Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein

Stefan Tillich, Martin Feldhofer, Wolfgang Issovits, Thomas Kern, Hermann Kureck,
Michael Mühlberghuber, Georg Neubauer, Andreas Reiter,
Armin Köfler, and Mathias Mayrhofer
Graz University of Technology
Institute for Applied Information Processing and Communications
Inffeldgasse 16a, A–8010 Graz, Austria
{stefan.tillich,martin.feldhofer}@iaik.tugraz.at,
{wolfgang.issovits,t.kern,hermann.kureck,m.muehlberghuber}@student.tugraz.at
{georg.neubauer,andreas.reiter}@student.tugraz.at, {arminko,mtmayr}@sbox.tugraz.at

Abstract

The weakening of the widely used SHA-1 hash function has also cast doubts on the strength of the related algorithms of the SHA-2 family. The US NIST has therefore initiated the SHA-3 competition in order to select a modern hash function algorithm as a "backup" for SHA-2. This algorithm should be efficiently implementable both in software and hardware under different constraints. In this paper, we present hardware implementations of the four SHA-3 candidates ARIRANG, BLAKE, Grøstl, and Skein with the primary constraint of minimizing chip area.

1 Introduction

The basic functionality of a cryptographic hash function is to take an arbitrarily long message[1] and deliver a fixed-size output, the so-called *message digest*. Hash functions are widely used in modern security protocols and applications like digital signatures, message authentication, password protection, and pseudo-random number generation. Following cryptanalytical advances, the popular SHA-1 algorithm [9] has been seriously weakened [2, 11]. As a reaction, the US National Institute of Standards and Technology (NIST) has recommended to switch to the SHA-2 family of hash functions [9]. However, as SHA-2 is very similar in structure to SHA-1, it is feared that the discovery of serious cryptanalytic attacks on SHA-2 might only be a matter of time.

As a response to this problem, NIST has set up the SHA-3 competition [7] with the goal of identifying one (or more) modern hash functions which can act as a drop-in replacement for the SHA-2 family. Following the official call for submissions in November 2007, 64 algorithms were proposed for SHA-3 in October 2008. The tentative timeline of the competition aims at the selection of a winner in the middle of 2012 after two evaluation rounds of roughly 18 months each. As of now, about a third of the candidates has not been selected for the first evaluation round, has been withdrawn, or has been conceded as broken. The rest of the hash functions is currently undergoing scrutiny by the cryptographic community and their implementation properties are evaluated on various platforms and under various constraints.

In the domain of hardware implementations, low-area constraints are often found in small embedded systems where the cost per produced unit is of paramount importance. Additionally, such systems often have a low power budget. Prominent examples include RFID tags and cryptographic smart-cards.

In order to estimate the suitability of some promising SHA-3 candidates, we have implemented ARIRANG, BLAKE, Grøstl, and Skein in hardware with the main emphasis on the reduction of the required silicon area. The hardware modules have been evaluated in regard to their size, throughput, and energy consumption. With this work we hope to make a valuable contribution to the hardware evaluation effort for the SHA-3 competition [4].

The rest of this paper is organized as follows. Common properties of hash functions in general and of our four implementations in particular are described in Section 2. More detailed elaborations on the four implementations are given in Sections 3, 4, 5, and 6. Each of these sections includes a brief outline of the hash algorithm and the most important properties of the corresponding hardware module. In Section 7 we summarize our practical results and compare the four implementations. Conclusions are drawn in Section 8.

2 Hash Functions and Their Implementation in Hardware

In the following, we briefly describe the principal concepts used in hash functions and the general design choices we have taken for our hardware implementations.

2.1 General Properties of Hash Functions

Most of the SHA-3 candidates employ a similar concept to transform the input message to the output message di-

[1]Normally, hash functions have an upper limit for the input message size, e.g. $2^{64} - 1$ bits for SHA-256.

