ST/hash/sha-3/.

[3]Information about theses submissions are available on the website: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

II. SHABAL

Shabal is a SHA-3 candidate designed by the Saphir project.

The Shabal construction is an iterative hash function based on a keyed permutation $\mathcal{P}$. The message blocks are 512-bit long. As required by the NIST, Shabal is declined in different versions, each corresponding to one output length, noted l_h, where $l_h \in \{192, 224, 256, 384, 512\}$. Shabal have the same structure for all output sizes, *i.e.* the TP for the FPGA implementations are the same.

The internal state is 1408-bit long, divided into 3 parts (A, B, C). As specified, A is respectively 12 $\times$ 32-bit words, and B, C are 16 $\times$ 32-bit words.

A. Notations

In this section, we introduce notations used in this paper. Let x and y be 32-bit words.

1) Bitwise operations: $x \oplus y$ represents the bitwise *exclusive or* (XOR) of x and y. We denote the bitwise logical *and* of x and y by $x \wedge y$. The complement of x is indicated by $\overline{x}$, equivalent to $x \oplus \texttt{0xFFFFFFFF}$. Finally $x \lll j$ means the left rotation of x by j, with $j \in \{1, 15\}$.

2) Wordwise operations: Shabal will also use *wordwise* operations such as addition and subtraction modulo 2^{32}. We denote the addition modulo 2^{32} by $\boxplus$, and $\boxminus$ for the subtraction modulo 2^{32}.

B. Description

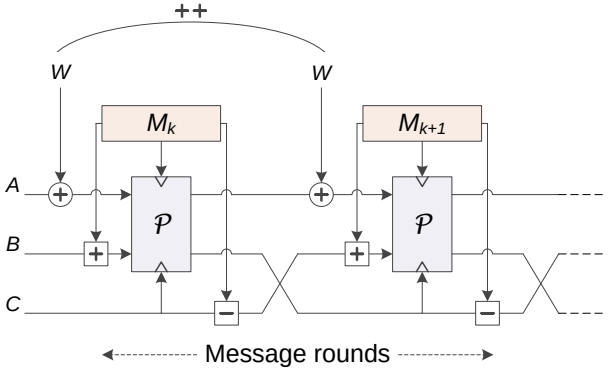

Fig. 1. Shabal mode of operation

Concerning the mode of operation, which is shown on see Fig. 1, the internal state is initialized by an Initial Vector (IV), which can be computed on the fly or pre-computed. The IV computation depends on the output length.

For the message rounds, the current message block M_k is added to the B part of the internal state. A counter W is incremented and XORed to the two leftmost words of A. The keyed permutation $\mathcal{P}$ is applied on A and B, with M_k and C as key materials. At the end, the message block M_k is subtracted to C. Finally B and C are swapped.

$\mathcal{P}$ is the core of the round function. It can be viewed as a Non-Linear Feedback Shift Register (NLFSR). The operations realized during permutation are quite simple: XORs, ANDs, NOTs, rotations and multiplications by small constant. Figure 2 illustrates one iteration of the permutation, which is applied 48 times by message round M_k.

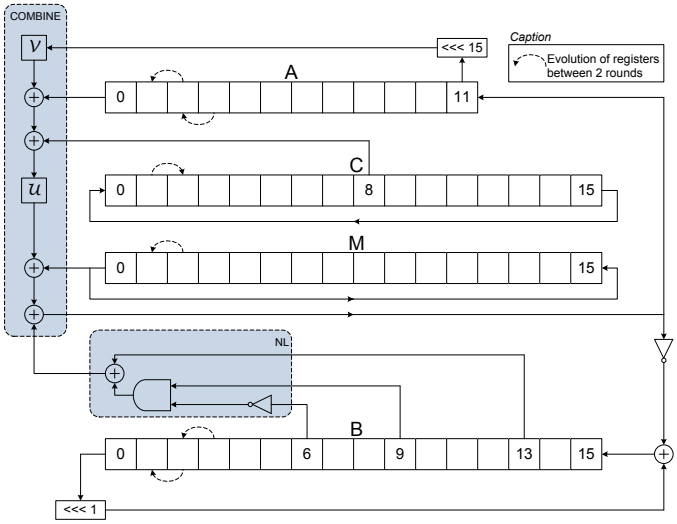

Fig. 2. Permutation $\mathcal{P}$ of Shabal

In this paper, we will introduce two notations concerning the permutation $\mathcal{P}$, in order to simplify upcoming figures: $COMBINE$ and NL. Each entity gathers some operations in one iteration in $\mathcal{P}$.

