HW/SW Architecture Exploration for an Efficient Implementation of the Secure Hash Algorithm SHA-256

Manel Kammoun, *Member, IEEE,* Manel Elleuchi, *Member, IEEE,* Mohamed Abid, *Member, IEEE,* and Abdulfattah M. Obeid, *Member, IEEE*

***Abstract*—Hash functions are used in the majority of security protocol to guarantee the integrity and the authenticity. Among the most important hash functions is the SHA-2 family, which offers higher security and solved the insecurity problems of other popular algorithms as MD5, SHA-1 and SHA-0. However, theses security algorithms are characterized by a certain amount of complex computations and consume a lot of energy. In order to reduce the power consumption as required in the majority of embedded applications, a solution consists to exploit a critical part on accelerator (hardware). In this paper, we propose a hardware/software exploration for the implementation of SHA256 algorithm. For hardware design, two principal design methods are proceeded: Low level synthesis (LLS) and high level synthesis (HLS). The exploration allows the evaluation of performances in term of area, throughput and power consumption. The synthesis results under Zynq 7000 based-FPGA reflect a significant improvement of about 80% and 15% respectively in FPGA resources and throughput for the LLS hardware design compared to HLS solution. For better efficiency, hardware IPs are deduced and implemented within HW/SW system on chip. The experiments are performed using Xilinx ZC 702-based platform. The HW/SW LLS design records a gain of 10% to 25% in term of execution time and 73% in term of power consumption.**

***Index Terms*—SHA256, Zynq 7000 based-FPGA, LLS, HLS.**

I. Introduction

Nowadays, the Field-Programmable Gate-Array (FPGA) is becoming a good alternative to the application-specific integrated circuit (ASICs) especially when dealing with such complex implementation like image or signal processing applications [1]. Indeed, thinks to the progress brought on programmable circuits, it becomes possible to design a System on Chip (SoC) component based on single or multiple processors and a programmable logic. This kind of system could be exploited in a wide range of applications for its flexibility, short time to market, low power and high capacity of integration. In addition, the reconfigurability of FPGA circuits boosts the designers to implement their own programs using a Hardware Description Language (HDL) and also to make several optimizations on the hardware architecture.

For decades, the Low-Level Synthesis (LLS) has been adopted as a design method for FPGA implementations as it is more reliable and requires an explicit coding of the control path. This option leads to better improve design capabilities by optimizing whatever parameters. Nevertheless, the designing of the final netlist takes much time an effort, practically for the case of complex algorithms. At this level, the hardware developers are front of a new challenge where many constraints should be taken into account to fulfill the market requirement. Consequently, it is time to think about new design methods which can help to economize timing constraint and facilitate the implementation task on FPGAs. The solution is to raise the abstraction level from LLS to High Level Synthesis (HLS) using a specific high level description language (Matlab, C/C++, etc...). For many reasons, the HLS becomes more and more useful than LLS [2] [3]. One of the key benefit of working with HLS is the ability to simulate multiple algorithms in the shortest times [4]. Moreover, modern HLS tools such as (Vivado HLS [5], Catapult-C [6], etc..) are able to provide an estimative report of area cost, frequency and latency time more quickly. Also, several optimizations can be exploited at the level of C function to better improve design performances in term of throughput and hardware cost. For instance, the usage of pipeline an unroll pragmas can help to reach higher throughputs at the cost of increasing logical gates. However, there are some restrictions that should be held on before working with HLS tools. First, it is not a simple conversion from high level language to RTL level. In fact, the code must be re-written with a specific way to be correctly implemented on FPGA platform. Second, some particular C instructions based on pointer and recursion are not synthesizable and can cause memory overhead in the context of FPGA. All these reasons prevent designers to go over optimizing more their architectures. Consequently, the main focus of this work will be devoted to study the influence of LLS and HLS design methods when facing a such computational application like cryptographic algorithm.

Most of application domains use secure techniques and algorithms to protect them from attacks while respecting the se-

Manuscript received January 7, 2021; revised April 1, 2021. Date of publication April 29, 2021. Date of current version April 29, 2021. The associate editor prof. Toni Perković has been coordinating the review of this manuscript and approved it for publication.

