2011 International Conference on Reconfigurable Computing and FPGAs

Area-efficient FPGA Implementations of the SHA-3 Finalists

Bernhard Jungk and Jürgen Apfelbeck
Hochschule RheinMain, Wiesbaden, Germany
Email: {bernhard.jungk, juergen.apfelbeck}@hs-rm.de

***Abstract*—Secure cryptographic hash functions are core components in many applications like challenge-response authentication systems or digital signature schemes. Many of these applications are used in cost-sensitive markets and thus low budget implementations of such components are very important.**

In the present paper, we focus on the new SHA-3 competition, started by the National Institute of Standards and Technology (NIST), which searches for a new hash function in response to security concerns regarding the previous hash functions SHA-1 and the SHA-2 family. This work adds new valuable data to the competition, by providing an evaluation of area-efficient implementations of all finalists.

Our results show, that it is possible to implement all candidates reasonably small. We focus on area-efficiency and therefore we do not rank the candidates by absolute throughput, but rather by the area and the throughput-area ratio. The results hint that Grøstl and Keccak are the best overall performers for compact implementations, if the throughput-area ratio is most important. The following candidate is BLAKE, while the Skein and JH implementations trail behind. The area ranking changes the results and puts JH on the top, followed by BLAKE, Grøstl, Keccak and Skein.

Key words: Cryptography, Hash Functions, SHA-3, Compact Implementation, FPGA

I. Introduction

The SHA-3 competition, run by the National Institute of Standards and Technology (NIST), spawns a lot of new research on hash functions. The competition itself is very much organized like the past AES competition (cf. [1]), and has the goal to overcome security problems and speculations about the SHA-1 (cf. [2]) and the SHA-2 family (e.g. [3], [4]). Similar to the former effort, this competition requires third party software and hardware implementations of all proposed candidates to evaluate the overall performance and resource requirements.

In the present paper, we describe FPGA designs of all SHA-3 finalists, namely BLAKE (cf. [5]), Grøstl (cf. [6]), JH (cf. [7]), Keccak (cf. [8]) and Skein (cf. [9]). For all candidates, an area-optimized implementation was developed and evaluated. Most of the applied optimizations are of architectural nature, reducing the number of slices by arranging the necessary registers, RAMs and logic or by pipelining. However, the serialization of the algorithms is the main tool to build area-efficient designs. This technique is often called folding (cf. [10]).

To the best of our knowledge, our results are the first published results for compact implementations of all finalists for Virtex-5 FPGAs. The only group we know of, which published compact designs of all finalists, reported their results only for Virtex-6 and Spartan-6 FPGAs (cf. [11]). The throughput-area ratio of our implementations ranks the candidates in the following order: Grøstl, Keccak, BLAKE, Skein, JH. If we disregard the throughput and look only on the area, the ranking changes and JH is the first one, followed by BLAKE, Grøstl, Keccak and the biggest one Skein.

The rest of the paper discusses previous and related work on FPGA implementations in Section II and then describes the hardware interface in Section III. In Section IV each design is described in detail, followed by the evaluation of the implementation results (Section V) and our conclusion (Section VI).

II. Previous work

There is plenty of previous work regarding FPGA implementations of the SHA-3 finalists. Most of the previous implementations (e.g. [10], [12]) are optimized for high-throughput and much less is known about compact designs. Early results on compact BLAKE implementations are available in [13]. They use a similar approach to the proposed design in this paper, but they use block RAM instead of distributed RAM and thus, the area results are not comparable. Here, all algorithms were implemented using distributed RAM. Thus all required resources are included in the slice count (Tab. II).

Results on compact implementations of Grøstl, Skein and JH were reported in [14]. The present work improves on these results either by reducing the area while roughly maintaining the throughput-area ratio or by improving both. Furthermore this work adds brand new results on BLAKE as well as on Keccak. The first complete study of compact implementations was published in [11]. They reported results for all SHA-3 finalists for Virtex-6 and Spartan-6 FPGAs. They implemented all candidates for both 256 and 512 bit digests. Some of their designs differ considerably from the ones presented in the present paper, e.g. the new Keccak design is much faster, while others achieve a very similar overall performance (e.g. Grøstl).