Here is one iteration of the permutation $\mathcal{P}$ as defined in the submission package:

$$\begin{aligned} A[i+16j \bmod 12] \leftarrow \mathcal{U}\big(&A[i+16j \bmod 12] \\ &\oplus \mathcal{V}(A[i-1+16j \bmod 12] \lll 15) \\ &\oplus C[8-i \bmod 16]\big) \\ &\oplus B[i+13 \bmod 16] \\ &\oplus (B[i+9 \bmod 16] \wedge \overline{B[i+6 \bmod 16]}) \\ &\oplus M[i]. \end{aligned}$$

It can be written as:

$$\begin{aligned} A[i+16j \bmod 12] \leftarrow &COMBINE(A[i+16j \bmod 12], \\ &A[i-1+16j \bmod 12] \lll 15, \\ &C[8-i \bmod 16], \\ &NL(B[i+13 \bmod 16], B[i+9 \bmod 16], \\ &\quad B[i+6 \bmod 16]), \\ &M[i]). \end{aligned}$$

III. OUR DIFFERENT HARDWARE IMPLEMENTATIONS

In this section, our different hardware implementations will be described, from the basic one to the improved-throughput one.

A. Our first implementation

Our first implementation is close to the original description given by the Shabal's submission package [11]. The previous

works [1], [2], [3] are quite similar. In the following, we discuss some implementation details which have been investigated in our study.

1) $\mathcal{P}$ as a NLFSR: As described by Shabal contributors (see Fig. 2), all our hardware implementations keep the NLFSR structure of the permutation $\mathcal{P}$. So after each iteration in $\mathcal{P}$, the whole internal state is shifted.

2) $\mathcal{U}$ and $\mathcal{V}$ with shift-then-add method: In one iteration in $\mathcal{P}$, it is applied two multiplications by small constant 3 and 5, denoted respectively by $\mathcal{U}$ and $\mathcal{V}$ (see COMBINE part in Fig. 2). We choose to implement each multiplication with the shift-then-add method.

3) Final additions in parallel: At the end of the permutation, additions of words of C to A are required[4] (see Fig. 3).

For j from 0 to 35
Do
- $A[j \bmod 12] \leftarrow A[j \bmod 12] \boxplus C[j+3 \bmod 16]$

Next j
Done.

Fig. 3. Last loop of $\mathcal{P}$

By unrolling this loop, it can be noticed that each new A word can be computed using three additions [2], [11] (see Fig. 4).

Two well-known ways of computing the new value of one A word can be used. The first one consists in cascading three Ripple-Carry Adders (RCAs). The second one consists in computing a multi-operand addition using four-to-two ([4:2]) Carry-Save Adders (CSAs) (see [12], Chapter 3 and Appendix A for more explanations). *A priori*, the second solution leads to a shorter critical path delay. However, this statement must be checked by a complete analysis based on the post-place and route results for both alternative architectures (see Section V). Indeed, Virtex 5 FPGAs contain embedded hardware structures for fast Ripple-Carry Addition.

For j from 0 to 11
Do
- $A[j] \leftarrow A[j] \boxplus C[j+3 \bmod 16]$
  $\boxplus C[j+15 \bmod 16] \boxplus C[j+27 \bmod 16]$

Next j
Done.

Fig. 4. Last loop of $\mathcal{P}$ unrolled

B. Unfolding method on $\mathcal{P}$

In order to improve performance, we perform an unfolding transformation [9] on $\mathcal{P}$. This technique is often used in designs which have a NLFSR structure, such as some stream ciphers like Grain and Trivium; it is also interesting to measure its efficiency on Shabal, since one iteration in $\mathcal{P}$ is viewed as a NLFSR (see Fig. 2).

[4]These additions are not depicted in Fig. 2 in order to keep this latter readable.

Fig. 5. DFG of the permutation $\mathcal{P}$

Considering the Data Flow Graph (DFG) of $\mathcal{P}$, we can unroll the 48 iterations as illustrated in Fig. 5. The internal state is updated at each iteration.

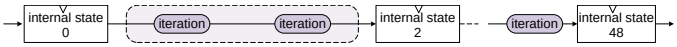

Fig. 6. Applying unfolding transformation with factor 2 to Fig. 5

This can be viewed as an unfolding transformation of "factor" 1. The requirement on this "unfolding factor" is to be a denominator of the iteration bound [9]; in Shabal permutation case: 48.

