Evaluation Of Compact FPGA Implementations For All SHA-3 Finalists

Bernhard Jungk
bernhard.jungk@hs-rm.de

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

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 evalute the finalists of the SHA-3 competition, started by the National Institute of Standards and Technology (NIST). This work adds new valuable data to the competition, by providing architectures for compact implementations of all finalists.
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 is the best overall performer for compact implementations, if the throughput-area ratio is most important. The following candidates are JH, Keccak and BLAKE, which are close together, while the current Skein architecture trails behind. The area ranking changes the results and puts JH on the top, followed by one of the BLAKE implementations, Grøstl, Keccak and Skein.

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

1 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]).

One of the other groups we know of, which published compact designs of all finalists, reported their results only for Virtex-6 and Spartan-6 FPGAs (cf. [11]). The other group implemented their designs for more architectures but in contrast to the results presented in [11] and our results, the results include block RAM. Therefore both earlier reports are difficult to compare to our results. Nonetheless we think, that our work adds valuable input to the SHA-3 competition: The throughput-area ratio of our implementations ranks the candidates for Virtex-5 devices in

the following order: Grøstl, JH, Keccak, BLAKE, Skein. If we disregard the throughput and look only on the area, the ranking changes a little bit. A different JH implementation is now on the top, followed by BLAKE, Grøstl and Keccak. Skein is, as before, the last candidate in this list. Recent results can usually be found in the ATHENa database[1].

The rest of the paper discusses previous and related work on FPGA implementations in Section 2 and then continues with a description of the hardware interface in Section 3. Then, in Section 4 each design is described in detail, followed by the evaluation of the implementation results (Section 5) and our conclusion (Section 6).

2 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. 2).

Results on compact implementations of Grøstl, Skein and JH were reported in [14], followed up by [15]. The present work improves on these results by supplying new implementations of JH and BLAKE, which take more or less the same number of slices than Grøstl and Keccak. Furthermore the present work supplies numbers for other Xilinx devices.

The first study of compact implementations for all SHA-3 finalists 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). These results were followed by a comparative study by Kaps et al. (cf. [16]). Compared to the present work, they use block RAM and thus their results differ significantly from our results.

3 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. [17]). 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. 1).

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

[1] http://cryptography.gmu.edu/athenadb/table_view

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

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.

4 Implementations

4.1 BLAKE-256

For BLAKE, we implemented two ideas, because the original idea was much smaller compared to Grøstl and Keccak. BLAKE consists of eight almost identical functions $G_0, \ldots, G_7$. Each one operates on 128 bit of 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 often 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 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 in this implementation 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.

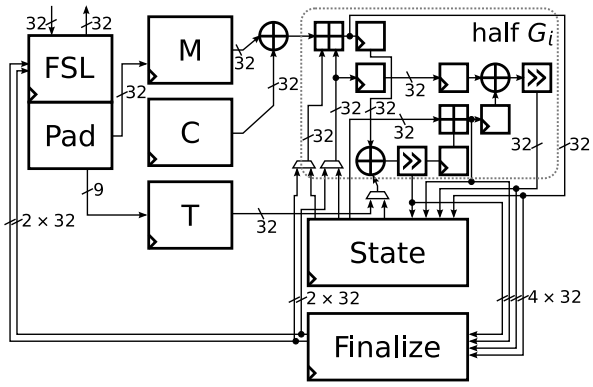

Fig. 1: The first BLAKE architecture.

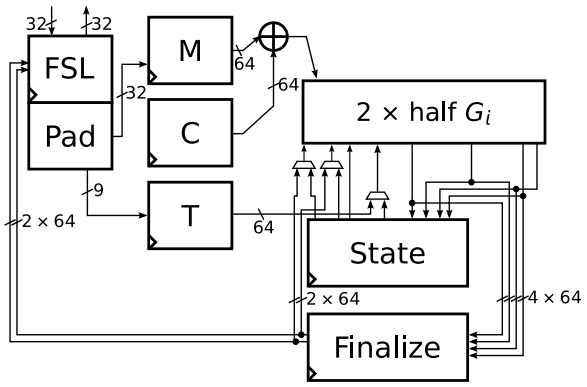

Fig. 2: The second BLAKE architecture.