There are other earlier ideas and results related to the present work. For example, the Grøstl hash function benefits from its similarity to the AES block cipher, because some

978-0-7695-4551-6/11 $26.00 © 2011 IEEE
DOI 10.1109/ReConFig.2011.16

of the optimizations applied to AES (e.g. [15], [16], [17], [18]) can be adapted to Grøstl. Good examples are the ideas for a compact AES implementation described in [17] or the AES S-box optimizations (e.g. [16]). One known actually non-optimization on most FPGA platforms are carry look-ahead adders, because they are actually often slower than the ripple-carry adder designs on FPGAs (cf. [19]), which affects mainly Skein and to a lesser extend BLAKE, which both rely on adders.

III. Hardware Interface

One important aspect of hardware architectures is the interface. Especially for compact implementations, the interface may have a major impact on the overall area. Thus all algorithms were implemented using the same interface.

This interface is compliant to the Fast Simplex Link (FSL) specification (cf. [20]). The FSL is a popular method to connect IP cores to microprocessors, e.g. the Xilinx Microblaze softcore processor. The FSL is a generic 32 bit wide unidirectional link with an optional FIFO. Two synchronous links form the bidirectional interface of all our implementations (see Tab. I).

Signal Name	I/O	Description
FSL_Clk	I	FSL Clock for synchronous FIFO mode
FSL_Rst	I	Peripheral reset
FSL_M_Data	O	Master input data (32 bits)
FSL_M_Write	O	Master writes data to the FIFO
FSL_M_Full	I	Master FIFO is full
FSL_S_Data	I	Slave output data (32 bits)
FSL_S_Read	O	Slave reads data from the FIFO
FSL_S_Exists	I	Data exists in the slave FIFO

Tab. I: Implemented I/Os of the FSL interface.

The incoming link (the hash function is the slave) is utilized to transfer the input message to the hash function in message blocks of exactly 512 (or 1088) bit. The bit and byte ordering is implemented according to the NIST specification. Hence, some effort in terms of area had to be spent to reorder the bits of the last byte in the case of Keccak, if the message length is not a multiple of 8 bits. All algorithms start the computation of the round function as soon as a complete message block has been transfered.

We implemented a streaming interface where each message block comes along with a length information. Thus, the total length of the input message does not need to be known beforehand and is only limited by the design of the hash functions, which is according to the NIST requirements at least 2^{64} bit.

- First, the total length of the actual message block is transfered as a 10 (or 11) bit vector. Additionally, the 11th (or 12th) bit is used to signal the end of the input message to handle the case when the last message block is exactly 512 (or 1088) bit long. If the number of bits is less than 512 (or 1088) in the last message block, nonetheless a complete message block with 512 (or 1088) bit has to be transfered. Thus this last message block will be filled with 0s.
- Then, the message block is sent in 32 bit blocks over the link, with a total number of 16 or 34 blocks.
- On the receiving side, the padding rule of the respective algorithm is applied. For the bit positions where the padding would pad the last transfered message block with 0s, the implementations skip the padding with 0s and assume that it is filled with 0s as mentioned above to save additional multiplexers.

The output is handled similarly using the other link (the hash function acts as the master) without sending the length information as this is known in advance.

For the area and speed measurements, only the implementation of control logic for the FSL is included. The FSL implementation itself is not included, because it is configurable (e.g. the size and implementation style of the FIFO) and thus the area and speed of the FSL link varies depending on the requirements of the application.

IV. Implementations

A. BLAKE-256

BLAKE consists of eight almost identical functions $G_0, \ldots, G_7$. Each one operates on 128 bit of the BLAKE's 512 bit state. Furthermore $G_4, \ldots, G_7$ depend on the output of $G_0, \ldots, G_3$. Thus four G_i functions can be theoretically computed in parallel. The present design (Fig. 1) uses some of the properties common to all G_i functions, to achieve a area-efficient design with reasonable throughput:

- Implementing one half of a G_i-function.
- Pipelining of the G_i function.
- Executing the G_i functions in the order $0, 1, 2, 3, 7, 4, 5, 6$. This ensures, that the pipeline never stalls (cf. [13]).

The first idea can be applied, because each G_i function consists of two almost identical halves, each computing two 32 bit additions, two 32 bit XORs and two 32 bit rotations and they differ only in the rotation constants.