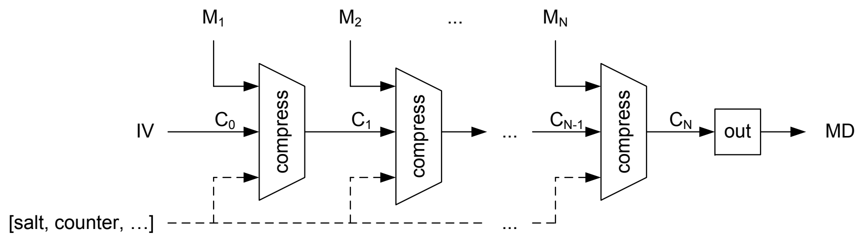

Figure 1. A hash function with sequential compression of message blocks.

gest. The message is padded to a size which is a multiple of a fixed block size and is segmented into individual message blocks. Message blocks are then processed sequentially (as in the four algorithms examined in this paper) or in a more parallel fashion (e.g. as inner nodes of a tree structure). The results of the processing of individual message block are connected together by so-called *chaining values*. The *compression function* of the hash algorithm takes the chaining value and a message block (and possibly some additional inputs) and produces an updated chaining value.

Figure 1 depicts a hash function with the compression function *compress*, which hashes the message blocks M_1 to M_N and produces the message digest MD. The chaining values are C_0 to C_N. At the beginning of the hashing, an initialization vector (IV) is used as first chaining value C_0. The compression of message block M_i yields the chaining value C_i. After all message blocks have been processed, the message digest is derived from the last chaining value C_N via the so-called *output function* (labeled *out* in Figure 1).

All SHA-3 candidates must support four message digest sizes: 224, 256, 384, and 512 bits. Most of the submissions define so-called *hash function families* which consist of separate algorithm variants for the different message digests. For example, the ARIRANG specification [3] defines the variants ARIRANG-224, ARIRANG-256, ARIRANG-384, and ARIRANG-512, where the trailing number indicates the length of the message digest. The variants usually differ only in some details, e.g. value of some constants, size of the chaining value, size of the inner state of the compression function.

2.2 Design Choices for Our Implementations

We have chosen to implement only the variants with a 256-bit message digest, i.e. ARIRANG-256, BLAKE-32, Grøstl-256, and Skein-256-256[2]. For the sake of brevity, only the implemented variants are discussed in the subsequent sections. All four implementations assume that the padding is performed externally and that only complete message blocks are supplied as input. The modules are fully self-contained, producing the message digest from the supplied message blocks without the need of any external memory. The interfaces allow connection to a microcontroller as coprocessor, e.g. in a system-on-chip implementation. After the optimization of the hardware architectures, the modules have been synthesized with the Cadence PKS shell and power has been simulated with Synopsys NanoSim.

[2]Skein-X-Y denotes the variant of Skein where X is the size of the inner state of the compression function, and Y is the size of the message digest.

3 ARIRANG

The hash function family ARIRANG has been designed by Chang *et al.* [3]. It reuses some transformations of the Advanced Encryption Standard (AES) [8].

3.1 Algorithm Description

After message padding, ARIRANG-256 initializes its 256-bit chaining value with an IV. A counter value is added to the chaining value, before it is mixed with the next message block in the compression function, yielding the next chaining value. The ARIRANG-256 compression function consists of 40 similar steps, where each step mixes in two 32-bit words from the so-called message schedule, which is an expansion of the message block. The message schedule generates 16 extra words via XORs from the 16 words of the 512-bit message block and from 16 constants.

A single step of the compression function transforms the eight 32-bit words of the intermediate state, performing eight XORs, four fixed-distance rotates and applying the so-called $G^{(256)}$ function twice. The step function is depicted in Figure 2. The $G^{(256)}$ function works on a single 32-bit word using the AES SubBytes and MixColumns transformations. After 20 steps and 40 steps, the original chaining value is XORed to the intermediate state of the compression function. The message digest is the chaining value after the processing of the last message block.

3.2 Implementation

We have implemented the low-area version of ARIRANG-256 in Verilog. The datapath is able to process a complete step of the compression function in a single clock cycle. To this end it includes two instances of the $G^{(256)}$ function, requiring eight S-boxes and two MixColumns multipliers. The S-boxes are implemented as simple hardware look-up tables by specification of the input-output relation in Verilog [10]. The 16 extra words

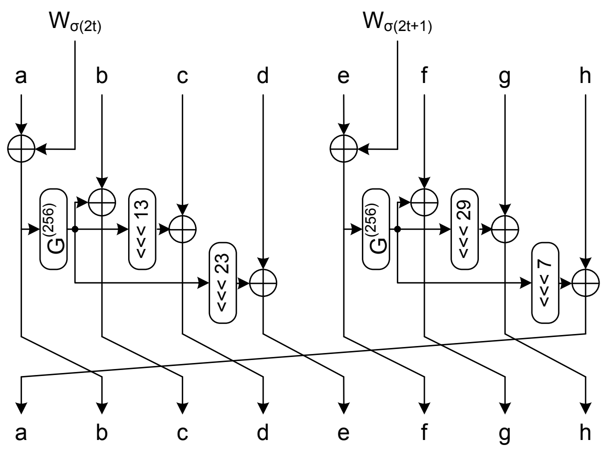

Figure 2. Step function of ARIRANG-256 for round t.

of the message schedule are generated directly from the message block and are not stored in dedicated registers.

