4

Secure Hashing: SHA-1, SHA-2, and SHA-3

Ricardo Chaves
University of Lisbon

Leonel Sousa
University of Lisbon

Nicolas Sklavos
University of Patras

Apostolos P. Fournaris
T.E.I. Western Greece

Georgina Kalogeridou
University of Cambridge

Paris Kitsos
T.E.I. Western Greece

Farhana Sheikh
Intel Corporation

CONTENTS

4.1 Introduction

A hash function is a cryptographic operation that converts a variable length input message to a fixed output called message hash or message digest. Hash functions can mostly be found in authentication schemes, random number generators, data integrity mechanisms, and digital signatures.

A modern cryptographic hash function can compute the hash value for any initial message. One of the basic hash function properties is the difficulty of finding two different messages with the same hash value (collision resistance). Due to this property, hash functions are widely used in popular security protocols like the Transport Layer Security or the Internet Protocol Security.

4.2 SHA-1 and SHA-2 Hash Functions

In 1993, the Secure Hash Standard (SHA) was first published by the NIST. In 1995, this algorithm was revised [309] in order to eliminate some of the initial weakness. The revised algorithm is usually referenced as SHA-1. In 2001, the SHA-2 hashing algorithm was proposed. The revised SHA-2 algorithm considers larger Digest Messages (DM), making it more resistant to possible attacks and allowing it to be used with larger data inputs, up to 2^{128} bits in the case of SHA512. The SHA-2 hashing algorithm is the same for the SHA224, SHA256, SHA384, and SHA512 hashing functions, differing in the size of the operands, the initialization vectors, and the size of the final DM. The next sections start by describing the round computation of the SHA-1 and SHA-2 algorithms followed by the description of the needed input data padding and expansion.

4.2.1 SHA-1 Hash Function

The SHA-1 produces a single output 160-bit message digest (the output hash value) from an input message. This input message is composed of multiple blocks. The input block, of 512 bits, is split into 80 × 32-bit words, denoted as W_t, one 32-bit word for each computational round of the SHA-1 algorithm, as depicted in Figure 4.1.

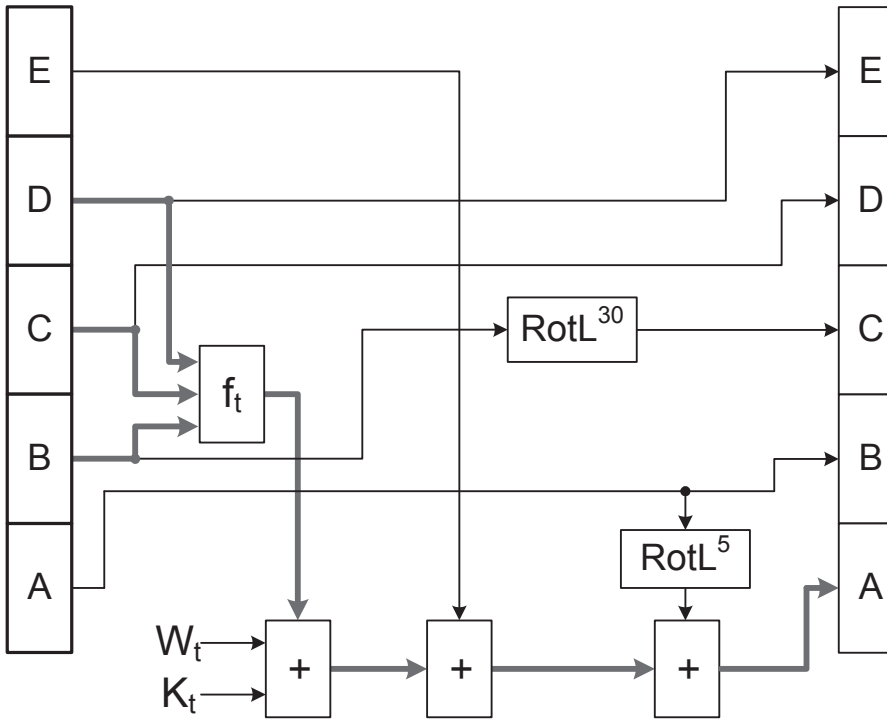

Figure 4.1
SHA-1 round calculation

Each round comprises additions and logical operations, such as bitwise logical operations (f_t) and bitwise rotations to the left ($RotL^i$). The calculation of f_t depends on the round (t) being executed, as well as the value of the round dependant constant K_t. The SHA-1 80 rounds are divided into four groups of 20 rounds, each with different values for K_t and the applied logical functions (f_t) [71].

Table 4.1 presents the values of K_t and the logical function executed, according to the round. The symbol $\wedge$ represents the bitwise AND operation and $\oplus$ represents the bitwise XOR operation.

The initial values of the A to E variables in the beginning of each data block calculation correspond to the value of the current 160-bit hash value

Table 4.1
SHA-1 logical functions

Rounds	Function	K_t
0 to 19	$(B \wedge C) \oplus (\overline{B} \wedge D)$	0x5A827999
20 to 39	$B \oplus C \oplus D$	0x6ED9EBA1
40 to 59	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8F1BBCDC
60 to 79	$B \oplus C \oplus D$	0xCA62C1D6

or Digest Message (DM). After the 80 rounds have been computed, the A to E 32-bit values are added to the current DM. The Initialization Vector (IV) or the DM for the first block is a predefined constant value. The output value is the final DM, after all the data blocks have been computed. In some higher level protocols such as the keyed-Hash Message Authentication Code (HMAC) [310], or when a message is fragmented, the IV may differ from the constant specified in [309].

To better illustrate the algorithm a pseudo code representation is depicted in Figure 4.2.

```
DM = DM_0 to DM_4 = IV
for for each data_block do
    W_t = expand(data_block)
    A = DM_0 ; B = DM_1 ; C = DM_2 ; D = DM_3 ; E = DM_4
    for t= 0, t≤79, t=t+1 do
        Temp = RotL^5(A) + f_t(B,C,D) + E + K_t + W_t
        E = D
        D = C
        C = RotL^30(B)
        B = A
        A = Temp
    end for
    DM_0 = A + DM_0 ; DM_1 = B + DM_1 ; DM_2 = C + DM_2
    DM_3 = D + DM_3 ; DM_4 = E + DM_4
end for
```

Figure 4.2
Pseudo code for SHA-1 function

4.2.2 SHA256 Hash Function

In the SHA256 hash function, a final DM of 256 bits is produced. Each 512-bit input block is expanded and fed to the 64 rounds of the SHA256 function in words of 32 bits each (denoted by W_t). Like in the SHA-1, the data scrambling is performed according to the computational structure depicted in Figure 4.3

by additions and logical operations, such as bitwise logical operations and bitwise rotations. The several input data blocks are mixed with the current state and the 32-bit round dependent constant (K_t).