- 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.

The second and larger implementation is based on the previously described ideas and doubles the number of half G_i implementations. Basically there are two ways to do this:

1. Use a full G_i function
2. Operate two half G_i functions in parallel

The first option was not implemented, because it is difficult to implement the full G_i function using a pipelined core without additional wait cycles. The second idea is straightforward, using a pipeline with half the depth (Fig. 2). While it is quite easy to evolve the first architecture into the second one, there are three small issues which have to be dealt with:

- We need additional multiplexers for the input or the output of the state RAM to deal with the different input patterns between $G_{0,1,2,3}$ and $G_{4,5,6,7}$.
- The first implementation of finalize stored intermediate inputs in distributed RAM. Now, the inputs are available earlier and in parallel, thus we need registers in addition to the RAM.
- We need 256 binary XORs compared to 64 ternary XORs in the smaller implementation, if we want to process all inputs in parallel. One additional clock cycle removes one half of the XORs.

The previous analysis on the number of clock cycles applies, but halved. The exception is the one additional clock cycle for the finalization. Thus the new implementation uses 115 clock cycles.

4.2 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

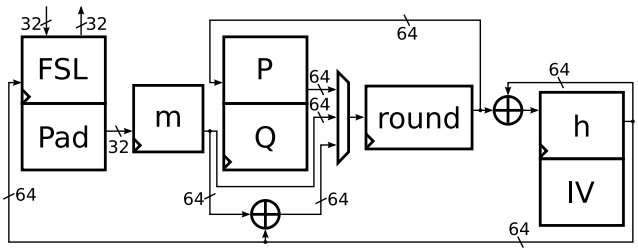

Fig. 3: The Grøstl architecture.

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. 3):

- 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 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 [18]. 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. 4). 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. [19]).

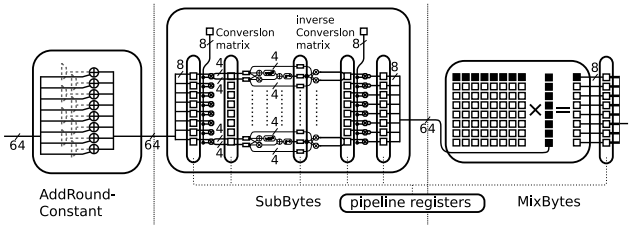

Fig. 4: Pipelining the Grøstl round function.

The optimization is similar to the one proposed for AES in [20]. 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. [21]). 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.

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 $8 \times 2 \times 10 = 160$).

4.3 JH-256

Similar to the two designs for BLAKE-256, we have two implementations for JH-256. While both designs of BLAKE-256 are quite similar, the faster JH-256 design is a more thorough redesign. The first 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. 5). 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 path 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.

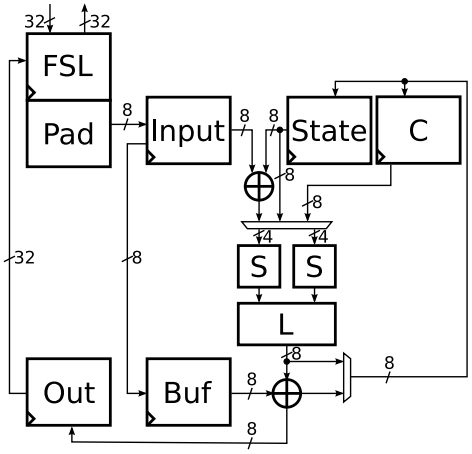

Fig. 5: The 8 bit JH architecture.

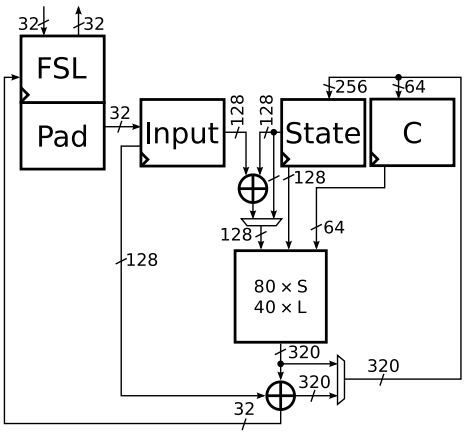

Fig. 6: The improved 320 bit JH architecture.

