International Journal of Engineering Trends and Technology
ISSN: 2231 – 5381 /doi:10.14445/22315381/IJETT-V69I6P210

Volume 69 Issue 6, 71-78, June 2021
© 2021 Seventh Sense Research Group®

The SHA3-512 Cryptographic Hash Algorithm Analysis And Implementation On The Leon3 Processor

Mouna Karmani, Noura Benhadjyoussef, Belgacem Hamdi, Mohsen Machhout

Electronics and Micro-Electronics Laboratory (E.µ.E.L),
Faculty of Sciences of Monastir, University of Monastir,
Tunisia

karmani.mouna@fsm.rnu.tn benhadjyoussef.noura@fsm.rnu.tn

***Abstract* — *Embedded systems are computer-based systems designed to execute specific functions. Such a system is, in general, embedded as part or unit of a complete device in order to control, monitor, and facilitate its operation. An embedded system includes hardware and software parts with fixed or programmable capabilities. Thus, with the ever-increasing role that software is playing in embedded systems, software performance is one of the embedded system implementation goals. In this paper, we consider the software cryptographic hash-functions implementation on hardware platforms. In fact, Hash functions are used in several information-security applications like message authentication codes, digital signatures, and other forms of authentication. In this work, we presented a detailed case study, including the SHA3-512 algorithm analysis and implementation on the LEON3 soft core processor. The SHA3-512 is programmed using the C language, and compiler optimization techniques are used to improve the efficiency of the obtained executable programs in order to be implemented on LEON3 using the ML507 Virtex-5 Xilinx FPGA board.*

Keywords — *SHA3-512 algorithm; simulation; debugging; LEON3 processor; FPGA implementation.*

I. INTRODUCTION

Cryptographic hashing is different from cryptographic encryption because encryption prevents passive attacks, while a cryptographic hash algorithm is used to create a unique digital fingerprint of data that can be used to prevent active attacks or falsification. Thus, Secure Hashing Algorithms (SHAs) are used to ensure information integrity for embedded system security [1]. The hash functions family are developed by the US National Security Agency (NSA) and published as standards by the US National Institute of Science and Technology (NIST). It was the appropriate algorithm to secure applications used by the US government agencies. An important feature of the SHA algorithms is that a minor change in the input leads to a major change in the output value [2-3]. SHAs are published by the NIST as a U.S. federal information processing standard, including SHA-0, SHA-1, SHA-2, and SHA-3. In fact, SHA-0 was the original version of the 160-bit hash function published in 1993 under the name "SHA". It was withdrawn and replaced by the SHA-1 revised version. The 160-bit SHA-1 hash function resembles the earlier MD5 algorithm; it was designed by NSA to be part of Digital Signature Algorithms (DSA). Since cryptographic weaknesses were disclosed in SHA-1, this standard was no longer used for most cryptographic uses after 2010. Therefore, SHA-1 was replaced by SHA-2, the family of two similar hash functions known as SHA-256 and SHA-512 with different block sizes. There are also truncated versions of each standard, known as SHA-224, SHA-384, SHA-512/224, and SHA-512/256 [4-5-6-7-8-9-10]. The SHA-2 hash functions are the most used hash algorithms for previous years. Nevertheless, an international competition was announced by the NIST in order to elaborate a new SHA-3 hash function. The competition was finalized in August 2015, and the KECCAK is the new SHA-3 final version [11]. The table below illustrated the SHA3 variants and specifications [12].

TABLE I. THE SHA3 VARIANTS AND SPECIFICATION

SHA3 Variant	Output size (bits)	Block size (the bitrate r) (bits)	Capacity c (bits)	Security
SHA3-224	224	1152	448	112
SHA3-256	256	1088	512	128
SHA3-384	384	832	768	192
SHA3-512	512	576	1024	256

The table above includes the output size, the bitrate, the capacity, and the security of each SHA3 variant. In this work, we focused only on the SHA3-512 variant. The newly developed algorithm can be used in many applications requiring a high level of security and integrity, such as online internet-based banking, shopping, and e-mail authentication. Different KECCAK hash function implementations have been elaborated using different hardware architectures performed using FPGA [13-14-15-16-17] and ASIC libraries.

In this paper, we consider the hardware/software codesign of the SHA3-512 KECCAK algorithm on the LEON3 processor. In fact, the LEON3/GRLIB source code is used to generate the bitstream code for the leon3 single-core processor in order to be implemented on a Xilinx Virtex-5 FPGA. To debug the leon3 based system-on-chip, we used the GRMON debug monitor. In this work, we used the GRMON3 to download, execute and debug the considered application on the leon3 soft processor. The paper is organized as follows: Section 2 presents the SHA3-512 algorithm specification. In section3, we detailed the SHA3-512 implementation on the LEON3 SPARC processor. Finally, section 4 concludes this paper.

II. The SHA3-512 KECCAK AlGORITHM

The KECCACK algorithm is a sponge-based construction to be applied to a fixed-length state S of b bits, where b equals r + c (r and c are, respectively, the bit rate and the capacity). A higher security and speed levels correspond, respectively, to higher c and r values. Fig.1 presents the state variables, which are the state, lane, slice, row, and column variables [12].