If the input to each G_i function would be 128 bit wide and all 128 bits of the input would become available at the same clock cycle, pipelining the G_i function would be inefficient because we would have pipeline stalls. Instead the inputs to each half round function become consecutively available in 32 bit blocks.

For most LUT-based FPGAs the additional registers do not require a lot more area, because they can usually be mapped together with the logic in the same slice. At the same time, the pipelined computation of the complete compression function does not need fundamentally more clock cycles, while the clock frequency will be higher. Thus, the throughput-area ratio increases. This pipeline is only efficient, combined with the third idea mentioned above. That is the G_7 function has

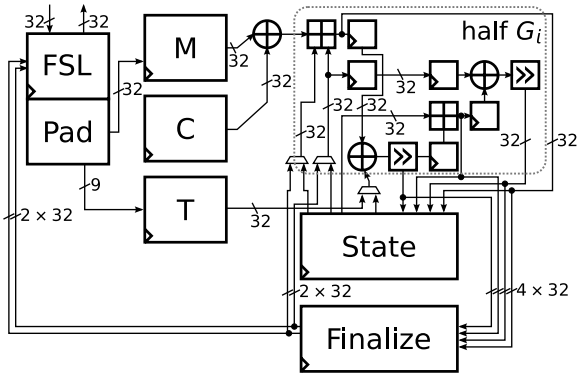

Fig. 1: The BLAKE architecture.

to be computed before G_4. Otherwise, we would run into pipeline stalls, again.

Another important feature of the design is its usage of distributed RAMs for the input message including double-buffering (M, 32×32), the round constants (C, 16×32), the state of BLAKE ($4 \times 4 \times 32$) and the chaining value used in by the finalization (8×32). The message length counter (T) is however implemented with 64 registers.

Overall, the compression function needs a moderate number of clock cycles:

- For every G_i function evaluation, it takes 2 clock cycles and thus, to compute a whole round we need 16 clock cycles.
- The round function is executed 14 times.
- Continuing with the next execution of the compression function is only possible 4 cycles later, due to the finalization after each compression function invocation.

Thus, each computation of the compression function takes 228 clock cycles.

B. Grøstl-256

The general idea to implement Grøstl in a lightweight manner is to fold the computation of a complete round into eight smaller parts. Thus only one eighth of the original round function has to be fitted into the design, at the expense of an eightfold increase of clock cycles necessary for the computation of the compression function. The compression function is designed very similar to AES and thus, a compact implementation may benefit from similar optimizations.

The implementation consists of three main details (Fig. 2):

- Usage of distributed RAM.
- An implicit `ShiftBytes` transformation.
- Pipelining of the round transformation.

We can use eight 8×8 distributed RAMs for the whole 512 bit state. For the Grøstl hash function, two such memories are necessary, one for each permutation P and Q. Both RAMs consist of eight individual RAMs representing the rows of the state matrix. The usage of the distributed RAM

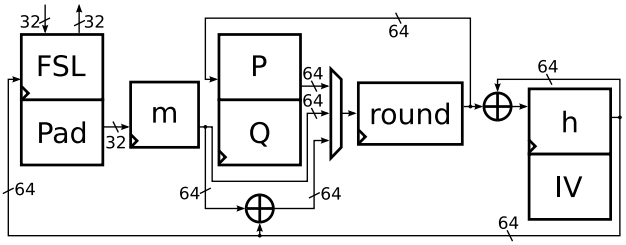

Fig. 2: The Grøstl architecture.

makes it possible to implement the `ShiftBytes` sub-transformation implicitly, by calculating appropriate read addresses. Furthermore both RAMs can be integrated into a single bigger 16×8 RAM, because it is possible to read and write alternately to the RAMs. This is a very important improvement over the earlier implementation reported in [21]. Furthermore a 8×8 RAM is needed for the storage of the intermediate output h of the compression function, which is very similar to the other memory.

The last important part of the optimization is the pipelining of the Grøstl round transformation (Fig. 3). In addition to the speed-up, we gain additional area savings. This is only possible, if we add enough pipeline stages, to store the complete internal state in the pipeline, before the first part of the computation is completed. Otherwise an additional round counter would be required, which would be used as offset to the read and write addresses (cf. [22]).