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

The first design needs at least 6720 clock cycles to compute the compression function completely (128 bytes state, 32 bytes constants and 42 rounds, thus $(128 + 32) \times 42 = 6720$), and is therefore very slow compared to the other implementations.

The second design expands the datapath from 8 bit to 320 bit and thus reduces the number of clock cycles per round by a large factor (Fig. 6), while the design itself stays resonably small. Compared to the previous design the following additional changes have been made:

- The grouping of the input and the buffering for the XOR happening later is basically the same with a wider datapath.
- The degrouping of the final hash value is no longer required, because the all 32 bit transfered over the FSL link available in one clock cycle.
- The S-Boxes and linear tranformations are implemented by manually instantiating `LUT6_2` instances. Of course, this does not work on older devices like the Xilinx Spartan-3.

All in all, the design needs 97,5% less clock cycles for the complete round function (168 cycles). This can be easily calculated, because the datapath is 40 times bigger than before.

4.4 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. 7). 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 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 $R_{25} = \iota \circ \chi$.

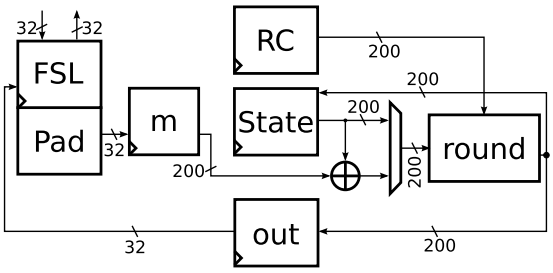

Fig. 7: The Keccak architecture.

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.

4.5 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. 8).

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

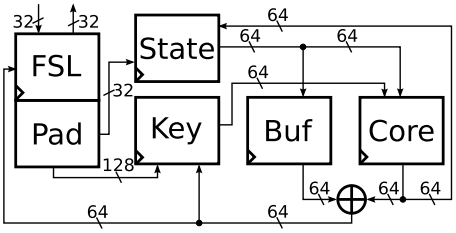

Fig. 8: The Skein architecture.

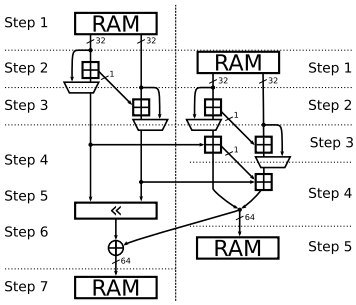

Fig. 9: Pipelining the Skein round function.

The pipeline itself consists of two distinct parts, each using 64 bit as input (Fig. 9). 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 $73 \times 8 = 584$).

5 Evaluation

We have implemented compact designs of all SHA-3 finalists and generated post place and route results for Virtex-5 FPGAs and 256 bit message digests. The search for optimal options and timing constraints was automated by a custom evaluation tool, similar to ATHENa (cf. [22]).

The numbers of the throughput, *tp*, and the throughput-area ratio, *tp-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.

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

The post-place and route results for the 7 implementations are shown in Tab. 2 (More results in Appendix A). Ranking the implementations by their throughput-area ratio, Grøstl is clearly wins, while JH-1, Keccak, BLAKE-1 and BLAKE-2 are close together. Skein and JH-1 are the weakest performers. The view changes, if we rank the implementations by area alone. Then JH-1 is the winner, followed by the BLAKE-1 implementation. Grøstl, Keccak-1, BLAKE-2 and JH-2 have very similar area requirements, while our current Skein design uses many more Slices and is thus on the last place.