M. Kammoun is with the Digital Research Center of Sfax, Sfax, Tunisia. M. Elleuchi and M. Abid are with the CES Research Laboratory, National Engineering School of Sfax, Digital Research Center of Sfax, Sfax, Tunisia. A. M. Obeid is with the National Center for Electronics, Communications and Photonics at KACST (e-mails: manelkammounenis@gmail.com, manelelleuchi@gmail.com, mohamed.abid_ces@yahoo.fr, obeid@kacst.edu.sa).

Digital Object Identifier (DOI): 10.24138/jcomss-2021-0006

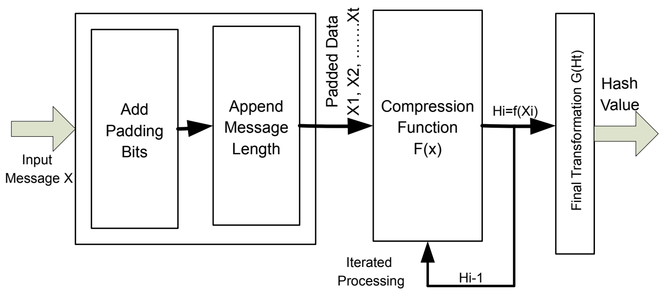


Fig. 1. The general model of hash function.

curity requirements. Several security protocol and frameworks are based on symmetric cryptographic techniques, Asymmetric cryptographic techniques, hybrid cryptography techniques and hash functions [7]. Hash functions are the most important key to keep data safe and secure. They are used as building blocks in various cryptographic and security services such as electronic commerce, digital signature and information authentication. SHA-2 is a family of hash functions that were designed by the US National Security Agency (NSA), based on SHA-1 and SHA-0. Like most of the hash functions, the SHA-2 takes as input a message of arbitrary size and produces a fixed size output.Fig. 1 represents a general model of this type of functions [8].

The size of hash is indicated by the suffix: 224 bits for SHA-224, 256 bits for SHA-256 and 512 bits for SHA-512.For instance, SHA-256 is a type of secure hash operation under the SHA-2 [9] banner with digest length of 256 bits. This family of hashing algorithms uses large digest messages making them more resistant to possible attacks and allowing them to be used with big amount of data blocks, up to 2128bits in the case of SHA-512.

The SHA-256 presents a model of hash algorithm that posses many computational rounds. In this context, there is a proliferation of various works which were trying to optimize the complexity of SHA-256 function using the hardware accelerators as provided in [10] and [11]. However, these solutions suffer from lack of flexibility and performance degradation. To overcome these deficiencies, two design methods are adopted in this work based on LLS and HLS synthesis. With system-on-chip (SoC) designs growing in complexity, system-level approaches that leverage on HLS and LLS techniques are becoming the workhorse of current SoC design flows. These solutions provide reasonable agreements in term of FPGA resources, throughput and power consumption. To highlight our contribution, the proposed LLS and HLS accelerators are integrated in HW/SW context in order to estimate performances in term of execution time and power consumption.

The remainder of this paper is organized as follows. Section II introduces some related works which had implemented SHA-256 algorithm under FPGA platform. Section III presents our proposed architectures implemented on the Xilinx ZC 702 evaluation board [12]. The experimental findings of the HW/SW implementations in term of throughput and power consumption are discussed in section IV. Finally, the conclusion and the futures works are provided in Section V.

II. Related Works

Several works had experimented the implementation of cryptographic hash functions on FPGA-based platform. For instance, the example given in [13] proposed an improved schemas of the SHA-256 algorithm implemented on Virtex-2 XC2VP-7. These designs were based on the rearrangement technique to compute the inner loop of the SHA-256 hash function such as computing values in advance and changing the control path without increasing the clock cycles. In best case, the maximum throughput achieved in this work was about 909 Mbps with an efficiency of 0.713 Mbps/ slice.

In [14], the authors reported a parallel architecture for an efficient usage of encryption/ decryption modules. The synthesis was done on Virtex 5 based platform which provided a rate of 405 Mbps for the SHA-256 implementation.

Furthermore, in [15] a design of SHA processor was described which implemented the three hash algorithms SHA-512, SHA-512/224 and SHA-512/256 in both Virtex-5 and Virtex-4 LX FPGA chips. The main purpose of its architecture is to reuse data to keep a high efficiency, minimize critical path and reduce the memory access through using cache memory. The implementation results demonstrated that the proposed design used fewer resources achieving higher performance and efficiency. Otherwise; the data transfer speed is around 50 Mbps.