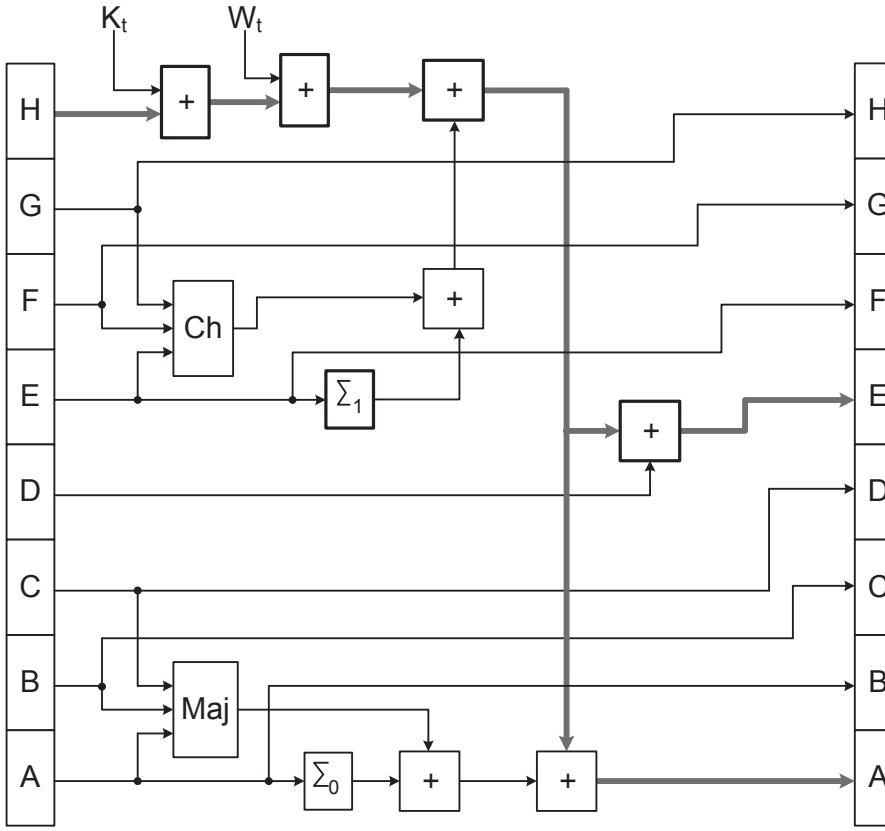

Figure 4.3
SHA-2 round calculation

The 32-bit values of the A to H variables are updated in each round and the new values are used in the following round. The IV for these variables is given by the 256-bit constant value specified in [309], being set only for the first data block. The consecutive data blocks use the partial DM computed for the previous data block. Each SHA256 data block is processed in 64 rounds, after which the values of the variables A to H are added to the previous DM in order to obtain the new value of the DM. Comparing the SHA round computation depicted in Figure 4.1 and Figure 4.3, it is noticeable a higher computational complexity of the SHA-2 algorithm in regard to the SHA-1 algorithm. To better illustrate this algorithm a pseudo code representation is depicted in Figure 4.4. The final Digest Message (DM) for a given data stream is given by the final DM value, obtained after the last data block is processed.

DM = DM_0 to DM_7 = IV
for each data_block i **do**

$W = expand$(data_block)
A = DM_0 ; B = DM_1 ; C = DM_2 ; D = DM_3
E = DM_4 ; F = DM_5 ; G = DM_6 ; H = DM_7

for t= 0, t≤ 63 {79}, t=t+1 **do**
$T_1 = H + \sum_1(E) + Ch(E,F,G) + K_t + W_t$
$T_2 = \sum_0(A) + Maj(A,B,C)$
$H = G$; $G = F$; $F = E$;
$E = D + T_1$
$D = C$; $C = B$; $B = A$
$A = T_1 + T_2$
end for

DM_0 = A + DM_0 ; DM_1 = B + DM_1
DM_2 = C + DM_2 ; DM_3 = D + DM_3
DM_4 = E + DM_4 ; DM_5 = C + DM_5
DM_6 = D + DM_6 ; DM_7 = E + DM_7
end for

Figure 4.4
Pseudo code for SHA-2 algorithm

4.2.3 SHA512 Hash Function

The SHA512 hash function computation is identical to that of the SHA256 hash function, differing in the size of the operands, 64 bits instead of 32 bits as for the SHA256. The DM has twice the width, 512 bits, and different logical functions are used [309]. The values W_t and K_t are 64 bits wide and each data block is composed of 16 × 64-bit words, having in total 1024 bits.

Table 4.2 details the logical operations performed in both the SHA256 and SHA512 algorithms, namely Ch, Maj, Σ_i, and σ_i, where $ROTR^n(x)$ represents the right rotation operation by n bits, and $SHR^n(x)$ the right shift operation by n bits.

4.2.4 Data Block Expansion for the SHA Function

The SHA-1 round computation, described in Figure 4.1, is performed 80 times, and in each round a 32-bit word, obtained from the current input data block (W_t), is used. Since each input data block only has 16 × 32-bit words (512 bits), the remaining 64 × 32-bit words are derived from the data expansion operation. This data expansion is performed by computing (4.1), where $M_t^{(i)}$ denotes the first 16 × 32-bit words of the i-th data block.

$$W_t = \begin{cases} M_t^{(i)} & , 0 \leq t \leq 15 \\ RotL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & , 16 \leq t \leq 79 \end{cases} \quad (4.1)$$

Table 4.2
SHA256 and SHA512 logical functions

Designation	Function
Maj(x,y,z)	$(x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
Ch(x,y,z)	$(x \wedge y) \oplus (\overline{x} \wedge z)$
$\sum_0^{\{256\}}(x)$	$ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
$\sum_1^{\{256\}}(x)$	$ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$
$\sigma_0^{\{256\}}(x)$	$ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
$\sigma_1^{\{256\}}(x)$	$ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$
$\sum_0^{\{512\}}(x)$	$ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$
$\sum_1^{\{512\}}(x)$	$ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$
$\sigma_0^{\{512\}}(x)$	$ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$
$\sigma_1^{\{512\}}(x)$	$ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$

For the SHA-2 algorithm, the round computation, depicted in Figure 4.3, are performed for 64 rounds (80 rounds for the SHA512). In each round, a 32-bit word (or a 64-bit word for the SHA512) from the current data input block is used. Once again, the input data block only has 16 words, resulting in the need to expand the initial data block to obtain the remaining W_t words. This expansion is performed by computing (4.2), where $M_t^{(i)}$ denotes the first 16 words of the i-th data block and the operator $+$ describes the arithmetic addition operation.

$$W_t = \begin{cases} M_t^{(i)} & , 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & , 16 \leq t \leq 63 \ \{\text{or } 79\} \end{cases} \quad (4.2)$$

4.2.5 Message Padding

In order to assure that the input data are a multiple of 512 bits, as required by the SHA-1 and SHA256 specification, the original message has to be padded. For the SHA512 algorithm the input data must be a multiple of 1024 bits.

The padding procedure for a 512-bit input data block is as follows: for an original message composed of n bits, the bit "1" is appended at the end of the message (the $n+1$ bit), followed by k zero bits, where k is the smallest solution to the equation $n + 1 + k \equiv 448 \mod 512$. The last 64 bits of the padded message are filled with the binary representation of n, i.e., the original message size. This operation is better illustrated in Figure 4.5 for a message with 536 bits (010 0001 1000 in binary representation).