The bulk of the sequential logic consist of registers for the 256-bit chaining value, the eight 32-bit working variables for the compression function, the 64-bit counter for the counter addition, the two 32-bit input words of the message schedule for the current step, and the 512-bit message block. A message block can be processed in only 44 clock cycles, which results in a relatively high throughput.

4 BLAKE

Aumasson *et al.* have designed the hash function family BLAKE [1]. BLAKE combines three known and analyzed concepts: The iteration mode HAIFA, the internal structure of the LAKE hash function (local wide-pipe design), and a modified version of the ChaCha stream cipher as compression function.

4.1 Algorithm Description

The BLAKE-32 compression function works on an internal state of 512 bits, represented as a (4×4)-matrix of 32-bit words. The chaining value is only half the size of the internal state. It is expanded to 512 bits in the initialization phase of the compression function, then mixed with the message block in a number of "ChaCha-like" rounds, and the result is compressed to 256 bits in the finalization phase, yielding the next chaining value. The designation as local wide-pipe design stems from the local expansion of the chaining value in the compression function.

The core of the compression function consists of the so-called *G function*. The G function takes four 32-bit words of the internal 512-bit state, combines it with two 32-bit words of the message block and two 32-bit constants and delivers four 32-bit words as result. The additional index input, ranging between 0 and 7, decides which message words and constants are used for the mixing. The G function features six additions modulo 2^{32}, six XORs and four individual word rotations by a fixed distance. Figure 3 shows the G function for index i. A single round consists of eight invocations of the G function: Four on the columns of the state and four on the diagonals of the state. Thus, in a high-performance hardware implementation, four G functions could be instantiated in parallel. A total of ten rounds is executed.

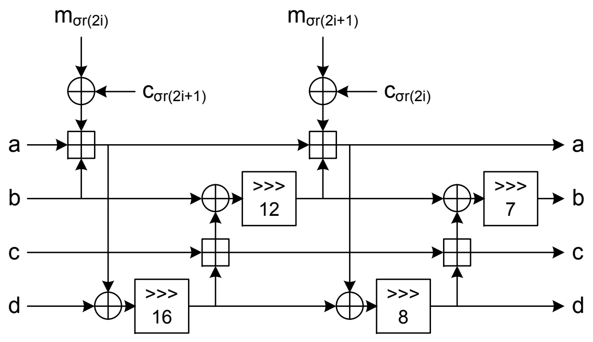

Figure 3. The G function of BLAKE-32 for index i.

The initialization and finalization parts of the compression function are rather simple and consist only of XOR operations. Initialization mixes the chaining value with a user-chosen salt, a counter of hashed bits, and some constants. The finalization takes the output of the ten rounds and combines it with the input chaining value and the salt.

4.2 Implementation

Our Verilog implementation of BLAKE-32 is centered around a 32-bit register file with a capacity of 48 words. It has four read ports and two write ports. All values required for the hashing are stored in the register file, which includes the input chaining value (8 words), the message block (16 words), the salt (4 words), the current and total bit length of the message (2 words each), and the internal state of the compression function (16 words). The permutations required for selecting message words and constants for the G function is implemented as appropriate addressing of the register file. The G function is split up into eleven individual steps, where intermediate words are calculated and stored back for subsequent use.

The processing of a 512-bit message block requires 1,038 clock cycles, including the loading of the block. The bulk of this time (970 cycles) is taken up by the ten rounds of the compression function.

5 Grøstl

Grøstl has been conceived by Gauravaram *et al.* [6]. It uses operations similar to those of AES and it is based on the wide-trail design strategy, which has already been employed in the design of AES. As Grøstl shares many features of AES, it can be expected to allow efficient implementation for a wide range of platforms and requirements.