In addition, a multi-mode architecture is presented in [16] which are able to perform either a SHA-384 or SHA-512 hash algorithm or to treat two independent SHA-224 and SHA-256 blocks. The main goal of this approach is to minimize remarkably the computational overhead with zero time latency caused by the processing of the input message. However, the maximum throughput achieved by the SHA-256 hardware block can only reach 308 Mb/s.

Another VLSI architecture is provided in [17] which can support three hash functions (256, 384 and 512).The principal contribution of this work is the allocation of the same area resources for all hash algorithms which can affect the speedup of the global design (about 291Mbps in case of SHA-256). There are also some other works focusing on SHA-256 hardware implementation such as [18] and [19]. For instance, in [18], authors implemented the SHA-256 secure hash algorithm in both Virtex-5 and Virtex-4 LX FPGA chips. The purpose of its architecture is to exploit data reuse technique to keep high efficiency, minimize critical paths and reduce memory access. The synthesis results using Virtex 5 device demonstrated a fewer FPGA resources in use while the data transfer speed was around 50 Mbps.

On the other hand, a Totally Self-Checking (TSC) design was implemented in [19] on Virtex 5 XC5VLX330 FPGA device. Hence, the different components of the SHA-256 function such as the Rotation/Shift registers and Multiplexers as well as the counter and addition components should obey to the described TSC rules. Moreover, a TSC system, even though it introduces a penalty in performance and in area consumption, is more efficient in term of throughput compared to the existing solutions as it can produce a throughput level up to 3.88 Gbps.

In contrast, the HLS was adopted as a design method in diverse fields such as financial [20], video coding [21] and stereovision [22] algorithms. Nevertheless, the number of published works of secure algorithms admitting HLS method is relatively tenuous. In this case, the hardware proposed in [23] is designed using Vivado HLS tool under Xilinix Zynq 7000 SoPC. After adding the suitable optimizations, it was capable to operate 1088 bits in 70 clock cycles.

At light of the above finding, we note that the recourse proposed solutions are entirely developed in hardware which allows achieving higher throughputs at the cost of affecting the flexibility of the design. Therefore, it is necessary to make into account the synchronization between hardware IP and bus interface throughputs when dealing with processor and FPGA. In this context, the next section will be devote to develop the hardware implementation of the SHA256 hash functions using low-level and high-level design methods under Zynq 7000 SoC platform.

III. Toward Efficient HW/SW Implementation

Several design methods can be explored to perform the implementation of the SHA-256 hash function. Usually, the SW solutions are more flexible and don't require a lot of time to verify and validate the IP which is not the case of the HW implementation. This last is more tended to satisfy real time constraint at low power cost rather than software at the cost of increasing the simulation time. In order to ensure the best trade-off between flexibility and performances, the HW/SW concept is considered as a best solution which combines a microprocessor system and a programmable logic both in the same chip.

Thereby, this section discusses the different proposed solutions (SW, HW and SW/HW) for the implementation of the SHA-256 hash algorithm. After studying the whole operation of the hash function, this last is implemented in SW environment using ARM Cortex A9 processor in order to estimate the most consuming part in the SHA-256 function. Based on profiling results, diverse hardware solutions are developed for the implementation of the critical function.

A. SHA-256: Specification and Complexity

The concept of cryptographic hash function consists of assigning a single relationship between the input message and the hash value. The ideal cryptographic hash algorithm should satisfy some criteria. At first, it should be hard or infeasible to invert a hash function in such a way that the hash output value h produces an input message *M* such that *H(M) = h*. Second, given an input m_1, it is difficult to produce the same hash value with another input value m_2. This feature refers to a weak collision resistance. The iterative structure is another property specific to hash security functions where the hash value of the current block is computed using the digest message of the previous block [24]. This leads to make the compression function output more secure and collision resistant. Thanks to these advantages, hash functions are today widely exploited in real life applications such as MD5 [25] and SHA-1 [26]. However, before proceeding any implementation task, it is necessary to present the secure hash algorithm characteristics as mentioned in table I below.

TABLE I
Secure hash algorithms characteristics.