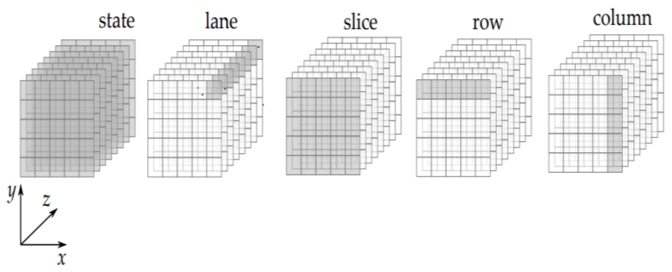


Fig. 1 The data structures used in KECCAK

In fact, the state S is a three-dimensional array with x, y, and z dimensions. All operations over x and y are taken modulo 5 while the operations over z are taken modulo w = b/25 (w= 64 for this case). The three dimensional state as a one-dimensional array state denoted by a and defined as $a(x,y,z)=s[w(x+5y)+z]$ such that $0 \leq x \leq 4$, $0 \leq y \leq 4$, and $0 \leq z \leq (w-1)$ [12].

The state consists of 5×5 lanes, each lane containing 64 bits. The state consists of 64 slices, and each slice contains 5*5=25 bits. For the SHA3-512, w is equal to 64, and b is equal to 1600. During the initialization and padding phases, the hash function transforms an input message with arbitrary length to a message with a fixed size. The sponge function consists of two phases: the absorbing phase and the squeezing one. The following algorithm [18] illustrates the KECCAK Sponge Function.

The KECCAK sponge function algorithm

```
Inputs: Msg (the message to be hashed)
The Initialization and Padding phase
S[x, y] = 0                                   ∀(x, y) in ([0, 4], [0, 4])
P= Msg||0x01||byte(d)||byte( r/8 )||0x01||0x00|| · · · ||0x00
The Absorbing phase
For each block Pi in P
    S[x, y]=S[x, y] ⊕Pi[x + 5y]               ∀(x, y) / x + 5y < r/w
    S= KECCAK−f[b](S)
The Squeezing phase
Z = empty string
while output is requested
    Z = Z||S[x, y]                            ∀(x, y) / x + 5y < r/ w
    S= KECCAK−f[b](S)
Output: Z
```

During the initialization phase, all the state S bits are set to zero. Then, during the padding phase, the data input is padded and divided into blocks of r bits. For the absorbing phase, the first r bits of the input message must be XORed with the first r bit states. Then, the output results will be interleaved using the permutation function. Lastly, all blocks are processed; the sponge design alters to the third phase. For the squeezing phase, the output blocks are the first r-bit of the state. The KECCAK-f function is illustrated by the following algorithm [18].

KECCAK−f function algorithm

```
Inputs: the state S and the constant values RC
for i = 0 to (n_r – 1)
    The Theta step
    C[x] = S[x, 0]⊕S[x, 1]⊕S[x, 2]⊕S[x, 3]⊕S[x, 4]        ∀x in [0, 4]
    D[x] = C[x − 1]⊕ROT(C[x + 1], 1))                      ∀x in [0, 4]
    S[x, y]=S[x, y]⊕D[x]                                   ∀(x, y) in ([0, 4], [0, 4])
    The Rho and Pi step
    B[y, 2x + 3y] = ROT(S[x, y], R[x, y])                  ∀(x, y) in ([0, 4], [0, 4])
    The Chi step
    S[x, y] = B[x, y]⊕((NOTB[x + 1, y]) AND B[x + 2, y])   ∀(x, y) in ([0, 4], [0, 4])
    The Iota step
    S[0, 0] = S[0, 0] ⊕ RC[i]
end for
Output: S
```

The SHA3-512 KECCAK-f hash function consists of n_r=24 identical rounds. For each round, five internal steps (Theta, Rho, Pi, Chi, and Iota) are executed. The KECCACK-f function is described as illustrated by the KECCAK−f function algorithm. For each round, the data input is mixed with the current state. S[x,y] represents a particular lane in the state. B[x,y], C[x], D[x] are intermediate variables, while RC[i] presents the round constant of the round i and R[x,y] is the offset constant rotation corresponding to the x and y values. Rot (S[ ],j) rotates one word S by j= R(x,y) bits positions [18]. The RC(i) constant values are given in [12]. Table II presents the padding steps and some sha3-512 C simulation results, including the padding steps, the data to be absorbed, the round 0 steps, and the final hash value.

TABLE II. THE PADDING STEPS, ROUND 0 STEPS, AND HASH VALUE FOR MESSAGE INPUT =MOUNA_NOURA_ATSIP_2020

The Padding Phase Steps

Phase 1: Converting the string message input to the hexadecimal form.

****

- Message input = Mouna_Noura_ATSIP _2020
- The encoded hexadecimal message input (MSG) = 4d6f756e615f4e6f7572615f41545349505f32303230
- Length(4d6f756e615f4e6f7572615f41545349505f32303230) =176 bits

Phase 2: Calculating **q** the number of bytes that must be appended to the original message in order to obtain a number of bits multiple of **r** (r=576 bits).

****

- The total number of padding bytes q is defined as follows [12]:

$$\mathbf{q = (r/8) - (m \bmod (r/8))}$$

where Length (Msg) = 8*m=176 bits
then m=176/8=22 and q=50>2

- The number of padding bytes q determines the hexadecimal form of the bytes.