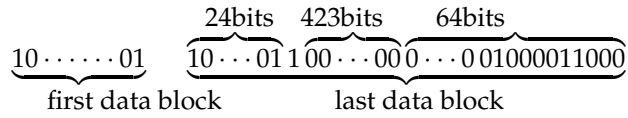

Figure 4.5
Message padding for 512-bit data blocks

For the SHA512 message padding, 1024-bit data blocks are considered with the last 128 bits, not 64 bits, reserved for the binary representation of the original message.

4.2.6 Hash Value Initialization

In the SHA-1 and SHA-2 standards, the initial value of the DM is a constant value, that is loaded at the beginning of the computation. The DM value can be loaded into the SHA state registers by using set/reset signals. However, if the SHA-2 algorithm is to be used in a wider set of applications and in the computation of fragmented messages, the initial DM is no longer a constant value. In these cases, the initial value is given by an Initialization Vector (IV) that has to be loaded into the A to E variables, in the SHA-1 case, and in the A to H variables, in the SHA-2 case.

4.3 SHA-1 and SHA-2 Implementations

This section presents the several techniques that have been proposed in the related state of the art allowing to achieve SHA-1/2 implementations with significant improvements in terms of area and throughput. Given the similarities between data paths of the SHA-1 and SHA-2 algorithms, the following details the proposed techniques with a particular focus on the SHA-1 algorithm. This allows us to simplify the description of the proposed improvements, given the simpler data path of the SHA-1 algorithm. Nevertheless, given their similitude, the presented techniques can also be exploited for the SHA-2 algorithm, as detailed in the presented state of the art. Regarding the SHA256 and SHA512 implementations, the main difference lays in the length of the data path. Sklavos [401] explores this to design a computational structure capable of computing both the SHA256 and the SHA512 hash functions with a negligible area and delay increase.

In its canonical form, SHA implementations are bound by the data dependency between rounds. The performance of the SHA implementation can be improved by registering the output of the data expansion block so it does not impact in the performance of the SHA compression structure. Carry Save

Adders (CSA) can also be used to perform the additions of intermediate values, only using one full adder, saving area resources, and reducing the computation delay [144, 400].

Independently of these optimizations to further improve the design of the SHA algorithm, data dependencies must be taken into account.

Dadda [93] improves the SHA computation by balancing the delay in the computation of the data expansion. This improvement starts by identifying the critical path in the structure performing the expansion of the input data blocks into 32-bit words, in particular for the SHA256 algorithm. The authors reduce the critical path by dividing the computation of the output block (W_t) into stages separated by a register. Allowing to double the operating frequency and consequently the throughput of the data expansion computation. This approach requires additional area resources for the pipeline registers and control logic and imposes an additional clock cycle of latency.

The computational structures of the SHA compressions are relatively simple. However, in order to compute the values of one round the values from the previous round are required. This data dependency imposes a sequentiality in the processing, preventing the parallel computation between rounds. Dadda [93] proposes the use of a quasi-pipeline technique in this compression stage, allowing for a fast pipelined SHA architecture, using registers to break the long critical path within the SHA core. However, given the high data dependency of the SHA algorithm, additional control logic is required in order to properly control the partial pipeline registers. Nevertheless, the proposed quasi-pipelined design achieves a shorter critical path, allowing to achieve higher data throughputs.

Chaves [70, 71] further improves the pipeline usage in the SHA implementation by analyzing the existing data dependencies, proposing the functional rescheduling of the SHA arithmetic operations. As depicted in Figures 4.2 and 4.4, the bulk of the SHA-1 and SHA-2 round computation is oriented for the computation of the A value (and E for the SHA-2 algorithm). The remaining values do not require any particular computation.

For the particular case of the SHA-1 algorithm the value of A is calculated with the addition of the previous value of A along with the other internal values. Nevertheless, since only the parcel $RotL^5(A_t)$ depends on the variable A_t, the value A_{t+1} can be pre-computed using the remaining values that do not require computation on that cycle, producing the intermediate carry and save values, as depicted in Figure 4.6. By splitting computation of the value A and rescheduling it to different computational cycles, the computation can be optimized using an additional pipeline stage. With this, the critical path is further reduced with a minimum area overhead. In this particular case the critical path is restricted to a bit-wise rotation, with as a negligible impact, a CSA, and a final addition, as depicted in gray in Figure 4.6. An identical rescheduling is performed for the SHA-2 algorithm, considering its more complex data path [70].

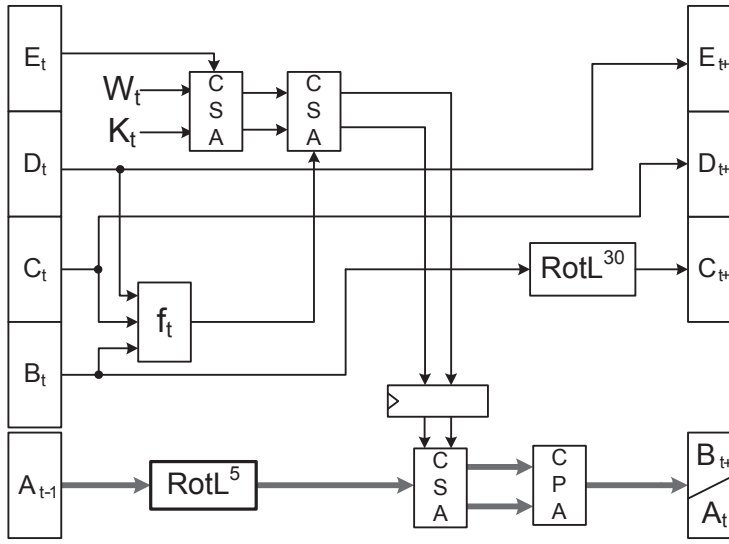

Figure 4.6
SHA-1 round rescheduling

Chaves [72] also considers hardware reuse in the addition of the final round value with the current Digest Message (DM) value. Rather than using one adder for the addition of each internal variable, A to H in the case of the SHA-2, the data dependencies are once more explored to minimize the needed hardware. The proposed DM addition also allows the loading of different IV at the cost of a few selection logic units. This hardware reuse allows to further reduce the footprint of the SHA implementations.

Going in the opposite direction, Lien [247] proposes the use of unrolling techniques to improve the maximum achievable throughput. Even though the data dependencies exist, only part of the computation is affected by this computation. As depicted in Figure 4.7 for the SHA-1 algorithm, by unrolling the computation twice, it is possible to compute in parallel two SHA rounds with a minimal increase in delay, regarding the canonical implementation, as illustrated by the gray lines in Figures 4.1 and 4.7.

The authors in [247], propose to unroll the loop up to five times, resulting in the parallel computation of 5 rounds with a similar computational delay. As expected, this loop unrolling results in an increase in area requirements, proportional to the considered loop unrolling. To further improve the delay the computation of the first logical function (f_t), depicted with a shaded box in Figure 4.7, can be rescheduled to the previous clock cycle. The authors propose the same approach to the SHA-2 algorithm. Regardless of its more complex data path, identical improvements are achieved for SHA-2.