The optimization is similar to the one proposed for AES in [17]. The main difference is the removal of the second memory necessary for the proposed AES implementation, which results in a significant additional area reduction for Grøstl due to its large internal state.

The S-box is based on finite field arithmetic, which is used to calculate each value on-the-fly instead of using a lookup table in distributed RAM. The basic idea is a change of the representation of each finite field element to a computationally more efficient one (cf. [16]). This change works, because all finite fields with the same cardinality are isomorphic. In addition to the area saved by this implementation style, it is possible to insert the pipeline registers in this S-Box implementation more easily than in a design based on lookup tables.

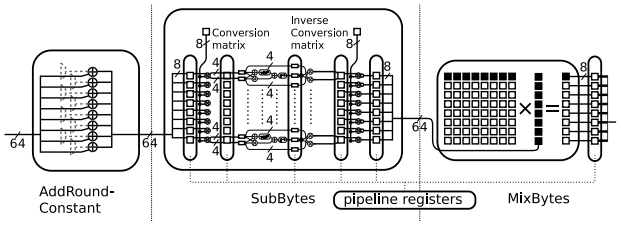

Fig. 3: Pipelining the Grøstl round function.

The performance of this architecture is quite good, because only 160 clock cycles are needed for a complete computation of the compression function (8 clock cycles per round for P and Q, 10 rounds and thus $8 \times 2 \times 10 = 160$).

C. JH-256

JH's design uses an internal state with 1024 bit and computes a very simple round function consisting of 256 4 bit S-boxes, 128 linear transformations on 8 bit each and a permutation layer, which shuffles the bits of the state in 4 bit blocks. The design can be easily folded (cf. [10]) to allow for a very compact implementation (Fig. 4). The logic in JH's round function is very small, thus pipelining does not increase the clock frequency very much. Unfortunately, due to the high number of rounds, the absolute throughput of a very compact JH implementation with an 8 bit wide data bus is quite low.

The current design uses distributed RAM to store the input, the internal state and the round constants. Additionally a RAM is used as a buffer to store the message after its initial usage for the message injection after the compression function completes all rounds.

JH has other interesting features, which are addressed in the design. The positive feature is the generation of the round constants, which are computed exactly in the same way the normal computation on JH's state is performed. That means the same logic can be used to compute the JH round as well as the round constants. This shared core of the JH architecture consists of two S-boxes and one linear transformation (L).

The less positive feature of JH for this kind of low-area design is the grouping and de-grouping, which reorder the bits of the input and output, respectively. These two functions are covered by the input and the output RAMs, which therefore are bigger than necessary for the required capacity.

The JH permutation is easily achieved by writing to the state RAM according to the specification of the permutation.

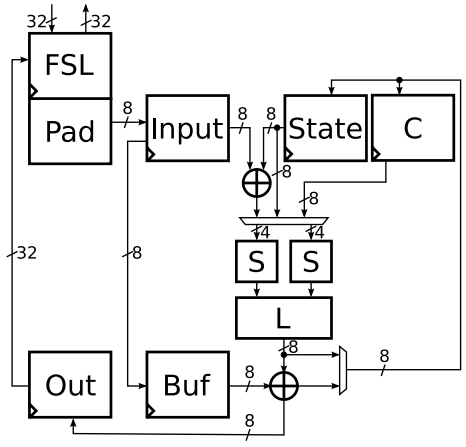

Fig. 4: The JH architecture.

For the S-boxes and the linear transformation, we used the Boolean expressions presented in [7].

The design needs at least 6400 clock cycles to compute the compression function completely (128 bytes state, 32 bytes constants and 40 rounds, thus $(128 + 32) \times 40 = 6400$), and is therefore very slow compared to the other implementations.

D. Keccak-256

The state of Keccak can be represented as a three-dimensional state $a[x][y][z]$, with $0 \leq x \leq 4, 0 \leq y \leq 4, 0 \leq z \leq 63$. Thus, the complete state consists of 1600 bit. To describe parts of the state, the following conventions of the authors of Keccak are used:

- A part of a state along the z-axis is called a lane.
- A two-dimensional part with fixed z is called a slice.