Phase 3: Determining the padded message hexadecimal form

****

q	The padded Message
1	Msg || 0x86
2	Msg || 0x0680
>2	Msg || 0x06 || 0x00... || 0x80

- The final padded Message=

```
4d6f756e615f 4e6f7572615f41545349
505f32303230060000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0000000000000080
```

The Round 0 and the final hash value

Message input =Mouna_Noura_ATSIP_2020

*******Data to be absorbed*******

```
4d6f756e615f4e6f7572615f41545349
505f3230323006000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000800000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
0000000000000000
```

```
------------------ROUND 0------------------
-----------------After THeta-----------------

a78bb7d0e3f7e8fd98a37051446b1126
242d536f73645549505f323032300600
9adeeadcc2be9c5eeae4c2be82a8a692
edd1110e053f426f7472615f41545349
505f3230323006809adeeadcc2be9c5e
eae4c2be82a8a692edd1110e053f426f
7472615f41545349505f323032300600
9adeeadcc2be9c5eeae4c2be82a8a692
edd1110e053f426f7472615f41545349
505f3230323006009adeeadcc2be9c5e
eae4c2be82a8a692edd1110e053f426f
7472615f41545349505f323032300600
9adeeadcc2be9c5e
```

```
--------------------After Rho&Pi-------------------------

a78bb7d0e3f7e8fde150f023f4d61e1d
fb0aa29a4aa2930bc60000ea4b064606
a797a6b73ab7b02f03630000f5250323
cbe9a5e9adce2dec542717f615443595
c2a1e047e8ad3d3a4e2eec2b886a2a89
3047e1a288d6224c129d5cd85710d554
600c00a0be646064 5e9adeeadcc2be9c
9a4aaa930bfb0aa2f6e5f4d2f456e716
2b886a2aa94e2eecbdb547473814fc08
a9243ab9b0af20aa5f32303230060050
49cbd4db1c595122f191819180340a8
6e615f4e2f4d6f757d05514d25d5c985
b547473814fc08bd
```

```
----------------After CHI--------------------
bd81b548e9d769ffe550f043f5d25a19
da9d048f7a132322c60811aa8a460ed6
e7c7e6942eb7a62f17651216e5251332
496945e8456725c658291bde15063714
c3e0e0479da83c1886a649c280a00645
5047e18220b2026c0c0f829217924bcc
e04c20b1bd5d60467e9f9fca5cc69ed0
98d2b6cb5cfbdfb262d0f197e4463716
2b88529229e52e4eeba747453814fc58
09e1fe7974ffc7ac563a3a1a390e08b8
09ab939d3b157a473e1d18181893c028
ee23597e3f656f4d358dc18e2dd49c87
93574f3814fe0815
```

```
---------------After IOTA-------------------
bc81b548e9d769ffe550f043f5d25a19
da9d048f7a132322c60811aa8a460ed6
e7c7e6942eb7a62f17651216e5251332
496945e8456725c658291bde15063714
c3e0e0479da83c1886a649c280a00645
5047e18220b2026c0c0f829217924bcc
e04c20b1bd5d60467e9f9fca5cc69ed0
98d2b6cb5cfbdfb262d0f197e4463716
2b88529229e52e4eeba747453814fc58
09e1fe7974ffc7ac563a3a1a390e08b8
09ab939d3b157a473e1d18181893c028
ee23597e3f656f4d358dc18e2dd49c87
93574f3814fe0815
```

```
hash=

e7   92   89   68   76   64   e3   d0
dc   f5   b1   c1   de   ba   54   5f
97   66   e5   2f   b0   f1   a8   ae
1c   6d   ce   5e   53   e8   df   74
7e   f6   81   eb   41   1c   88   b5
70   c0   28   09   c4   de   bd   b1
70   0b   c6   49   fd   af   90   ac
30   3b   4d   d6   bb   ac   58   b8
```

III. THE SHA3-512 IMPLEMENTATION ON LEON3 SPARC PROCESSOR

For IP cores and tools development, Gaisler is a world leader for commercial and aerospace embedded processors based on the SPARC architecture [11]. The LEON3 processor is a SPARC-V8 synthesizable open-source VHDL core. This Gaisler product includes a full development environment and a GRLIB IP library. The LEON3 template design [19] is extremely configurable and adequate for system-on-a-chip (SOC) designs. The source code is available under the GNU GPL license. In addition, LEON3 is available under a low-cost commercial license, allowing it to be used in commercial applications to a fraction of the comparable IP core cost. The considered processor is suitable for embedded applications, combining low power consumption and low complexity with high-performance capability. The LEON3 open source core is available for free from open-source Gaisler Research communities [20]. In this work, we realized a hardware/software SHA3-512 analysis and implementation on the LEON3 soft core processor. The adapted codesign flow is illustrated in Fig. 2 [21].