5.1 Algorithm Description

The chaining value of Grøstl-256 and each input message block consists of eight 64-bit words. Larger message digests require an internal state and message blocks of 16 words. Two distinct permutation functions (called P and Q permutation) are defined based on "AES-like" transformations (AddRoundKey, SubBytes, ShiftBytes, and MixBytes), which are repeated in a number of similar rounds. Figure 4 depicts the processing of a single 64-bit word in a round of the P permutation, where S denotes an AES S-box. Note that the ShiftBytes transformation is done by the diagonal addressing of the state matrix. The Q permutation looks very similar, with the exception of the round constant, which is added to a different byte location.

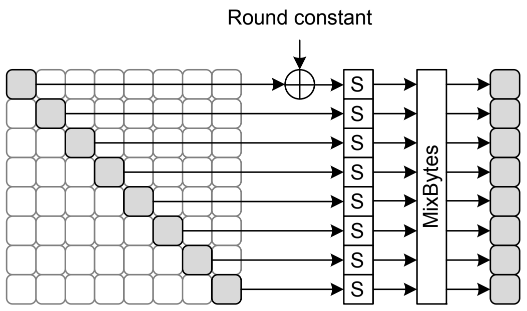

Figure 4. Processing of a 64-bit word of the internal state in a round of the Grøstl P-permutation.

The P and Q permutations are applied to two data blocks which depend on the chaining value and an input message block. The two resulting blocks are then used to update the chaining value.

5.2 Implementation

We have implemented Grøstl-256, where the P and Q permutations transform data blocks of 512 bits. Our compact hardware implementation makes use of a 64-bit datapath which processes the columns of a data block sequentially. The datapath can be switched between the functionality of the P and Q permutation and it consists of two parts: The first part comprises the transformation AddRoundConstant and most of SubBytes, while the second part includes the rest of SubBytes and the MixBytes transformation. ShiftBytes is performed in between these parts on a complete 512-bit data block by dedicated logic.

Two 512-bit blocks are processed by the datapath in an interleaved fashion: One block is piped through the first part of the datapath, while the other block is passed through the second part. Three 512-bit blocks are stored during the hashing operation: The two intermediate values of both permutations and the most recent chaining value.

6 Skein

The hash function family Skein has been developed by Ferguson *et al.* [5]. It is designed to allow fast implementation on modern 64-bit processors.

6.1 Algorithm Description

Skein is based on the tweakable block cipher[3] Threefish, with a block size of 256, 512, or 1,024 bits. The block size of Threefish equals the message block size of the input message and can be chosen independently from the size of the message digest. The Matyas-Meyer-Oseas mode is used to construct the Skein compression function from Threefish. The specification of the format of the tweak and a padding scheme complement the so-called Unique Block Iteration (UBI) chaining mode. UBI is used in Skein for message compression and as output function, but also for IV generation and other optional operation modes (e.g. tree hashing, keyed hashing).

For each Threefish block size, the size of the message digest can be set to an arbitrary value. In order to distinguish the different Skein variants, a Threefish block size of X bits together with a message digest size of Y bits is designated as Skein-X-Y. The message digest size can be changed in a hardware implementation very easily with virtually no overhead.

Threefish consists of a number of similar rounds, which feature only three simple operations: Addition modulo 2^{64}, XOR, and bit permutation. The Threefish rounds operate on the intermediate state, which is organized as a number of 64-bit words. The so-called MIX operation depicted in Figure 5 transforms two 64-bit words simultaneously. The rotation distance (rot_dist) varies with the Threefish block size, the round index and the position of the two 64-bit words in the Threefish state. A Threefish round transforms all state words with MIX and then shuffles the words with a fixed permutation. After each fourth round, a subkey is added to the state.

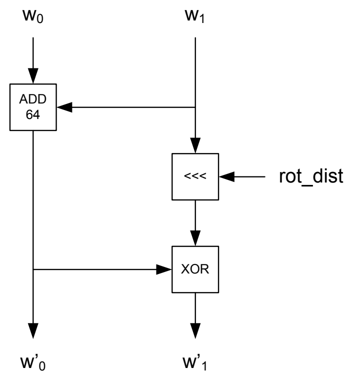

Figure 5. The Threefish MIX-operation.

Subkeys are derived from the cipher key and tweak through a simple key schedule, which also features 64-bit modulo addition. The number of rounds is 72 for Threefish block sizes of 256 and 512 bits, while Threefish-1024 has 80 rounds.

[3]In addition to the plaintext and key, a tweakable block cipher has another input (the tweak), which provides an additional degree of freedom.