The Keccak hash function uses five functions θ,ρ,π,χ and ι, which are consecutively computed each round. 24 of these rounds are computed for Keccak-256. The functions θ,χ and ι use a number of XORs, ANDs and NOTs, while ρ and π are permutations which only reorder the state.

An external message is mapped along the lanes, that means, the first 64 bits are mapped to the lane with $x = 0, y = 0$, the second to $x = 1, y = 0$ and so on. Therefore it is easy to see how to implement Keccak by computing the five functions iteratively on each lane (e.g. [11]), but this implementation technique is inefficient.

The present implementation instead computes the Keccak permutation on eight slices in parallel (Fig. 5). The implementation uses three key ideas:

- The state is stored in 25 8×8 distributed RAMs.
- The ρ permutation can be implemented with the help of additional registers.
- The round function has to be rescheduled.

The first two ideas play nicely together. The reason for the choice of the RAM-layout is the ρ permutation. We cannot store the state in a 200×8 distributed RAM, because the ρ function rotates the bits on each lane with a different rotation constant. Furthermore, the rotations are not aligned on 8 bit, therefore we cannot store the output directly to a single 8 bit memory cell. Instead, we split the writes of a byte according to the ρ permutation, e.g. for $x = 0$ and $y = 1$, the lane is

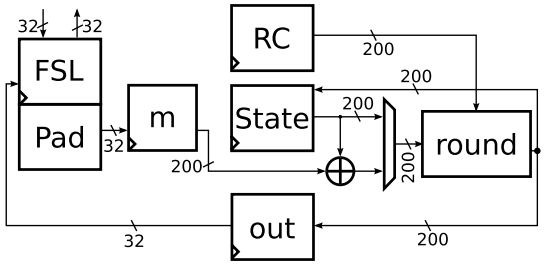

Fig. 5: The Keccak architecture.

rotated by 36 bits, and therefore the first 4 bits of the first byte are written to address $\lfloor \frac{36}{8} \rfloor$ and the second half to the next memory cell, together with the lower half of the next byte. Therefore, we have to use additional registers to store the intermediate values.

Furthermore, this architecture requires that the Keccak round function is rescheduled, adding an artificial 25th round. In the first round we execute $R_1 = \pi \circ \rho \circ \theta$, in round second and all following rounds except the last one, we execute $R_i = \pi \circ \rho \circ \theta \circ \iota \circ \chi$. The last round consists only of $R_{25} = \iota \circ \chi$.

Since we are computing 200 bit per sub-round and each sub-round takes exactly one clock cycle, a complete round is computed in 8 clock cycles. The Keccak implementation performs 25 of these rounds, therefore it takes 200 clock cycles for a complete compression function invocation. Note, that Keccak uses message blocks of 1088 bits and thus it computes more than twice as many input bit per compression function call, than the implementations of the other candidates.

E. Skein-256

Skein is an ARX-based design. That means it uses addition, rotation and XOR for its round function. Each addition, rotation and XOR works on 64 bit of Skein's 512 bit state. Therefore it is very natural to use a 64 bit wide data-path throughout the implementation (Fig. 6).

As in all the other designs, the state (16×64) is stored in distributed RAM, which has 1024 bit, to write the output of the round function back to the RAM while still reading the current state. Furthermore, the input is buffered in an additional RAM (8×64) for the second message injection after the computation of all rounds, to allow the loading of a new message block while the computation using the current message block is still running. Additionally, the key schedule uses a 9×64 RAM to store the keys.

Efficient implementations of Skein for FPGAs are quite a challenge, which is mainly caused by the 64 bit adders and further complicated by the rotations which impact the routing on an FPGA device. Together both features have a significant impact on the maximum achievable clock frequency. Pipelining the round function is the obvious

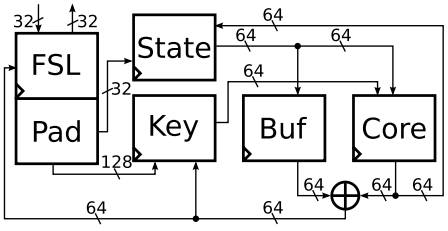

Fig. 6: The Skein architecture.