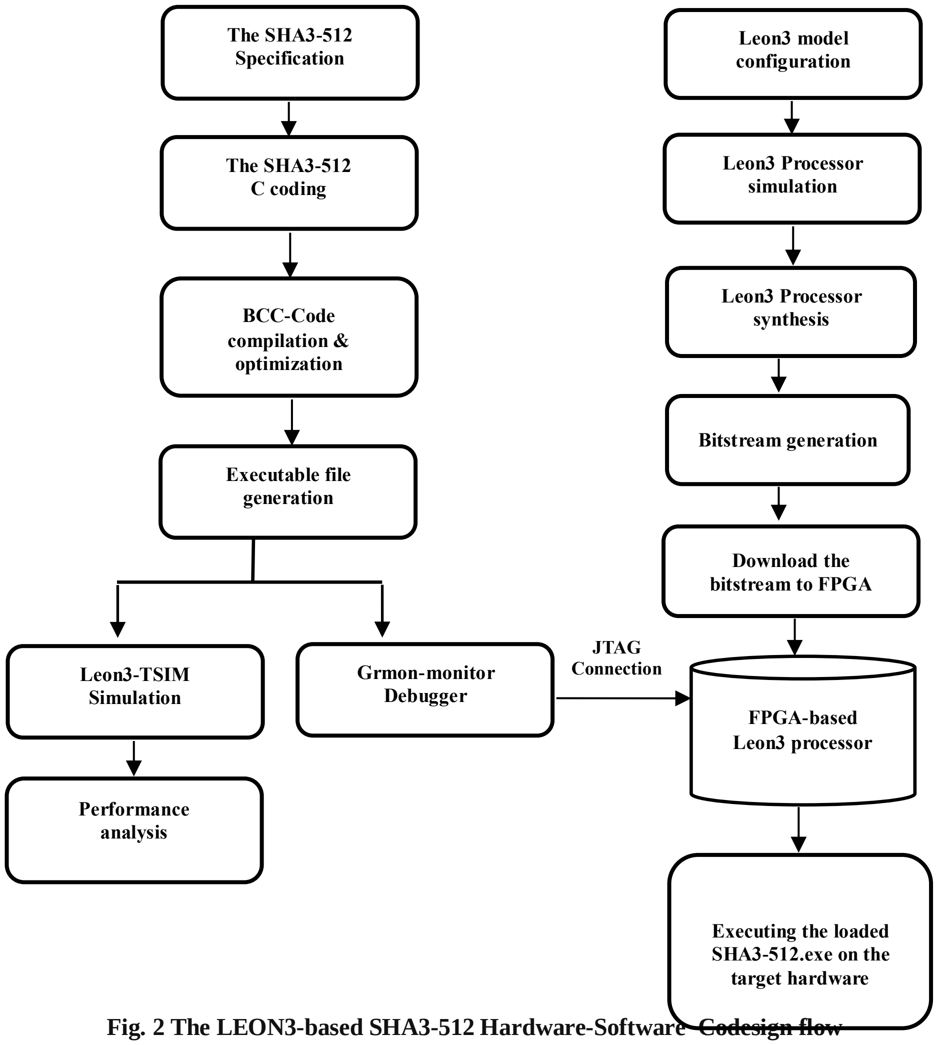
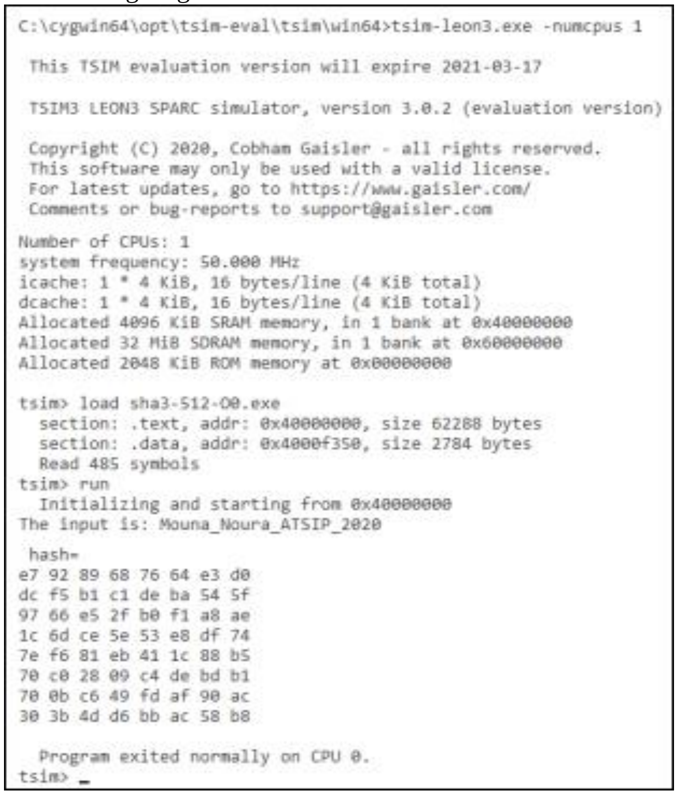

The SHA3-512 Specification
The SHA3-512 C coding
BCC-Code compilation & optimization
Executable file generation
Leon3-TSIM Simulation
Performance analysis
Grmon-monitor Debugger
JTAG Connection
Leon3 model configuration
Leon3 Processor simulation
Leon3 Processor synthesis
Bitstream generation
Download the bitstream to FPGA
FPGA-based Leon3 processor
Executing the loaded SHA3-512.exe on the target hardware

Fig. 2 The LEON3-based SHA3-512 Hardware-Software Codesign flow

The LEON3 processor hardware configuration is realized using the GRLIB implementation tools. In fact, Gaisler provides LEON3 template designs for many FPGA boards. The model is extremely configurable and uses the AMBA 2.0 Advanced High-Speed Bus (AHB) interfacing with other IP cores. The LEON3 implementation can be realized on Windows using the Cygwin tool or on Linux operating system. In this work, we used the Cygwin environment. In fact, the LEON3 design configuration is defined through a configuration file automatically generated using a Graphical