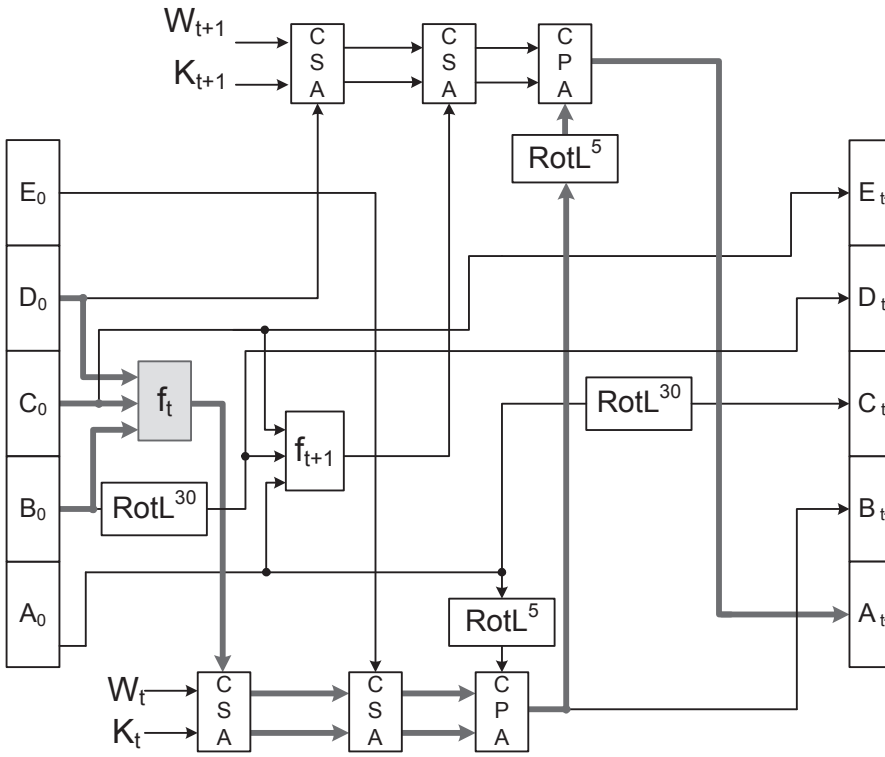

Figure 4.7
SHA-1 2× loop unrolling

Given the concurrent computation of more than one round, the hardware performing the block expansion also need to be modified in order to output several round input values (W_t) for the compression stage [281]. McEvoy [281] further improves the SHA-2 throughput by combining the use of unrolling techniques with the quasi-pipeline solution proposed by Dadda [93].

All of the above proposed improvements to the SHA-1 and SHA-2 computation play with the data dependencies in order to improve the resulting structures, in area and/or throughput [242, 390]. Nevertheless, these optimizations are limited by the existing data dependencies. Lee [242] proposes the Iterative Bound Analysis method to identify the architectural performance limit imposed by the data dependencies of these algorithms. With this approach, the authors once more used pipeline and unrolling techniques to further improve the performance metrics of the obtained implementations.

Table 4.3 presents the resulting performance metrics considering the several implementation technologies considered in the related SHA-1 and SHA-2 state of the art.

Table 4.3
SHA-1 and SHA-2 implementations

Algorithm	Technology [ASIC/FPGA]	Area [Gates/Slices]	Throughput [Mbps]	Efficiency [Throughput/Area]
SHA-1 Gremb. [144]	Virtex	730	462	0.63
SHA-1 Lien [247]-basic	Virtex-E	484	659	1.36
SHA-1 Lien [247]-unrolled	Virtex-E	1484	1160	0.78
SHA-1 Chaves [71]	Virtex-E	388	840	2.16
SHA-1 Chaves [71]	Virtex-2P	565	1420	2.51
SHA256 Sklavos [400]	Virtex	1060	326	0.31
SHA256 McEvoy [281]	Virtex-2	1373	1009	0.74
SHA256 Chaves [70]	Virtex-2	797	1184	1.49
SHA256 Chaves [70,133]	Virtex-5	433	1630	3.76
SHA256 Dadda [93]	0.13μm	n.a.	7420	n.a
SHA256 [242]	0.13μm	22025	5975	0.27
SHA256 [369]	0.13μm	15329	2370	0.15
SHA512 Sklavos [400]	Virtex	2237	480	0.21
SHA256/512 Sklavos [401]	Virtex	2384	291/467	0.12/0.20
SHA512 Gremb. [144]	Virtex	1400	616	0.44
SHA512 Lien [247]-basic	Virtex-E	2384	717	0.30
SHA512 Lien [247]-unrolled	Virtex-E	3521	929	0.26
SHA512 Chaves [70]	Virtex-E	1680	889	0.53
SHA512 McEvoy [281]	Virtex-2	2726	1329	0.49
SHA512 Chaves [70]	Virtex-2	1666	1534	0.92
SHA512 [242]	0.13μm	43330	9096	0.21
SHA512 [369]	0.13μm	27297	2909	0.11

From the presented results it is clear that significant improvements to the SHA-1 and SHA-2 implementations can be achieved if the data dependencies are properly explored and hardware re-usage is considered. However, it is also clear that a tradeoff has to be made between less area demanding and higher throughput structures. The results suggest that adequate compromises can be achieved in order to achieve a higher throughput per used area resources.

4.4 SHA-3

One of the widely accepted hash functions of past years is SHA-1, which was introduced by the National Institute of Standards and Technology (NIST) in 1995 [311]. SHA-1 processes 512-bit message blocks, operating at 80 rounds, and outputs a 160-bit message digest as described in earlier sections of this chapter. For many years, SHA-1 appeared to be collision resistant, until several researchers have proven that it is possible, under certain conditions, in a computationally efficient way to find message collisions, thus reducing the SHA-1 security level to that of an 80-bit block cipher. Due to this issue, in 2001 NIST announced the follower of SHA-1, denoted as SHA-2. The SHA-2 hash function standard employs four hash functions with different and longer message digests at 224, 256, 384, and 512 bits. Unfortunately, this standard did not meet NIST's high expectations, so in October 2008 the institute announced an open competition for a new cryptographic hash function called SHA-3, aimed at replacing SHA-1 and SHA-2 [308].

The competition for SHA-3 had at first 64 algorithm submissions. After a thorough study, 51 out of the 64 candidates passed to round 1. NIST selected 14 out of 51 algorithms for round 2 and concluded at five algorithms for the final competition round (round 3). Those finalist algorithms were BLAKE, Grøstl, JH, Keccak, and Skein and met all the required SHA-3 criteria specified by NIST. The most noticeable such criteria were high security, diversity, analysis, and performance. Moreover, NIST had set the requirement that the function to be chosen as the SHA-3 standard will have to be an open-source hash algorithm, be available worldwide for public use, and be suitable for a wide range of software and hardware platforms. Furthermore, the SHA-3 winner algorithm had to support a message hash of 224, 256, 384, and 512 bits and a maximum length message of at least $(2^{64} - 1)$ bits. Finally, NIST specified additional characteristics to be evaluated for the new SHA-3 choice, like simplicity, flexibility, computational efficiency, memory use, and licensing requirements. On October 2012 the competition ended and NIST announced the SHA-3 winner. Out of the five excellent algorithms that reached round 3, Keccak was chosen as the new SHA-3 standard.