With an unfolding method of factor 2, we compute 2 iterations in $\mathcal{P}$ in one clock cycle (see Fig. 6).

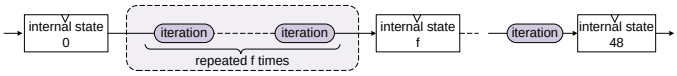

Fig. 7. Applying unfolding transformation with factor f to Fig. 5

As a general rule, the unfolding method with factor f gathers f iterations between internal state updates (see Fig. 7).

So, if the critical path delay is not multiplied by a factor greater than the factor f when the number of clock cycles per message block is divided by f, the new design obtains a better throughput (see relation (1) in Section I).

This method is detailed in the following sections for unfolding 2, 3, 4, 6 and 8 iterations. Other divisors of 48 are also possible values for f (12, 16, 24), but applying only unfolding method, they do not bring any improvement on the throughput. For these great factors, it will be necessary to apply retiming methods[9].

1) Unfolding 2 iterations in $\mathcal{P}$: So the unfolding method transforms the NLFSR. Here is represented the transformation using the unfolding factor 2 (see Fig. 8).

It shows how we put the computation of two values in a same iteration. The result $A[11]$ depends on $A[10]$. So when $A[10]$ is computed, it is chained directly in the rotation by 15, without taking the value from A.

2) Unfolding 3 iterations in $\mathcal{P}$: As explained in the previous section, we apply the method with the factor 3. Adapting the previous figure for 3 computations is quite immediate.

3) Unfolding 4 iterations in $\mathcal{P}$: Now with an unfolding factor equals to 4, the new iteration computes 4 iterations before updating the internal state(see Fig. 9).

Due to dependence between computed values, the first input of $NL(3)$ is one of the new value computed at the first iteration. So, in order to avoid synchronization issues, we decide to directly input this value in $NL(3)$. This implies that $NL(3)$ takes only 2 of these inputs in the register latch B.

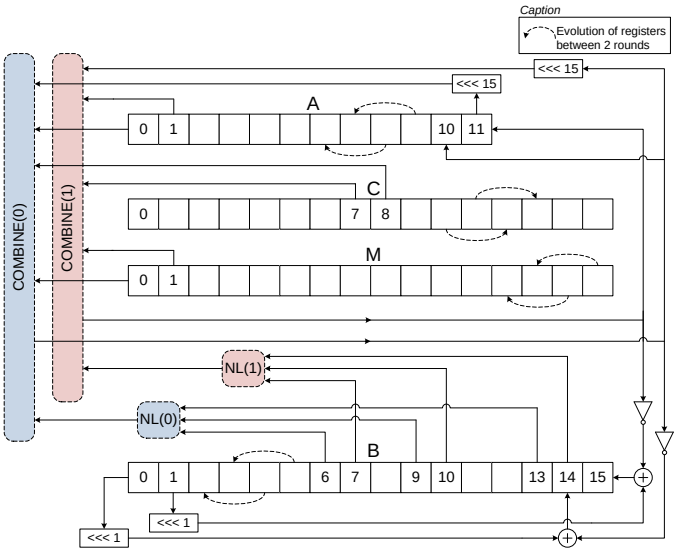

Fig. 8. Two iterations of the permutation $\mathcal{P}$

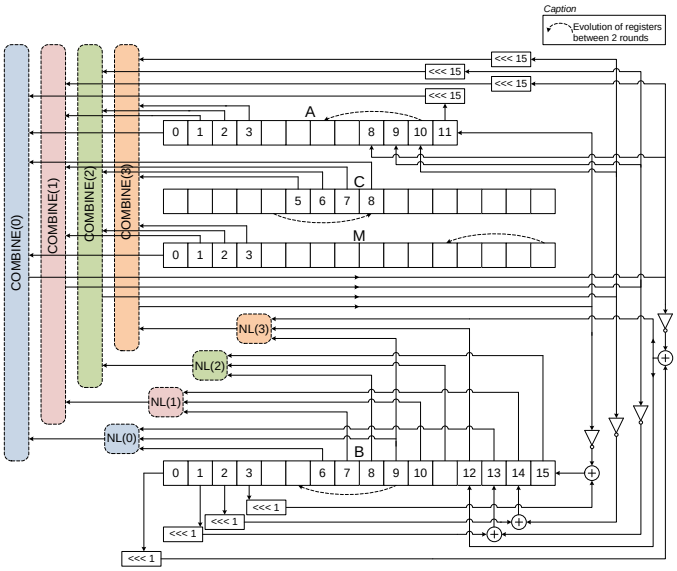

Fig. 9. Four iterations of the permutation $\mathcal{P}$

4) Unfolding 6 iterations in $\mathcal{P}$: As mentionned before, a factor greater than 3 leads to map some intermediate values directly on previous outputs, *i.e.* before storing them in register files.