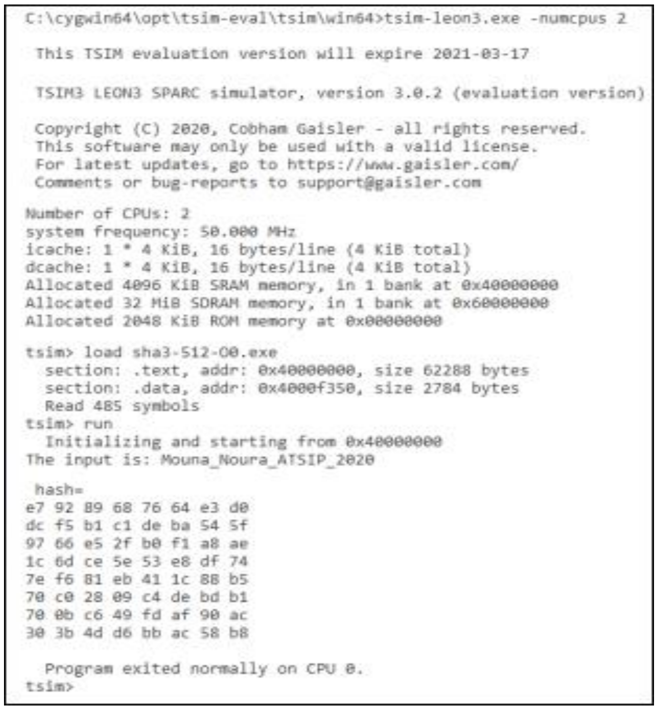
User Interface (GUI) based on tkconfig and using the command ‘make xconfig’. Once the configuration step is completed, the bitstream file is generated using the Xilinx ISE design tool and loaded onto the xc5vf70t xilinx FPGA using a JTAG cable. On the other hand, in order to compile the SHA3-512 code, we use the Bare-C Cross Compiler (BCC). It is based on the GNU compiler tools and Newlib standalone C-library. This cross-compiler allows the compilation of C and C++ applications and can also be used to compile eCos kernels [21-22]. Once the executable file is generated (the SHA3-512.exe), we analyze the application performance using TSIM, which is a generic SPARC architecture simulator for emulating LEON-based embedded systems. The primary site for TSIM is the Cobham Gaisler website where the latest version of TSIM can be ordered and evaluation versions can be downloaded. In fact, the last evaluation version is available online since 18 September 2020 and simulates a basic dual-core LEON3 system and can be made to simulate a single core system using the ‘-numcpus’ option [23].

In the following section, we will analyze our application performance using the 3.2.0 TSIM version in order to simulate the SHA3-512 in a single and dual-core LEON-based computer system. Fig.3 and Fig.4 presented, respectively, the considered SHA3-512 TSIM simulation results using single and dual-core LEON3.



```
C:\cygwin64\opt\tsim-eval\tsim\win64>tsim-leon3.exe -numcpus 1

 This TSIM evaluation version will expire 2021-03-17

 TSIM3 LEON3 SPARC simulator, version 3.0.2 (evaluation version)

 Copyright (C) 2020, Cobham Gaisler - all rights reserved.
 This software may only be used with a valid license.
 For latest updates, go to https://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

Number of CPUs: 1
system frequency: 50.000 MHz
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
Allocated 4096 KiB SRAM memory, in 1 bank at 0x40000000
Allocated 32 MiB SDRAM memory, in 1 bank at 0x60000000
Allocated 2048 KiB ROM memory at 0x00000000

tsim> load sha3-512-O0.exe
  section: .text, addr: 0x40000000, size 62288 bytes
  section: .data, addr: 0x4000f350, size 2784 bytes
  Read 485 symbols
tsim> run
  Initializing and starting from 0x40000000
The input is: Mouna_Noura_ATSIP_2020
 hash=
e7 92 89 68 76 64 e3 d0
dc f5 b1 c1 de ba 54 5f
97 66 e5 2f b0 f1 a8 ae
1c 6d ce 5e 53 e8 df 74
7e f6 81 eb 41 1c 88 b5
70 c0 28 09 c4 de bd b1
70 0b c6 49 fd af 90 ac
30 3b 4d d6 bb ac 58 b8

  Program exited normally on CPU 0.
tsim> _
```

Fig. 3 The SHA3-512 TSIM 3.2 simulation using a single core LEON3



```
C:\cygwin64\opt\tsim-eval\tsim\win64>tsim-leon3.exe -numcpus 2

 This TSIM evaluation version will expire 2021-03-17

 TSIM3 LEON3 SPARC simulator, version 3.0.2 (evaluation version)

 Copyright (C) 2020, Cobham Gaisler - all rights reserved.
 This software may only be used with a valid license.
 For latest updates, go to https://www.gaisler.com/
 Comments or bug-reports to support@gaisler.com

Number of CPUs: 2
system frequency: 50.000 MHz
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
Allocated 4096 KiB SRAM memory, in 1 bank at 0x40000000
Allocated 32 MiB SDRAM memory, in 1 bank at 0x60000000
Allocated 2048 KiB ROM memory at 0x00000000

tsim> load sha3-512-O0.exe
  section: .text, addr: 0x40000000, size 62288 bytes
  section: .data, addr: 0x4000f350, size 2784 bytes
  Read 485 symbols
tsim> run
  Initializing and starting from 0x40000000
The input is: Mouna_Noura_ATSIP_2020
 hash=
e7 92 89 68 76 64 e3 d0
dc f5 b1 c1 de ba 54 5f
97 66 e5 2f b0 f1 a8 ae
1c 6d ce 5e 53 e8 df 74
7e f6 81 eb 41 1c 88 b5
70 c0 28 09 c4 de bd b1
70 0b c6 49 fd af 90 ac
30 3b 4d d6 bb ac 58 b8

  Program exited normally on CPU 0.
tsim>
```

Fig. 4: The SHA3-512 TSIM 3.2 simulation using a single core LEON3

From Fig.3 and Fig.4, we can conclude that the SHA3-512 application is running successfully on a single and dual-core LEON3 simulator. The two simulations are executed using the same executable file on the same machine and without using any optimization level. The considered SHA3-512 performance analysis without using any compiler optimization is reported in Table III. Actually, compiler optimization techniques can be used for resource-constrained embedded systems in order to enhance the executable programs efficiency [24]. Compiler optimizations have been used to improve performance, and researchers have investigated the effect of code optimizations on energy consumption and system reliability. In fact, they found that compiler optimizations decrease the energy consumption since they decrease the machine instructions number needed to execute a computation [25]. In addition, the reduction of the executed instructions number reduces the probability of program affection by transient hardware errors [26]. This study proves that compiler optimizations had only a minor impact on the error sensitivity of the investigated benchmark programs. Thus, in the current work, we consider the SHA3-512 algorithm implementation on the LEON3 softcore processor, and we rely on compiler optimization techniques to enhance the efficiency of executable programs. Hence, before generating the executable file, we used the GCC compiler to compile the considered SHA3-512 application with different optimization levels. Once each executable file is generated the