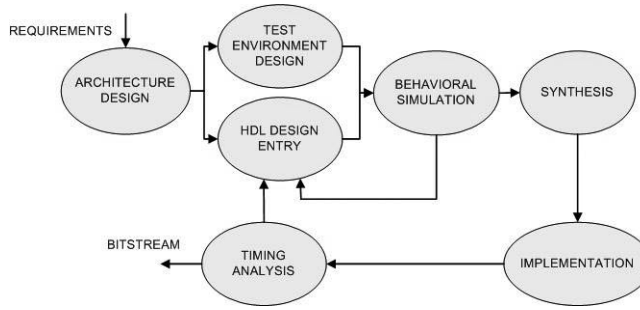

Figure 4.8
FPGA design flow

In this section, we describe and evaluate the 5 SHA-3 finalists and analyze their efficiency from hardware perspective in order to justify NIST choice for Keccak. To achieve that, we implement the 5 finalists in FPGA technology and extract performance measurements. FPGAs were chosen as the best mean implementing the various operations and specification of the five finalist cryptographic hash functions since they are reprogrammable and allow rapid prototyping. Similar approach for SHA-3 candidates evaluation was followed by others research groups where compact implementations were used for comparisons [209] or 0.13 μm standard-cell CMOS technology [150] or embedded FPGA Resources (like DSPs and BRAMs) [388]. However, this chapter's approach is entirely different since all the functions have been implemented in FPGA under the same design philosophy thus ensuring the accuracy of the compared results and providing fairness in comparisons.

4.4.1 FPGA Background

Field Programmable Gate Arrays (FPGA) are integrated circuits that can be configured by a VLSI designed after fabrication. This configuration is defined using a hardware description language (HDL), like VHDL and Verilog, or a schematic design, and it can be used in order to implement any logical function. The HDLs are mostly used in large structures, but schematic entries are preferred for easier visual image design. The design flow employed by the most FPGA designers can be seen in Figure 4.8.

During architecture design, the requirements and problems of the target system functionality are analyzed and a document is provided as output presenting the interfaces and the operations of the system's structural blocks. During environment design, the system's test environment and suitable behavioral models are specified and realized. The outcome of environmental design is used in the behavioral simulation stage where the outputs of the HDL model are compared with the provided behavioral model. During synthesis stage, the behaviorally verified HDL code is converted to a digital cir-

cuit schematic, denoted as netlist. The netlist is used in the implementation stage to designate the FPGA components needed for the realized system. The output of this stage is a file called bitstream generator that is used for timing analysis operations in order to verify if a design follows the timing rules, set by the developer [280].

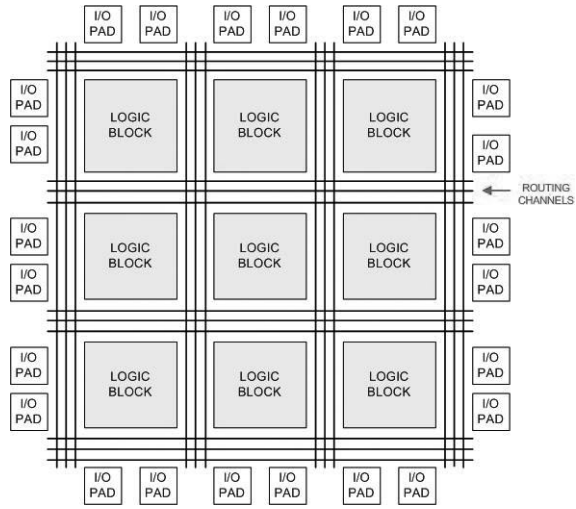

Figure 4.9
FPGA structure

The architecture of an FPGA consists of an array of logic blocks along with appropriate routing channels and I/O pads, as described in Figure 4.9 [228]. Each logic block contains lookup tables (LUT), flip-flops and special purpose hardware structures like full adders, etc. A typical logic block is shown in Figure 4.10. Up-to-date FPGA devices may also include DSP elements and RAM blocks (depending on the FPGA manufacturer and family). Due to the logic block array, the FPGA can compute any complex combinational function or simple logic gate configuration.

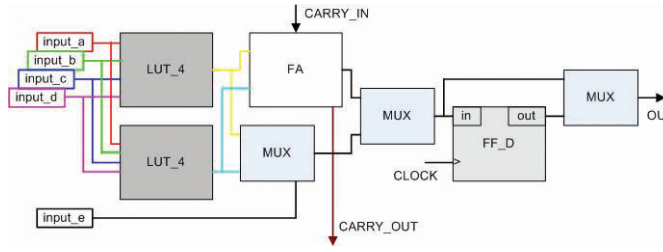

Figure 4.10
The FPGA logical cell

FPGAs apart from digital operations can also support analog functions. The most common such function is an FPGA characteristic capable of making pin signals propagate faster and stronger on high speed channels. Furthermore, the connection to differential signaling channels and the mixed signal FPGAs that let blocks operate as a system-on-chip, make FPGAs advantageous for many applications.

Originally, FPGAs were applied in telecommunications and networking letting microprocessors handle all other computation work. However, the computation application demands for low area, small delay, high speed, and parallelism, over the years could not be matched by microprocessors. FPGAs, having the ability to fix into the silicon higher level functionality, reduce the required area, and provide high speed to every function, are ideal candidates to address the above issue. So, nowadays, digital signal processing, computer vision and medical imaging, aerospace, cryptography and defense systems, computer hardware, and ASIC prototyping are gradually been migrated on FPGA technology replacing other implementation platforms like ASICs, general or specialized microprocessors.

4.4.2 SHA-3 Finalist Hash Function Algorithms

4.4.2.1 Blake

BLAKE hash function follows the iteration mode of HAIFA (Framework for Iterative Hash Functions). BLAKE internal structure is based on the LAKE hash function (which is a version of stream cipher ChaCha) and its associated compression algorithm. The main benefits of the BLAKE is its strong security, its high performance, and its parallelism. BLAKE hash family consists of four different function members: BLAKE-28, BLAKE-32, BLAKE-48, BLAKE-64. The characteristics of each family member are basically different in the word length they use for the data transformation, in the applied message block, the produced message digest, as well as in the used salt. The BLAKE members specifications are described in Table 4.4.