5) Unfolding 8 iterations in $\mathcal{P}$: The unfolding method is similarly applied with the factor 8. This version is not depicted in order to stay clear in this paper.

IV. Hardware considerations

A. Around the mode of operation

A cryptographic hash function outputs a hash message by breaking an input message into a serie of fixed-length blocks, and by operating on them. Many hash algorithms include initialization, padding and finalization steps.

The initialization step uses IV to modify the internal state before inserting message blocks.

The input message is then padded so that its length becomes a multiple of the block length. We consider that the padding step is done in software: already padded message blocks are used as inputs to hardware cores. This statement allows to compare our implementation with the state-of-the-art. However the padding could be computed in hardware [13].

Finally, a hash function can use finalisation step before outputting the digest.

We focus on implementing the mode of operation, excluding initialisation, padding and final steps. The throughput of the algorithm is determined by the throughput of a message round. This is especially true for long messages.

B. Interfaces

In order to take into account the restricted number of I/O pins in the place and route process, we have to use a wrapper [14]. This latter connects the I/O of our core to I/O pins of the FPGA. To ensure that the wrapper does not affect the throughput of the design, the number of clock cycles needed to load data must be less than the number of clock cycles needed for core computation. So, 32-bit I/O are needed for the first version (resp. 32-bit I/O for the second, 64-bit for the third and 128-bit I/O for the fourth) in order to load data in less than 48 cycles (resp. 24, 16, 12, 8, 6) [5].

V. Results

The four proposed Shabal hardware modules described in Section III were implemented in VHDL. Each description was then verified against the official test vectors using simulations with ModelSim®. Synthesis and Place and Route process were carried out using Xilinx™ ISE® tools (v11.1i). Since our main goal was to optimize throughput, synthesis options were then chosen to maximize the operating frequency of all the presented designs. The target device was a Xilinx™ Virtex-5® FX70T, speed grade 3, package FF1136 (xc5vfx70t-3ff1136).

In this section, we firstly analyse the relative performance of our four hardware implementations of Shabal before explaining why our results should be considered for the SHA-3 Contest.

A. Analysis of our implementation results

Post-Place and Route results for the Shabal compression function are summarized in Table I. Besides TP, we introduce the *ratio* "throughput per slice" in order to determine which Shabal version makes the most efficient use of Virtex-5® area.

[5]Instead of increasing continuously I/O bus width, a second faster I/O clock could be used on top of the main clock used for core computation. This solution has not been implemented since we want to manage only one clock.

Unfolding factor f	Area (slices)	Max. Freq. (MHz)	TP (Gbps)	TP/Area (Mbps/slice)
1	**1533**	**247**	2.634	1.718
2	1556	144	3.072	1.974
3	1607	101	3.232	**2.011**
4	1715	76	**3.242**	1.890
6	1884	50	3.200	1.698
8	1958	37	3.157	1.612

TABLE I
SHABAL IMPLEMENTATION RESULTS

It must be noticed that in our first implementation, the critical path is from register A[11] to register B[15] (see Fig. 2) and not on the final adders. Due to the CSAs use, the critical path changed and the performance improved. For the next versions, replacing CSAs by RCAs leads to similar performance. We keep the CSAs for all the implementations, in order to measure the unfolding influence.

Our first implementation can be viewed as a reference implementation in order to measure the benefits brought by our other implementations.

Using the unfolding method, it is expected that the frequency will be lower. However, if the loss factor is less than the unfolding factor, the TP will be improved. This effect happens for the factors up to 4. But with 6 and 8, the TP decreases, showing up the limit of the unfolding method due to operations added and more complex place-route step. We hope that this effect can be reduced with retiming method described in [9].

Concerning area, the unfolding method leads to a slight overhead, improving the FPGA usage (TP/slices *ratio*). This ratio increases up to 2.011 for the factor 3.