The considered application is ready to be loaded into the LEON3 processor using a JTAG-based GRMON monitor connection. To run and download the SHA3-512 software on the LEON3, we have to use the GRMON Gaisler tools. GRMON allows the communication with the processor for non-intrusive monitoring and debugging and providing full access to internal peripherals. Thus, in order to communicate with GRMON via a serial cable (JTAG), we include a debug support unit (DSU) when elaborating the LEON3 model configuration. GRMON is also a general debug monitor for RISC-V and SOC-designs based on the GRLIB IP library [11]. It supports USB, JTAG, UART, Ethernet, and SpaceWire debug links and includes different functions such as downloading and execution of LEON applications and Read/write access to all memories and system registers [22]. On the target hardware, the debug interface for the LEON3 can be of various types, such as JTAG, Ethernet, and SpaceWire debug interfaces. Fig.5 presents the GRMON Debug Monitor interfaced with the implemented LEON3 processor via JTAG connection.

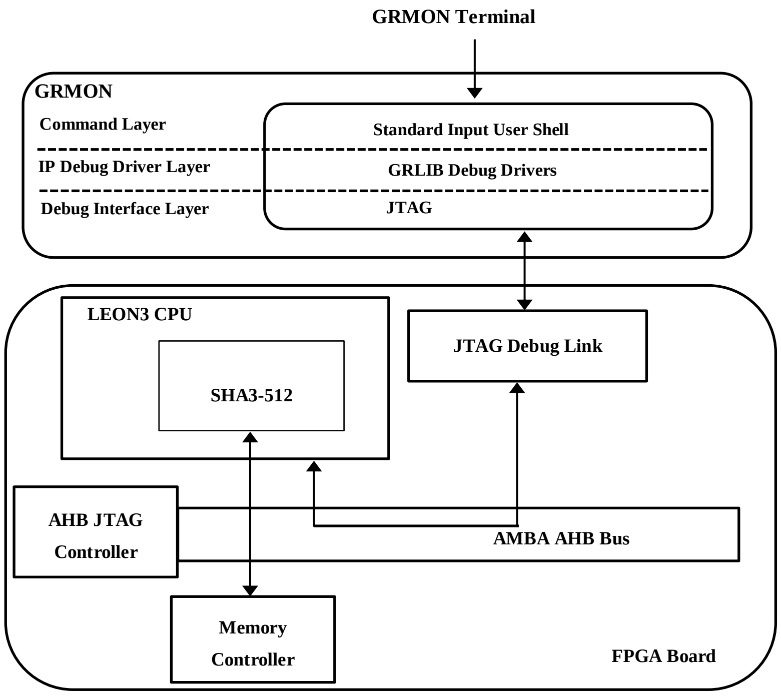
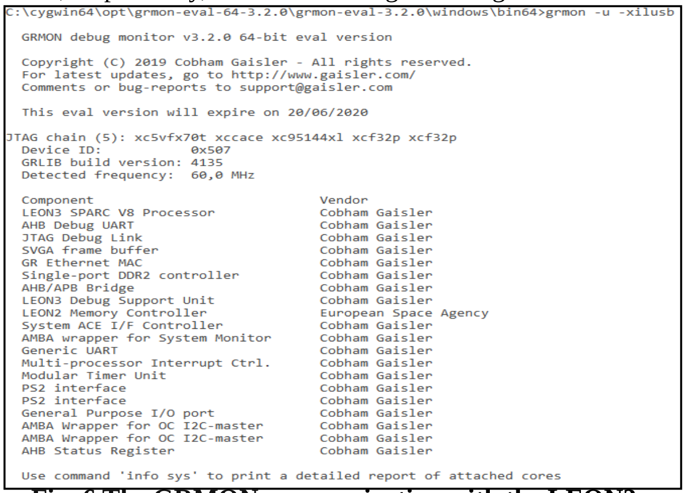


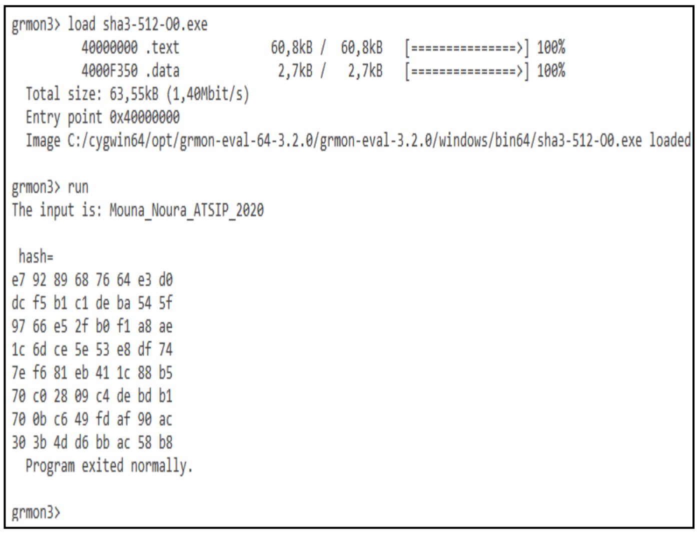
Fig. 5 The SHA3-512 GRMON Debugging on the implemented LEON3-processor