Implementation	Area GEs	Latency cycles	Block size bits	Clock freq. MHz	Throughput Mbit/s	Energy mW/MHz	Energy/bit nJ/bit
ARIRANG-256	23,732	44	512	61.30	713.3	1.67	0.14
BLAKE-32	25,569	1,038	512	31.25	15.4	2.66	5.39
Grøstl-256	14,622	196	512	55.87	145.9	2.21	0.85
Skein-256-256	12,890	1,034	256	80.00	19.8	1.62	6.54

Table 1. Implementation results.

6.2 Implementation

We have implemented a fully autonomous version of Skein-256-256 in Verilog. It consists of a 64-bit datapath with a temporary 64-bit register, a register file with a capacity of 16 words, a control FSM, and a 32-bit AMBA APB interface. The 64-bit adder has been implemented in a generic fashion with Verilog's native '+' operator in order to allow the synthesizer the greatest flexibility for optimization. A Threefish MIX operation can be processed in five clock cycles. For a complete 256-bit message block, 1,021 clock cycles are required.

7 Practical Results

The four hash function designs have been implemented with the Cadence PKS shell with enabled low-power option[4] targeting a 0.35 μm standard-cell library from austriamicrosystems. Power simulation has been performed using Synopsys NanoSim, which is a so-called "near Spice" transistor-level simulator.

Table 1 lists the most important characteristics of the four hardware implementations. Silicon area is given in gate equivalents (GEs), in this case the total area (excluding pads) divided by the size of a 2-input NAND cell (NAND20). Latency gives the number of clock cycles required for processing a single message block. Block size lists the size of such a message block. The stated clock frequency is the maximal value for a supply voltage of 3.3 V under typical operating conditions. Throughput is given as maximum sustainable throughput for this clock frequency (any one-time setup and finalization cost for the hashing operation is ignored). The energy consumption is given in mW/MHz to facilitate comparison. Additionally, the energy per hashed message bit is stated.

The ARIRANG implementation is rather large, but boasts by far the highest throughput, mainly due to its small number of processing cycles per block. Additionally, its energy consumption is rather small. On the other hand, the Skein module is the smallest and requires the least energy. Despite the high clock frequency of 80 Mhz, the throughput is comparatively low. The Grøstl implementation achieves relatively good values in all categories. Our BLAKE implementation compares rather unfavorably, having the worst values for all characteristics.

[4]The low-power option enables the automatic insertion of clock-gating cells and the inclusion of sleep logic.

8 Conclusions

We have evaluated four candidate algorithms for the SHA-3 competition towards the goal of compact hardware implementation. Skein allowed for the smallest implementation with the lowest energy consumption, but with a comparatively low throughput. At a similarly low energy consumption, the ARIRANG implementation achieved a 35-times higher throughput, but required twice the silicon area. The Grøstl implementation offered a well-rounded result, being nearly as small as Skein, with a relatively high throughput and a moderate energy consumption.

References

[1] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE, version 1.3. Available online at `http://131002.net/blake/blake.pdf`, 2008.

[2] Christophe De Canniére and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.

[3] Donghoon Chang, Seokhie Hong, Changheon Kang, Jinkeon Kang, Jongsung Kim, Changhoon Lee, Jesang Lee, Jongtae Lee, Sangjin Lee, Yuseop Lee, Jongin Lim, and Jaechul Sung. ARIRANG : SHA-3 Proposal. Available online at `http://ehash.iaik.tugraz.at/uploads/2/2c/Arirang.pdf`, October 2008.

[4] ECRYPT 2. SHA-3 Hardware Implementations. `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations`.

[5] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Available online at `http://www.skein-hash.info/sites/default/files/skein1.1.pdf`, November 2008.

[6] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, and Søren S. Thomsen Martin Schläffer. Grøstl – a SHA-3 candidate. Available online at `http://www.groestl.info/Groestl.pdf`, October 2008.

[7] National Institute of Standards and Technology (NIST). Cryptographic Hash Algorithm Competition Website. `http://csrc.nist.gov/groups/ST/hash/sha-3`.

[8] National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.

[9] National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard, October 2008. Available online at `http://www.itl.nist.gov/fipspubs/`.

[10] Stefan Tillich, Martin Feldhofer, Thomas Popp, and Johann Großschädl. Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. *Journal of Signal Processing Systems*, 50(2):251–261, January 2008.

[11] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005, 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.