As expected, the frequency is higher in the basic implementation. In terms of TP (resp. TP/slice) the unfolding factor 4 (resp. 3) produces the best implementation, with an area similar to the basic one. Among the four implementations, the third is the most efficient. The last version points out the limit of the unfolding method concerning Shabal architecture.

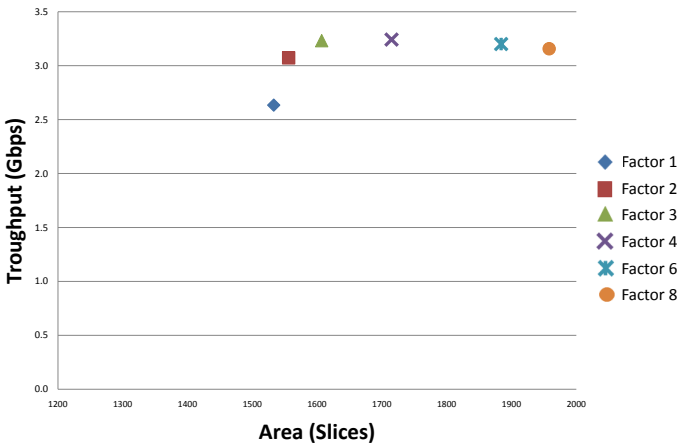

Fig. 10. Improvement of TP/slices on each implemented version

1) Our main contribution: As expected, the critical path of each design increases with the degree of unfolding (see Fig. 10). However the TP is improved up to 3.242 Gbps. Results have shown that the best TP is obtained by unfolding 4 iterations in $\mathcal{P}$. This version presents improvements concerning TP with slight overhead, comparing to the first implementation (see Table I). So the unfolding method with factor 3 leads to a better efficient implementation. They also shows that it is possible optimize Shabal even if it has limited parallelizability.

B. Improvements on previous works

Reference	Area (slices)	Max. Freq. (MHz)	TP (Gbps)	TP/Area (Mbps/slice)
[2]	2768	138	1.450	0.523
[15]	1171	126	2.588	2.210
[16]	1251	214	2.282[a]	1.390
Our work	1607	101	**3.232**	2.011

[a]Replacing SASEBO 16-bit interface by one like ours

TABLE II
COMPARISON OF PREVIOUS WORKS

In Table II, we summarize the results of our best high-efficiency implementation, *i.e.* better TP per slice ratio, and compare it with the results for other Shabal implementations on Virtex-5®. Our goal is to provide an high-efficient design. Taking the third implementation (unfolding factor = 3), our design produces substantial improvements in Shabal hardware implementation. Despite its restricted parallelizability, Shabal shows a good high-efficient performance in hardware with acceptable area usage. In terms of TP, our implementation with unfolding factor equals to 4 outperforms the state-of-the-art by a significant margin (+20%).

VI. CONCLUSION

In the SHA-3 Contest, efficiency of hardware implementation is an important criterion for ranking the candidates. Moreover, the state-of-the-art proposes only basic designs. This is why we propose in this paper a high-throughput per slice optimized implementation of Shabal on a Virtex-5 FPGA that outperforms the state-of-the-art.

Particularly, the unfolding transformation leads to interesting results, through our proposed designs. Mainly, our best design is the first which produces a throughput over 3 Gbps, with a reduced overhead in area comparing to the state-of-the-art. A higher throughput rate makes Shabal attractive for some applications which needs efficiency in constrainted area.

Also, we have shown that the unfolding method has some limits. A perspective is to combine this with retiming methods [9].

Low-area cost allows to put several Shabals in the same FPGAs, useful to process several independant messages in parallel. In this kind of applications, the throughput can be multiplied by the number of implemented Shabal cores.

Finally, in a wider context, we hope that our work provides a significant contribution in the hardware benchmarking of the SHA-3 candidates.

ACKNOWLEDGMENT

This work was partially supported by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014.

REFERENCES

[1] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely. 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 2009/510, 2009. http://eprint.iacr.org/.

[2] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan, and W. P. Marnane. FPGA Implementations of SHA-3 Candidates: CubeHash, Grostl, LANE, Shabal and Spectral Hash. In *Proc. Digital System Design − DSD*, pages 783–790, 2009.

[3] M. Bernet, L. Henzen, H. Kaeslin, N. Felber, and W. Fichtner. Hardware Implementations of the SHA-3 Candidates Shabal and CubeHash. *IEEE Midwest Symposium on Circuits and Systems*, pages 515–518, 2009.