As indicated in Fig.5, the communication with the LEON3 processor implemented on the xc5vf70t Xilinx FPGA is ensured via JTAG cable through the GRMON debug monitor. The GRMON communication with the LEON3 processor and the sha3-512 loading and execution are, respectively, illustrated in Fig.6 and Fig.7.



```
C:\cygwin64\opt\grmon-eval-64-3.2.0\grmon-eval-3.2.0\windows\bin64>grmon -u -xilusb

  GRMON debug monitor v3.2.0 64-bit eval version

  Copyright (C) 2019 Cobham Gaisler - All rights reserved.
  For latest updates, go to http://www.gaisler.com/
  Comments or bug-reports to support@gaisler.com

  This eval version will expire on 20/06/2020

JTAG chain (5): xc5vfx70t xccace xc95144xl xcf32p xcf32p
  Device ID:           0x507
  GRLIB build version: 4135
  Detected frequency:  60,0 MHz

  Component                            Vendor
  LEON3 SPARC V8 Processor             Cobham Gaisler
  AHB Debug UART                       Cobham Gaisler
  JTAG Debug Link                      Cobham Gaisler
  SVGA frame buffer                    Cobham Gaisler
  GR Ethernet MAC                      Cobham Gaisler
  Single-port DDR2 controller          Cobham Gaisler
  AHB/APB Bridge                       Cobham Gaisler
  LEON3 Debug Support Unit             Cobham Gaisler
  LEON2 Memory Controller              European Space Agency
  System ACE I/F Controller            Cobham Gaisler
  AMBA wrapper for System Monitor      Cobham Gaisler
  Generic UART                         Cobham Gaisler
  Multi-processor Interrupt Ctrl.      Cobham Gaisler
  Modular Timer Unit                   Cobham Gaisler
  PS2 interface                        Cobham Gaisler
  PS2 interface                        Cobham Gaisler
  General Purpose I/O port             Cobham Gaisler
  AMBA Wrapper for OC I2C-master       Cobham Gaisler
  AMBA Wrapper for OC I2C-master       Cobham Gaisler
  AHB Status Register                  Cobham Gaisler

  Use command 'info sys' to print a detailed report of attached cores
```

Fig. 6 The GRMON communication with the LEON3 processor



```
grmon3> load sha3-512-O0.exe
         40000000 .text            60,8kB /  60,8kB   [===============>] 100%
         4000F350 .data             2,7kB /   2,7kB   [===============>] 100%
  Total size: 63,55kB (1,40Mbit/s)
  Entry point 0x40000000
  Image C:/cygwin64/opt/grmon-eval-64-3.2.0/grmon-eval-3.2.0/windows/bin64/sha3-512-O0.exe loaded

grmon3> run
The input is: Mouna_Noura_ATSIP_2020

 hash=
e7 92 89 68 76 64 e3 d0
dc f5 b1 c1 de ba 54 5f
97 66 e5 2f b0 f1 a8 ae
1c 6d ce 5e 53 e8 df 74
7e f6 81 eb 41 1c 88 b5
70 c0 28 09 c4 de bd b1
70 0b c6 49 fd af 90 ac
30 3b 4d d6 bb ac 58 b8
  Program exited normally.


grmon3>
```

Fig. 7 The sha3-512 loading and execution

As indicated in a previous section, the GCC compiler (a BCC package) is used to elaborate on different levels of optimization (-O, -O1, -O2, etc..) to optimize the code performance and size. Table III. summarized all TSIM and GRMON results obtained for each level of optimization.

Table III. The SHA3-512 Implementation Performance Analysis

SHA3-512 Performance Analysis on a single core LEON3 using TSIM 3.2.0

Optimization Level	Without optimization	-O/-O1	-O2	-Os
Cycles	429440	315478	304087	306006
Instructions	236171	169742	163389	165334
Overall CPI	1.82	1.86	1.86	1.85
CPU performance (@ 50.0 MHz)	27.50 MOPS	26.90 MOPS	26.87 MOPS	27.01 MOPS
Cache hit rate (%)	97.3	97.0	96.8	96.9

SHA3-512 Performance Analysis on a dual-core LEON3 using TSIM 3.2.0

Optimization Level	Without optimization	-O/-O1	-O2	-Os
Cycles	430083	316117	304744	306660
Instructions	236516	170087	163734	165679

SHA3-512 GRMON Debugging Analysis

Optimization Level	Without optimization	-O/-O1	-O2	-Os
Text Address	0x40000000	0x40000000	0x40000000	0x40000000
Data Address	0x4000F350	0x4000DB70	0x4000C2F0	0x4000C2B0
Total size (kB)	63,55	57,58	51.45	51,39
Throughput (Mbit/s)	1.40	1,39	1,27	1,27

Table III presented the SHA3-512 performance analysis where the number of cycles and instructions, the overall CPI (Cycles per instruction), the CPU performance in MOPS (Millions of Operations Per Second), and the Cache hit rate (%) are indicated for both single and dual-core LEON3 using the 3.2.0 TSIM version. The SHA3-512 GRMON debugging analysis is also presented for different optimization levels. By comparing the single and dual-core TSIM simulation results, we remark that the presented results show only a little enhancement, made by the dual-processor architecture, for the number of cycles and instructions. These results can be explained by the fact that the SHA3-512 code was not written to exploit parallel architectures, and this is also due to the compiler non-optimization for multicore architectures. As seen in Table III, the unoptimized compilation (-O0 optimization level) takes more time and memory than any optimized one. The -O0 generates un-optimized code but has the fastest compilation time. Using the -O or -O1 optimizer, the compiler attempt to reduce the code size and the execution time. The -O2 optimizes even more. The compiler performs all the -O supported optimizations and uses more aggressive automatic sub-programs inlining and loops vectorization. This option increases both compilation time and performance of the generated code compared to the –O optimization level. The -Os optimizes for size (the total size) and enables all -O2 optimizations except those that often increase code size

IV. CONCLUSIONS

Hash functions are used for many cryptographic applications, such as digital signatures and message authentication codes. This paper presents a detailed analysis of the SHA3-512 cryptographic hash algorithm and its implementation on the LEON3 soft core processor. Single and dual-core performance analyses based on TSIM simulation and GRMON debugging tools are presented. In addition, compiler optimization techniques are used to improve the efficiency of executable programs. The simulation results demonstrate the performance benefits of using compiler optimizations for computer-based embedded systems. However, we observe that the performance gain for multicore architecture is not so considerable since the used SHA3-512 code was not designed to exploit a multicore architecture. Hence, this work can be extended by multi-threading the SHA3-512 code in order to be simulated and debugged on multicore and multiprocessor-based architectures.