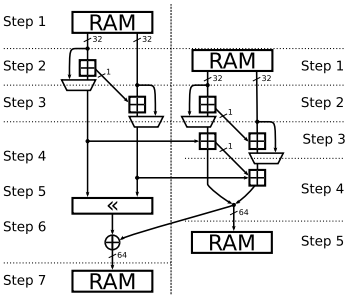

Fig. 7: Pipelining the Skein round function.

countermeasure, but this is not as easy as for other hash functions like Grøstl.

The pipeline itself consists of two distinct parts, each using 64 bit as input (Fig. 7). The 3 necessary 64 bit adders are split into 32 bit adders and used in a way, that only 3 of them are necessary. The rotation and the last XOR are distributed over 3 clock cycles, which eases the burden on the placement and routing tools. The two parts have two different lengths, such that the pipeline never stalls. This is caused by the permutation of Skein, which makes it complicated to find a good strategy for pipelining the round function.

The performance of this design is dominated by the large number of rounds required. Overall the architecture requires 584 clock cycles for one execution of the compression function (72 rounds + 1 extra round for the last key injection and 8 clock cycles for each round, thus $73 \times 8 = 584$).

V. Evaluation

We have implemented compact designs of all SHA-3 finalists and generated post place and route results for Virtex-5 FPGAs and a 256 bit message digest. The search for optimal options and timing constraints was automated by a set of scripts, brute forcing a high number of options, similar to ATHENa (cf. [**?**]).

The numbers of the throughput, tp, and the throughput-area ratio, $tp\text{-}area$ are calculated by the following formula, where p is the clock period, b is the block size, $cycles$ is the number of clock cycles for the round transformation and $area$ is the number of used slices.

$$\text{tp} = \frac{b}{p \times cycles}$$

$$\text{tp-area} = \frac{tp}{area}$$

The five implementations give very different post place and route results (Tab. II). Ranking the implementations by their throughput-area ratio, Grøstl is fighting with Keccak for the first place, with Grøstl winning clearly. The two algorithms are followed by BLAKE, which is still very competitive. Skein and JH are the weakest performers. It looks very different, if we rank the implementations by area alone:

Algorithm	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
Grøstl	368	0	305	975	2.64
Keccak	393	0	159	864	2.19
BLAKE	251	0	211	477	1.90
Skein	519	0	299	262	0.50
JH	193	0	283	23	0.11

Tab. II: Implementation results for Virtex-5 FPGAs.

Algorithm	Slices	BRAM	MHz	MBit/s	MBit/s/Slice
Grøstl	293	0	330	960	3.27
Keccak	188	0	285	145	0.77
BLAKE	175	0	347	132	0.75
Skein	291	0	200	223	0.76
JH	304	0	299	222	0.73

Tab. III: Third party results for Virtex-6 FPGAs by [11].

JH is the winner, closely followed by BLAKE. Grøstl and Keccak are close together, while Skein follows a little bit behind on the last place.

Digging a little deeper into why JH is that slow quickly reveals the choice of the 8 bit data path as the main problem. However, the results given in [11] indicate that an implementation with a much broader data path will require more area, while still being slower than Grøstl and Keccak. Thus an improvement was not pursued further for the time being.

For Skein, which does not suffer from a low clock frequency and which uses only a small number of cycles per round, the high number of rounds kills the performance. It is not as clear as for JH how to solve this problem. However, the results from [11] show, that at least for Virtex-6 FPGAs, the area required by the implementation could be smaller.

The picture gets more complete, if we compare the new results to the previous implementations from [11] (Tab. III). They are for Virtex-6 FPGAs and therefore not directly comparable, but the overall ranking can be compared. From their results, we took the timing results for the 256 bit digest results. Their area-optimized results show a very similar trend. The only similarities between our results and the results from [11] are the quite good performance of Grøstl and the mediocre results of Skein. While our JH implementation is obviously worse, we show, that Keccak and BLAKE can be quite competitive and the dominance of Grøstl is not that obvious than hinted by the previous results. It may even be possible to re-design our BLAKE implementation. This re-design should use more area, comparable to the Grøstl and Keccak designs and thus might significantly improving the throughput. Therefore the throughput-area ratio could be similar to the results of Grøstl and Keccak.

One other obvious difference between the two comparative studies is, that the results from [11] are almost consistently smaller. This fact is not only due to differences in the designs, but can also be attributed to the padding function which we deliberately included for all candidates and which is missing in the other results. Earlier results show, that the padding function can add up to 20% to the overall area (cf. [21]).