[4] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA. Cryptology ePrint Archive, Report 2010/173, 2010. http://eprint.iacr.org/.

[5] M. Hell, T. Johansson, and W. Meier. Grain-A Stream Cipher for Constrained Environments. eSTREAM, ECRYPT Stream Cipher. *2005/010, ECRYPT (European Network of Excellence for Cryptology)*, 2005.

[6] C. De Canniere and B. Preneel. Trivium Specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.

[7] R. Hoare, P. Menon, and M. Ramos. 427 Mbits/sec Hardware Implementation of the SHA-1 Algorithm in an FPGA. *International Association of Science and Technology for Development (IASTED) Journal*, pages 381–142, 2002.

[8] R. Lien, T. Grembowski, and K. Gaj. A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. *Topics in Cryptology–CT-RSA 2004*, pages 1995–1995, 2004.

[9] Y. K. Lee, H. Chan, and I. Verbauwhede. Design Methodology for Throughput Optimum Architectures of Hash Algorithms of the MD4-class. *J. Signal Process. Syst.*, 53(1-2):89–102, 2008.

[10] Virtex-5 Family Overview, Xilinx. 2009. http://www.xilinx.com/support/documentation/virtex-5.htm.

[11] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau. Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition. Submission to NIST, 2008.

[12] M. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[13] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane. A Hardware Wrapper for the SHA-3 Hash Algorithms. Cryptology ePrint Archive, Report 2010/124, 2010. http://eprint.iacr.org/.

[14] Z. Chen, S. Morozov, and P. Schaumont. A Hardware Interface for Hashing Algorithms. Cryptology ePrint Archive, Report 2008/529, 2008. http://eprint.iacr.org/.

[15] R. Feron and J. Francq. FPGA Implementation of Shabal: Our First Results. SHA-3 Zoo, 2008. http://ehash.iaik.tugraz.at/uploads/d/d4/FPGA_Implementation_of_Shabal_-_First_Results.pdf.

[16] K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta. Evaluation of Hardware Performance for the SHA-3 Candidates Using SASEBO-GII. Cryptology ePrint Archive, Report 2010/010, 2010. http://eprint.iacr.org/.

APPENDIX

Let's take the example of the computed value $A[0] \leftarrow A[0] \boxplus C[3] \boxplus C[15] \boxplus C[11]$ in the loop described in Fig. 4, for $j = 0$. Initially, the operands $A[0]$, $C[3]$, $C[15]$ and $C[11]$ use a conventional binary representation. For example,

$$A[0] = (A[0]_{31} \cdots A[0]_1 A[0]_0)_2 = \sum_{i=0}^{31} A[0]_i 2^i, \text{ with } A[0]_i \in \{0, 1\}.$$

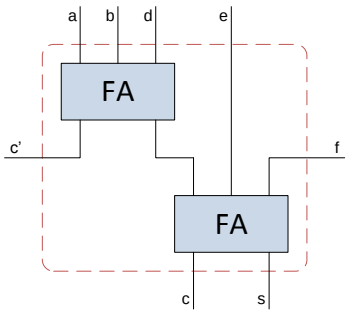

Fig. 11. [4:2] CSA_i

The [4:2] CSA_i (see Fig. 11) sums up four-bit inputs that are $a = A[0]_i$, $b = C[3]_i$, $d = C[15]_i$ and $e = C[11]_i$ and outputs two bits in the redundant Carry-Save (CS) form.

It must be noted that the c' value of the [4:2] CSA_i is connected to the f value of the CSA_{i+1}. Thus, 32 [4:2] CSAs are needed per 32-bit 4-input addition.

The output of a 4-input addition using [4:2] CSAs (in our example $A[0]$) is given in CS. In this notation, the number $A[0]$ is represented in radix 2 using digits $A[0]_i \in \{0, 1, 2\}$ coded by 2 bits such that $A[0]_i = A[0]_{i,c} + A[0]_{i,s}$ where $(A[0]_{i,c}, A[0]_{i,s}) \in \{0, 1\} \times \{0, 1\}$:

$$A[0] = \sum_{i=0}^{31} A[0]_i 2^i = \sum_{i=0}^{31} (A[0]_{i,c} + A[0]_{i,s}). \quad (2)$$

In order to convert the result $A[0]$ in a conventional binary representation, the final step shown in Equation 2 must be computed with Ripple-Carry Adder.