Regardless of the differences that each BLAKE family member may have, the architecture of each member is practically the same. The main hardware components and functions that are used in BLAKE architectures are described in Figure 4.11. The BLAKE algorithm consists of two parts: the com-

Table 4.4
BLAKE Hash family

Hash Algorithm	Word	Message	Block	Digest	Salt
BLAKE-28	32bits	$< 2^{64}$	512	224	128
BLAKE-32	32bits	$< 2^{64}$	512	256	128
BLAKE-48	64bits	$< 2^{128}$	1024	384	256
BLAKE-64	64bits	$< 2^{128}$	1024	512	256

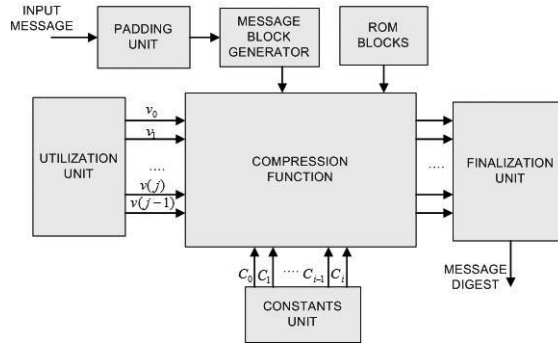

Figure 4.11
Generic BLAKE architecture

pression function and the iteration mode. The compression function needs four inputs: the chain value h, the block message m, the salt value s, and the counter value t. The BLAKE hashing process follows three stages: the initialization process, the round function, and the finalization process. The first process uses h,s,t to create a 4×4 matrix that initializes a 16-word value v to come up with v_i $(i = 0, \ldots, 15)$ different initial states.

These states are inputted to the round function, where the compression function G_i is computed in r partially parallel rounds for $i = 0, ...7$. The output of this stage is a new state value v that is used for generating the chain value $h' = h'_i$ for $i = 0, ...7$ during finalization. In the finalization stage, values h, s, v are XORed in order to produce a new chain value $(h'_i = h_i \oplus s_{i \mod 4} \oplus v_i \oplus v_{i+8})$. BLAKE-32 needs $r = 14$ rounds of its compression function to come up with a result while BLAKE-64 needs $r = 16$ rounds [23], [398].

The message in BLAKE is extended through padding so that its length is congruent to 447 modulo 512 (BLAKE-28/BLAKE-32) or 895 modulo 1024 (BLAKE-48/BLAKE-64). The padded bits that will be added are from 1 to 512. The first bit in this sequence is 1, followed by zeros. Finally, for BLAKE-32 and 64, at the padding's end a bit 1 is added followed by a 64-bit unsigned big-endian representation of the message's bit length l. Though the BLAKE iteration hashing the padded message is divided into 16-word blocks, if the last block contains none of the bits present in the first block, the counter is set to zero and the final message digest is outputted.

4.4.2.2 Grøstl

The quality of analysis, the strong security features, and the well defined design principles are some of the most significant advantages of the Grøstl hash function algorithm. The hash function of this algorithm is based on round iterations of a compression function f with inputs original message blocks and

the previous round's f output (Figure 4.12). The initial message is divided in m_i blocks, each of l bits [136]. Value l for Grøstl-224 and -256 is equal to 512, using 10 rounds while for Grøstl-384 and -512, is 1024 with 14 rounds.

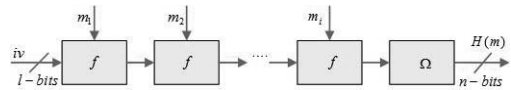

Figure 4.12
The Grøstl cryptographic hash function

The compression function f, as depicted in Figure 4.13, is computed following the equation $f(h,m) = P(h \oplus m) \oplus Q(m) \oplus h$ where Q and P are two permutation operations.

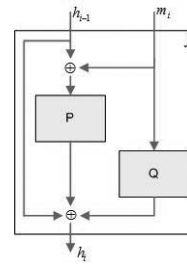

Figure 4.13
The Grøstl compression function

Permutations P and Q are based on the Rijndael block cipher algorithm (AES) and their design consists of four round transformations: AddRoundConstant, SubBytes, ShiftBytes, and MixBytes [136], [28]. AddRoundConstant XORs a round dependent constant $C[i]$ (where i is the Grøstl round) with every byte of each l bits input state A. The output of this function is $A \longleftarrow A \oplus C[i]$. SubBytes transformation replaces all bytes in the state matrix with other values taken from the Grøstl S-boxes (similarly to the AES algorithm's Sboxes). ShiftBytes makes a cyclic shift of all bytes in a row of the matrix A according to a vector $\sigma = [\sigma_0, \sigma_1, \sigma_2, ...\sigma_7]$ where σ_i represents the number of bits to be shifted in row i. Shiftbytes in Q permutation uses the vector: $\sigma = [1,3,5,7,0,2,4,6]$ and in P the vector $\sigma = [0,1,2,3,4,5,6,7]$. In functions where larger permutations are required, the ShiftBytesWide transformation is used in Q_{1024} with vector $\sigma = [1,3,5,11,0,2,4,6]$ and P_{1024} with $\sigma = [0,1,2,3,4,5,6,11]$. Finally, MixBytes transforms the bytes of each column in the state matrix A to elements of the finite field F_{256}. This means that each column of a matrix A is multiplied with an 8×8 matrix B in F_{256}. Each row of matrix B is left-rotated related to each previous row of B. The output message is equal to $A \longleftarrow B \times A$.

4.4.2.3 JH

The JH hash function family has four members, JH-224, JH-256, JH-384, JH-512. The main characteristics of JH are high parallelism in computing, use of the same hardware components for all four JH family members, and easy implementation. JH uses a generalized AES design methodology and in combination with the JH compression function provides strong security.

JH operates using a hash block (H) of 1024 bits and message blocks (M_i) of 512 bits as presented in Figure 4.14 [456]. The final output bits are compressed using the following function $H_i = F_d(H_{i-1}, M_i)$.

The JH compression function F_d consists of several steps as can be seen in Figure 4.14. Initially, the left half of the hash block H_{i-1} is XORed with the message block M_i. The generated message goes into a bijective function E_d, which is a block cipher with constant key. This function consists of a set of operations that are iterated for 35 rounds. The two 4-bit Sboxes contained in E_d are updated in every round based on a round constant vector, consisting of 256 bits. This constant vector is computed in mostly parallel and its update is based on the SBox chosen to be used in each round. The final operation of E_d is a linear transformation and permutation, where, at least, one 4-bit Sbox is used. Finally, the left part of the value generated from the bijective function E_d, becomes the hash value of the initial message's leftmost part.

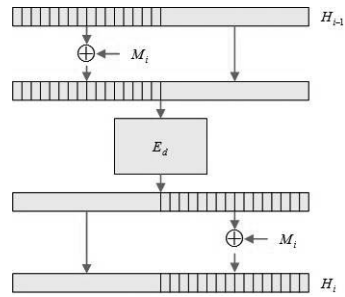

Figure 4.14
The JH compression function

Similarly, the right half of H_{i-1} is also inputted to the E_d function without XORing with M_i. This part passes through the same compression procedure like the left part and then is XORed with M_i to become the hash value of the initial message's rightmost part.

4.4.2.4 Keccak

Keccak is the algorithm chosen by NIST as the new SHA-3 standard. This algorithm is based on the sponge construction described in Figure 4.15, taking advantage of its double resistance to attacks and collisions [39]. Moreover, this construction allows multiple lengths for input and output messages and

Table 4.5
Keccak bitrate r values

Hash bits	Bitrate r
224	1152 bits
256	1088 bits
384	832 bits
512	576 bits

leads to the fastest implementations compared to all the other four candidates. Keccak is a family of hash functions, each member of which is characterized by two values, the bitrate r and the capacity c (Keccak[r,c]). The bitrate is dependent on the output hash size, according to Table 4.5, and the factor c, the capacity of the hash function, is equal to $c = 1600 - r$ state bits [40]. The input and output size of the messages at every round of Keccak is a 5×5 matrix, with entries of 64-bit words, (a total of 1600 bits). A complete permutation of the Keccak f function needs 24 rounds, each of those rounds consisting of five steps $(\theta, \rho, \pi, \chi, \iota)$.