VI. Conclusion

The present paper focuses on area-efficient FPGA implementations of the SHA-3 finalists. One optimized implementation of each candidate was designed and evaluated. The performance of Grøstl is considered the best alongside Keccak in terms of the throughput-area ratio, BLAKE follows closely, while Skein and JH trail behind. If the focus lies on pure area consumption, the situation reverses and it is much easier to implement a really small JH or BLAKE design. Keccak, Grøstl and Skein are much bigger.

There is still room for improvements of almost all implementations. For example, the area of Keccak could probably be further reduced by making a design which uses only 4 slices in parallel and JH could be much faster, if more parallelism is used.

Acknowledgment

We would like to thank Steffen Reith for his help and insightful comments on various aspects of this work. This research was supported in part by BMBF grant 17N1308.

References

[1] R. F. Kayser, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," in *Federal Register*. National Institute of Standards and Technology, November 2007, vol. 72, pp. 62 212–62 220.

[2] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *Proceedings of Crypto*, ser. Lecture notes in computer science, vol. 3621. Springer, 2005, pp. 17–36.

[3] S. Sanadhya and P. Sarkar, "New collision attacks against up to 24-step SHA-2," in *Progress in Cryptology-INDOCRYPT*, ser. Lecture notes in computer science, vol. 5365. Springer, 2008.

[4] T. Isobe and K. Shibutani, "Preimage attacks on reduced Tiger and SHA-2," in *Fast Software Encryption*, ser. Lecture notes in computer science, vol. 5665. Springer, 2009.

[5] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE," Submission to NIST, 2010. [Online]. Available: http://www.131002.net/blake/blake.pdf

[6] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate," Submission to NIST, 2010. [Online]. Available: http://groestl.info/Groestl.pdf

[7] H. Wu, "The Hash Function JH," Submission to NIST, 2011. [Online]. Available: http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf

[8] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "The Keccak SHA-3 submission," Submission to NIST, 2011. [Online]. Available: http://keccak.noekeon.org/Keccak-submission-3.pdf

[9] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein Hash Function Family," Submission to NIST, 2010. [Online]. Available: http://www.skein-hash.info/sites/default/files/skein1.3.pdf

[10] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs," *Ecrypt II Hash Workshop*, 2011.

[11] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale, and F.-X. Standaert, "Compact FPGA Implementations of the Five SHA-3 Finalists," *Ecrypt II Hash Workshop*, 2011.

[12] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O'Neill, and W. Marnane, "FPGA Implementations of the Round Two SHA-3 Candidates," The second SHA-3 Candidate Conference, 2010.

[13] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "Compact implementations of BLAKE-32 and BLAKE-64 on FPGA," in *FPT*, 2010, pp. 170–177.

[14] B. Jungk, "Compact implementations of Grøstl, JH and Skein for FPGAs," *Ecrypt II Hash Workshop*, 2011.

[15] D. Canright and D. A. Osvik, "A More Compact AES," *Selected Areas in Cryptography: 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, pp. 157–169, 2009.

[16] D. Canright, "A Very Compact S-Box for AES," in *Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2005, pp. 441–455.

[17] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2003, pp. 319–333.

[18] N. Pramstaller, S. Mangard, S. Dominikus, and J. Wolkerstorfer, "Efficient AES Implementations on ASICs and FPGAs," in *Advanced Encryption Standard – AES*. Springer-Verlag, 2005, pp. 98–112.

[19] S. Xing and W. W. h. Yu, "FPGA Adders: Performance Evaluation and Optimal Design," *IEEE Des. Test*, vol. 15, pp. 24–29, 1998.

[20] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*, Xilinx, 2010.

[21] B. Jungk and S. Reith, "On FPGA-Based Implementations of the SHA-3 Candidate Grøstl," *International Conference on Reconfigurable Computing and FPGAs 2011*, pp. 316–321, 2010.

[22] B. Jungk and S. Reith, "On FPGA-based implementations of Grøstl," Cryptology ePrint Archive, Report 2010/260, 2010.

[23] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, ser. FPL '10. IEEE Computer Society, 2010, pp. 414–421.