The JH-1 design is obviously slow because of the 8 bit data path and the large number of clock cycles per round and thus an implementation with a wider data path (JH-2) is a lot faster. Skein is also slow, but it neither suffers from a low clock frequency nor a large number of clock

Algorithm	**Slices**	**BRAM**	**MHz**	**MBit/s**	**MBit/s/Slice**
Grøstl	368	0	305	975	2.64
Keccak-1	393	0	159	864	2.19
(Keccak-2	379	0	159	864	2.29)
BLAKE-1	251	0	211	477	1.90
BLAKE-2	374	0	163	725	1.94
Skein	519	0	299	262	0.50
JH-1	193	0	283	21.5	0.11
JH-2	377	0	278	847	2.24

Tab. 2: 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. 3: Third party results for Virtex-6 FPGAs by [11].

cycles per round. Instead the highly iterative nature of Skein kills the performance. It is not as clear as for JH how to solve this problem, because it is not easily to see how to decrease the number of clock cycles without increasing the area by a large margin. However, the results from [11] show, that at least for Virtex-6 FPGAs, the area required by an implementation can be less.

Three SHA-3 candidates have two results each in Tab. 2: Keccak and BLAKE and JH. The idea behind the Keccak-2 result was to remove the bit reordering of the input required by the Keccak specification to investigate how much area is required by this irregularity in the design. Each input bit needs an additional multiplexer, and thus the result shows, that this is not very significant. This is not very suprising, because only 32 input bits are processed in each clock cycle.

For BLAKE, the second and larger implementation was pursued because of the large area gap between the implementations BLAKE-1, Grøstl and Keccak. As we can see, the throughput-area of BLAKE stays roughly the same and thus the ranking stays the same. A similar reason applies for implementing JH a second time, which gave much better results compared to the JH-1 implementation.

The picture gets more complete, if we compare the new results to the previous implementations from [11] (Tab. 3). They are for Virtex-6 FPGAs and therefore they are not directly comparable to the Virtex-5 results. From their results, we took the values which were optimized for timing and for the 256 bit digest results. 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-1 implementation is obviously worse, we show, that JH, Keccak and BLAKE can be quite competitive and the dominance of Grøstl is not that obvious than hinted by the previous results.

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. [18]).

6 Conclusion

The present paper focuses on area-efficient FPGA implementations of the SHA-3 finalists. At least one optimized implementation of each candidate was designed and evaluated. The throughput-area ratio of Grøstl is the best and at least on of the implementations for JH, Keccak, BLAKE follows with little distance while Skein trails behind after a large gap. If the focus is the area consumption, the situation is different. It is much easier to implement JH really small.

One BLAKE design is also quite small, while all other implementations are in the 350-400 slices range, except Skein, which is quite large.

There is still room for improvements and developments of all implementations. For example, the area of Keccak can be further reduced by making a design which uses only 4 or 2 slices in parallel. Furthermore the second BLAKE implementation could be designed smaller, but probably slower, using one full G_i instead of two halves running in parallel. However, the most important next step is to improve the Skein implementation, which can certainly be implemented in the 350-400 slices range but probably without significantly improving the throughput. Therefore Skein is likely to be the worst of all finalists for compact FPGA implementations.

Acknowledgment

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

A Results for Spartan-3, Spartan-6 and Virtex-6 FPGAs

All designs except the new JH implementation are realized in device-independent VHDL code, thus they can be synthesized for the other Xilinx targets. The JH implementation for Spartan-3 FPGAs uses the logic as proposed as Boolean formula in the submission of JH. The results in this appendix are for other Xilinx FPGA families, namely Spartan-3 (Tab. 4), Spartan-6 (Tab. 5) and Virtex-6 (Tab. 6). The overall picture stays roughly the same for these devices. The only exception is the improved JH implementation for Spartan-3 FPGAs, which is on the fifth place in the throughput-area ratio ranking, while it is the second best design on the other

Algorithm	**Slices**	**MHz**	**MBit/s**	$\frac{\textbf{MBit/s}}{\textbf{Slice}}$
BLAKE-1	948	88.6	198	0.20
BLAKE-2	1716	71.6	318	0.18
Grøstl	1220	148	473	0.38
JH-1	807	124	9.4	0.01
JH-2	2060	113	344	0.16
Keccak	1665	71.2	387	0.23
Skein	1347	128	112	0.08

Tab. 4: Implementation results for Spartan-3 FPGAs.

Algorithm	**Slices**	**MHz**	**MBit/s**	$\frac{\textbf{MBit/s}}{\textbf{Slice}}$
BLAKE-1	257	155	477	1.85
BLAKE-2	413	113	725	1.75
Grøstl	344	236	975	2.83
JH-1	171	241	22	0.12
JH-2	372	185	847	2.27
Keccak	420	122	864	2.05
Skein	418	210	262	0.62

Tab. 5: Implementation results for Spartan-6 FPGAs.