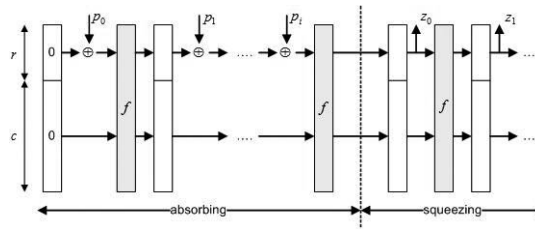

Figure 4.15
The Keccak sponge construction

The basic functionality of Keccak can be summarized in the following steps:

- initialization of the message state to 0 at every round,
- padding of the input message and dividing it into r bits,
- XORing of these bits with the initial bits of the state and performing an f permutation operation called absorbing,
- apply the block permutation squeezing, at the same rate as in the previous steps, and output the required bits from the hashed outputs z_i .

4.4.2.5 Skein

Skein is characterized by speed, simplicity, security, parallelism, and flexibility. The hash algorithm is based on three basic components: the Threefish

block cipher, the Unique Block Iteration (UBI) chaining, and an argument system, which contains a configuration block and optional arguments [124], [426]. The general idea of this algorithm is similar to Keccak sponge function, providing Skein with the ability to generate arbitrary hash values from fixed size initial messages of 256, 512, or 1024 bits. The difference from other approaches is relevant to the state of capacity and security, because Skein tweaks the bits in a way that is unique for its block. This operation is possible through the use of Threefish tweakable block cipher.

Threefish cipher uses XOR, addition, and rotation in order to come up with a ciphertext result. It uses an N-bit encryption key, which is a power of 2 equal or greater to 256 and a 128-bit tweak in order to encrypt an N-bit plaintext block. Furthermore, Treefish needs 72 rounds for Threefish-256, -512 and 80 rounds for Threefish-1024 to come up with a result. Each round is characterized by a specific number of nonlinear mixing functions, denoted as MIX [124], which are 2 for Threefish-256, 4 for -512, and 8 for -1024. MIX uses as input two 64-bit words. After every 4 rounds, an N-bit subkey is added. The Skein one round operations are presented in Figure 4.16.

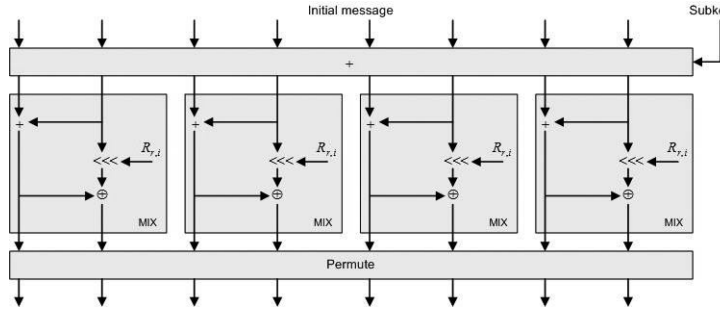

Figure 4.16
One round of Skein

The UBI chaining mode uses the Threefish algorithm in order to build the Skein compression function in order to map an arbitrary input size message to a fixed size output. Skein uses the following arguments:

- a key, to choose if Skein will use the MAC or the KDF function,
- a configuration block,
- the personalization string, which is used to create different functions for different uses,
- a hashed public key,
- the key derivation identifier,
- the nonce value, used in stream cipher mode,

- the message that is going to be hashed, and
- the generated output.

Skein was implemented in a version suitable for HW acceleration in a system-on-chip platform in [445]. In this publication, the authors present an area-efficient, energy-efficient Skein-512 implementation which offers 58Gbps throughput at a total latency of only 20 clock cycles. The core datapath is unrolled to accommodate 8 rounds of Threefish which are pipelined. This allows parallel computation of two independent hashes, but requires two tweak generators and two key schedulers to independently supply two subkeys to keep the hardware pipeline filled with two independent messages during each cycle. This increases latency from 10 cycles to 20 cycles. The total area for the implementation is 60.4Kgate equivalents.

4.5 Reconfigurable Hardware Architecture Approaches

One of the most important criteria that NIST set in the SHA-3 competition was performance efficiency. As such, all the candidate algorithms were implemented in various platforms in order to provide accurate, detailed, and comprehensive performance measurements for the evaluation process. Hardware implementation is a vital part of this process since it provides an insight of how the candidate algorithms behave in restricted hardware resources systems [218], [399]. Especially during round 3 of the competition, where all algorithms offered very strong security, the performance criteria played a very important role in determining the SHA-3 winner.

Hardware implementation for testing the SHA-3 finalists is mainly focused on FPGA technology. FPGA design offers flexibility, high performance, reconfigurability, and is a widely used implementation tool for testing but also as a commercial application solution. The design approach to be used for implementing the five SHA-3 finalists can be derived from the algorithms' basic functionality that is based on iterations (rounds). As such, iterative design, parallelism, and pipelining techniques can be very successful at improving the hash function implementation efficiency in terms of speed and/or required hardware resources. In this section, we describe the hardware approaches used in this chapter's implementations of the five SHA-3 finalist hardware architectures [350].

4.5.1 Iterative Design

Iterative design approach is very fitting for implementations where iterative operations are needed. Thus, it is ideal for hash functions and is presented in Figure 4.17. In this design approach, the initial message is configured in two

separated steps. Firstly, the message is padded following the specifications of each hash algorithm in order to be expanded to the algorithm's appropriate bit length. Secondly, the message scheduler produces a message sub-block provided to every round or step of the algorithm. The data produced in each round of the algorithm, can be saved in a memory block, such as ROM. An iteration block implements the hash function functionality for one round (including the compression function), and this block is reused as many times as the number of rounds with appropriate inputs provided from the rest of the architecture's components. The various operations of each hash function round that are completed in several steps are synchronized using the hash finite state machine.

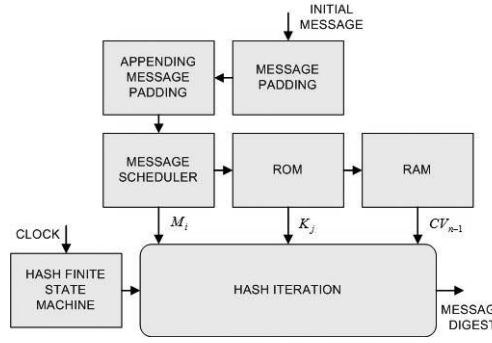

Figure 4.17
Hash function iterative design approach

Finally, the output block of hash iteration is XORed with the initial hash values, which are stored for this function until the end of any algorithm iteration. Block RAMs can be used to store parts of the output at the end of any iteration, in order to become inputs for the next hash function round. It is obvious that depending on the characteristics of the implemented hash algorithm, the iterative design is adapted, varied, or qualified accordingly.