REFERENCES

[1] D. R. Stinson and M. B. Paterson, “Cryptography: theory and practice. Boca Raton: CRC Press, (2017).

[2] M.harran, W.Farrelly, K.Curran, A method for verifying integrity & authenticating digital media, Applied Computing and Informatics, 14(2) (2018) 145-158.

[3] V. Pachghare, Cryptography, and information security. PHI Learning Pvt. Ltd., (2019).

[4] Xiaoyun Wang, Yiqun Lisa Yin, et Yu Hongbo, Finding collisions in the full SHA-1, in Annual International Cryptology Conference, Springer, Berlin, Heidelberg, (2005) 17–36.

[5] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512., in ISC (A. H. Chan and V. D. Gligor, eds.), 2433 of Lecture Notes in Computer Science, Springer, (2002) 75–89.

[6] Xiaoyun Wang, et Yu Hongbo, How to break MD5 and other hash functions, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, Berlin, Heidelberg, (2005) 19–35.

[7] R. Martino and A. Cilardo, SHA-2 Acceleration Meeting the Needs of Emerging Applications: A Comparative Survey, in IEEE Access, 8, (2020) 28415-28436.

[8] Kazumaro Aoki, Jian Guo, Krystian Matusiewicz, et al., Preimages for step-reduced SHA-2, in International Conference on the Theory and Application of Cryptology and Information Security, Springer, Berlin, Heidelberg, (2009) 578–597.

[9] Richard F. Kayser, Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family, Fed. Regist. 72 (212) (2007).

[10] Turan, Meltem So€nmez, Ray Perlner, Lawrence E. Bassham, et al., Status report on the second round of the SHA-3 cryptographic hash algorithm competition, NIST Interagency Report (2011) 7764.

[11] Martin Åberg; Development of an RV64GC IP core for the GRLIB IP Library,2nd-RISC-V-Meeting-2019-10-01Paris

[12] Dworkin, M, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, (2015) https://doi.org/10.6028/NIST.FIPS.202.

[13] K. Latif, M. Muzaffar Rao, A. Aziz and A. Mahboob, Efficient hardware implementations and hardware performance evaluation of SHA-3 finalists, NIST Third SHA-3 Candidate Conf., Washington, DC, March, (2012) 22–23.

[14] S. Bayat-Sarmadi, M. Mozaffari-Kermani, and A. Reyhani-Masoleh, Efficient and Concurrent Reliable Realization of the Secure Cryptographic SHA-3 Algorithm, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33(7), (2014) 1105-1109.

[15] F. Kahri, H. Mestiri, B. Bouallegue, M. Machhout High-Speed FPGA Implementation of Cryptographic KECCAK Hash Function CryptoProcessor, Journal of Circuits, Systems, and Computers , 25 (4) (2016).

[16] E. Homsirikamol, M. Rogawski, and K. Gaj Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs, Cryptology ePrint Archive Report, George Mason University, (2010).

[17] Deepthi Barbara Nickolas , Mr. A. Sivasanka."Design of FPGA Based Encryption Algorithm using KECCAK Hashing Functions". International Journal of Engineering Trends and Technology (IJETT). 4(6) (2013) 2438-244.

[18] C.Paar, J. Pelzl SHA-3, and The Hash Function Keccak, Springer, An extension chapter for Understanding Cryptography — A Textbook for Students and Practitioners, (2010).

[19] J.Gaisler and M.Isomak. LEON3 GR-XC3S-1500 template design, Copyright Gaisler Research, (2006) 1-153.

[20] Gaisler Research, GRLIB IP Core User’s Manual, Version 2019.4, (2019).

[21] M. Karmani, N. Benhadjyoussef, B. Hamdi and M. Machhout.A Hardware-Software Codesign Case Study: The SHA3-512 algorithm Implementation on the LEON3 Processor, 2020 5th International Conference on Advanced Technologies for Signal and Image Processing (ATSIP), Sousse, Tunisia, (2020).

[22] Gaisler Research, BCC User's Manual, Version 2.1.0, November (2019).

[23] Gaisler Research, TSIM3 Simulator User's Manual, Version 3.0.2, September (2020).

[24] B. Sangchoolie, F. Ayatolahi, R. Johansson and J. Karlsson, A Study of the Impact of Bit-Flip Errors on Programs Compiled with Different Optimization Levels, 2014 Tenth European Dependable Computing Conference, Newcastle, UK, (2014) 146-157.

[25] G. Nazarian, C. Strydis and G. Gaydadjiev, Compatibility Study of Compile-Time Optimizations for Power and Reliability, in 14th Euromicro Conf. on Digital System Design (DSD), Oulu, Finland, (2011).

[26] T. C. May, M. H. Woods, Alpha-Particle-Induced Soft Errors in Dynamic Memories, IEEE Transactions on Electron Devices, vol. ED-26, no. 1, (1979) 2-9.