Algorithm	Slices	MHz	MBit/s	$\frac{\text{MBit/s}}{\text{Slice}}$
BLAKE-1	260	263	590	2.26
BLAKE-2	419	204	908	2.18
Grøstl	328	365	1168	3.56
JH-1	221	442	33	0.14
JH-2	352	344	1048	2.97
Keccak	397	197	1071	2.69
Skein	406	316	277	0.68

Tab. 6: Implementation results for Virtex-6 FPGAs.

devices. For the Spartan-3, the root cause is probably the manual instantiation of the `LUT6_2` primitive for the other devices. Nonetheless the throughput-area ratio of BLAKE, Keccak and JH is close together and Grøstl is always on the first place. Skein is always the slowest candidate and therefore has the poorest throughput-area ratio - except for the slow JH implementation.

References

1. Kayser, R.F.: Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. In: Federal Register. Volume 72. National Institute of Standards and Technology (November 2007) 62212–62220
2. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Proceedings of Crypto. Volume 3621 of Lecture notes in computer science., Springer (2005) 17–36
3. Sanadhya, S., Sarkar, P.: New collision attacks against up to 24-step SHA-2. In: Progress in Cryptology-INDOCRYPT. Volume 5365 of Lecture notes in computer science., Springer (2008)
4. Isobe, T., Shibutani, K.: Preimage attacks on reduced Tiger and SHA-2. In: Fast Software Encryption. Volume 5665 of Lecture notes in computer science., Springer (2009)
5. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to NIST (2010)
6. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (2010)
7. Wu, H.: The Hash Function JH. Submission to NIST (2011)
8. Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: The Keccak SHA-3 submission. Submission to NIST (2011)
9. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family. Submission to NIST (2010)
10. Homsirikamol, E., Rogawski, M., Gaj, K.: Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs. Ecrypt II Hash Workshop (2011)
11. Kerckhof, S., Durvaux, F., Veyrat-Charvillon, N., Regazzoni, F., de Dormale, G.M., Standaert, F.X.: Compact FPGA Implementations of the Five SHA-3 Finalists. Ecrypt II Hash Workshop (2011)
12. Baldwin, B., Hanley, N., Hamilton, M., Lu, L., Byrne, A., O'Neill, M., Marnane, W.: FPGA Implementations of the Round Two SHA-3 Candidates. The second SHA-3 Candidate Conference (2010)
13. Beuchat, J.L., Okamoto, E., Yamazaki, T.: Compact implementations of BLAKE-32 and BLAKE-64 on FPGA. In: FPT. (2010) 170–177
14. Jungk, B.: Compact implementations of Grøstl, JH and Skein for FPGAs. Ecrypt II Hash Workshop (2011)
15. Jungk, B., Apfelbeck, J.: Area-Efficient FPGA Implementations of the SHA-3 Finalists. In: Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on. (2011) 235 –241
16. Kaps, J.P., Yalla, P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., Pham, J.: Lightweight implementations of SHA-3 candidates on FPGAs. In: Progress in Cryptology – INDOCRYPT 2011, Springer Berlin / Heidelberg (2011) 270–289
17. Xilinx: LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c). (2010)
18. Jungk, B., Reith, S.: On FPGA-Based Implementations of the SHA-3 Candidate Grøstl. International Conference on Reconfigurable Computing and FPGAs 2011 (2010) 316–321

19. Jungk, B., Reith, S.: On FPGA-based implementations of Grøstl. Cryptology ePrint Archive, Report 2010/260 (2010)
20. Chodowiec, P., Gaj, K.: Very compact FPGA implementation of the AES algorithm. In: Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag (2003) 319–333
21. Canright, D.: A Very Compact S-Box for AES. In: Proceedings of 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag (2005) 441–455
22. Gaj, K., Kaps, J.P., Amirineni, V., Rogawski, M., Homsirikamol, E., Brewster, B.Y.: 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. FPL '10, IEEE Computer Society (2010) 414–421