4.5.2 Pipeline Design

The pipelining design approach can be used to process a large amount of data in a faster way, because through pipeline registers the critical path of the datapath can be considerably reduced. The main disadvantage of this technique is the high chip covered area due to the additional registers.

Using the above technique, inner-round hash function pipeline design can be achieved by dividing round operations into pipeline stages separated by registers as depicted in Figure 4.18.

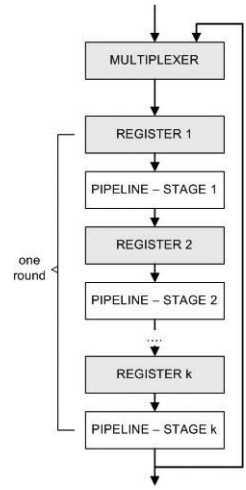

Figure 4.18
Inner-round pipelining

4.5.3 Results

Following the previous section's design approaches we have implemented all five SHA-3 finalist hash function algorithms using the VHDL hardware description language and the XILINX FPGA Virtex 6 device in an effort to identify the performance characteristics that led NIST into choosing Keccak as the new SHA-3. All implementations follow similar design principles in order to provide comparison accuracy and fairness.

We use throughput, area, and the throughput-per-area metrics in order to determine the performance of the implemented hash function algorithms. For the area metric the number of FPGA slices was used rather than dedicated FPGA resources such as block RAMs or DSPs [334] for the sake of fairness in comparisons. We excluded the use of block RAMs since they can be too big for the hash function's actual needs. The obtained results are for 256-bit message digest and are presented in Table 4.6 for the iterative design approach and in Table 4.7 for the pipeline design approach.

Table 4.6
Iterative design implementation results

Implementation	Throughput (Mbps)	Area (Slices)	Throughout/Area
BLAKE	2460	1790	1.37
Grøstl	10310	2800	3.68
JH	6010	922	6.51
Skein	1506	1200	1.25
Keccak	**11991**	**1210**	**9.90**

Table 4.7
Pipeline design implementation results

Implementation	Throughput (Mbps)	Area (Slices)	Throughout/Area
BLAKE	5319	2048	2.60
Grøstl	19764	3248	6.08
JH	7410	1503	4.93
Skein	2258	1406	1.60
Keccak	**17429**	**1507**	**11.56**

As can be observed from Tables 4.6 and 4.7 the SHA-3 finalist hash function implementations achieved remarkable time performance and very good area results. However, the performance efficiency, more accurately measured by the *throughput/area* ratio, of the SHA-3 winner (Keccak) is considerably better than the ration of the other finalist implementations.

To further verify this remark, comparisons where made of the proposed implementations with previous works in terms of the synthesis and time performance, provided in Table 4.8. Note that iterative architectures are symbolized as $NAME_{1R}$ while the pipeline architectures are symbolized as $NAME_P$. In [209] compact implementations using 64-bit bus internal datapath were proposed for the five finalists using a common methodology, which allows fair performance comparisons. The implementations of [209] do not employ any dedicated FPGA resources such as block RAMS or DSPs. Both of the proposed design techniques (iterative and inner-pipeline designs) are much better in terms of time performance compared to the implementations in [209] although they are worse in terms of area resources. However, the *throughput/area* performance ratio of the proposed implementations is far better than the designs in [209].

For the five implementations in [150] a 0.13 μm IBM process using standard-cell CMOS technology was used. One round of the hash function algorithms was selected as design methodology. We provide the results of [150] in Table 4.8 for the sake of completeness since comparisons between ASIC and FPGA designs are not compatible (lack of fairness).

Finally, in [388] implementations, dedicated FPGA resources such as block RAMs and DSPs were employed for a one round design approach similar to [150]. The proposed implementations outperform the identical one presented in [388].

Table 4.8 results of previous works indicate that the best time performance is achieved by Grøstl hash function while the second better by Keccak hash function. In terms of area resources BLAKE hash function implementation is the most compact implementation while Keccak is the biggest one. However, the implementations proposed in this book chapter based on the iterative and pipelining approach, providing in general better results than the other similar works, highlight the considerable *throughput/area* perfor-

Table 4.8
SHA-3 finalists comparison results

Implementations	Throughput (Mbps)	Area (Slices)	Throughput/Area
BLAKE [209]	132	117	0.75
BLAKE [150]	2.13 Gbps	34.15 kGEs	62.47
BLAKE [388]	1534	662	2.31
$BLAKE_{1R}$ Proposed	2460	1790	1.37
$BLAKE_P$ Proposed	5319	2048	2.60
Grøstl [209]	960	260	3.27
Grøstl [150]	9.31 Gbps	124.34 kGEs	85.58
Grøstl [388]	8057	1627	4.95
$Grøstl_{1R}$ Proposed	10310	2800	3.68
$Grøstl_P$ Proposed	19764	3248	6.08
JH [209]	222	240	0.73
JH [150]	3.05 Gbps	49.29 kGEs	34.08
JH [388]	3120	1066	4.95
JH_{1R} Proposed	6010	922	6.51
JH_P Proposed	7410	1503	4.93
Keccak [209]	145	144	0.77
Keccak [150]	10.67 Gbps	42.49 kGEs	14.7
Keccak [388]	11252	1338	8.41
$Keccak_{1R}$ Proposed	11991	1210	9.90
$Keccak_P$ Proposed	17429	1507	11.56
Skein [209]	223	240	0.77
Skein [150]	3.05 Gbps	66.36 kGEs	39.71
Skein [388]	2359	1264	1.86
$Skein_{1R}$ Proposed	1506	1200	1.25
$Skein_P$ Proposed	2258	1406	1.60

mance ratio of Keccak in comparison with the other implementations. This remark provides a very good justification of NIST's choice for Keccak as the new SHA-3 standard since from overall performance perspective this algorithm has the best potential.

4.6 Conclusions

NIST SHA-3 hash competition ended in October 2012 selecting Keccak among 5 candidates as the successor of SHA-2. In this chapter we described SHA-1, SHA-2, and SHA-3 finalists, suggested design approaches applicable to all those finalists and provided implementations based on these approaches in order to evaluate the finalists hash functions in an effort to justify NIST's choice. All finalist algorithms are highly secure and have their

advantages. BLAKE is very strong in security, performance and parallelism. Grøstl has significant quality of analysis, security features and well defined design values. JH favors parallel computing, easy implementation. Skein can provide outcomes fast, in a simple manner and favors parallelism and flexibility. However, Keccak having the privilege of the sponge construction, that takes advantage of its double resistance to attacks and collisions, is the most strong among the other finalist due to its elegant design, flexibility and excellent efficiency in hardware implementations. This efficiency was highlighted in our work by the provided FPGA hardware implementation results that clearly indicated Keccak's advantage in throughput per area performance metrics, suggesting a well balanced, from implementation perspective, algorithm worthy of being SHA-3. In 2014, NIST published the draft FIPS 202 "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions" and the standardization process is in progress as of April 2015.