Algorithms	Word (w)	Message size	Block (m)	Digest	Digest rounds number
SHA-1	32	2^{64}	512	160	80
SHA-256	32	2^{64}	512	256	64
SHA-512	64	2^{128}	1024	512	80

The different steps followed to generate the digest message using SHA-256 hash algorithm are explained as follow:

SHA-256 operates in the same manner as MD5 and SHA-1. The length of input message is first padded in such away the result length is a multiple of 512 bits. Second, it is parsed into 512-bits blocks $M^{(1)}$, $M^{(2)}$...,$M^{(N)}$. The message blocks are computed sequentially one by one, starting from an initial hash value $H^{(0)}$ as given in equation 1

$$H^{(i)} = H^{(i-1)} + C_M(i)(H^{(i-1)}) \tag{1}$$

where C is the compression function, + means word-wise mod 2^{32} and $H^{(i)}$ is the hash of M.

Generally, the SHA-256 can be expressed in form of four functions. Indeed, the 'sha256_init' function initializes the eight 32bits variables $H^{(0)}$, $H^{(1)}$, $H^{(2)}$, $H^{(3)}$, $H^{(4)}$, $H^{(5)}$, $H^{(6)}$, $H^{(7)}$ for use with 'SHA256_Update' and 'SHA256_final'. The 'SHA256_final' is called when all data has been added via 'SHA256_Update' and loads a message digest. On the other hand, the 'SHA256_Transform' is used by 'SHA256_Update' and 'SHA256_final' to hash the 512-bit input blocks and build the core of the algorithm. Fig. 2 shows the pseudo codes description of the SHA-256 functions.

1) Preprocessing(Overview): As prior hash algorithms, SHA-256 is computed as follow: the hash message is first padded so that the final length (L) will be a multiple of 512 bits [27]. Then, a single 1-bit is append in the end of the message followed by *K* zero bit, where k refers to the smallest positive solution to the equation *L+K+1*=448 mod 512. A 64-bit representation of *L* is added to the result of the padding. For instance, taking an example of a message (8-bit ASCII) *"abc"* which has the length equal to 8×3=24. This latter is padded to a 1, then (448-(24+1))=423 zero bits and finally to its length to get the 512-bit binary message as presented in Fig. 3 [28].

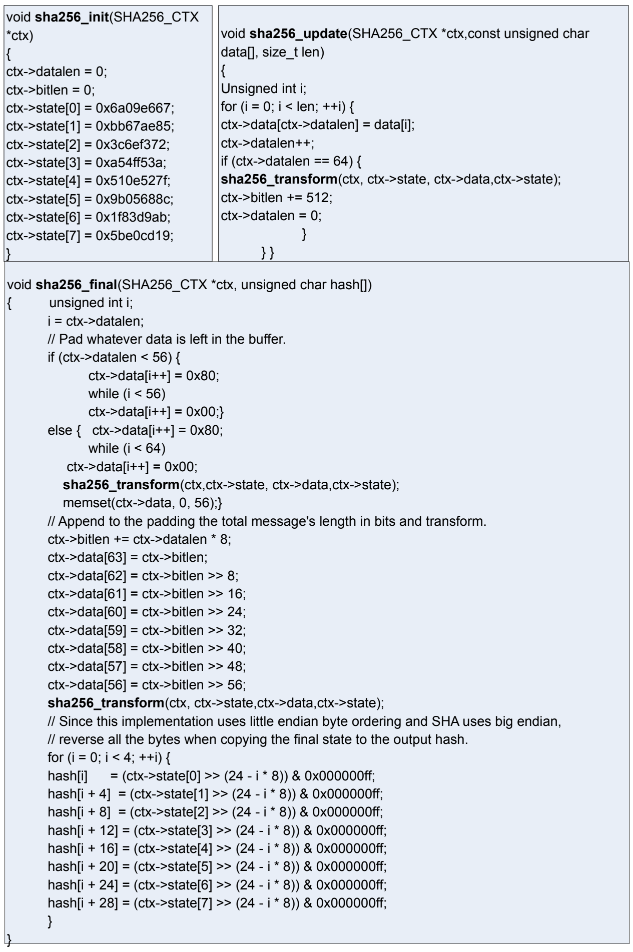
This message is parsed into individual N=512 bit blocks $M^{(1)}$, $M^{(2)}$,..,$M^{(N)}$ and then passed to the message expander.

2) Hash Operation(Overview): A set of logical functions are used in the SHA-256 algorithm and operate on 32-bit words [29]. These functions are illustrated in equations 2, 3, 4, 5, 6 and 7

$$Ch(x,y,z) = (x \wedge y) \oplus (\ x \wedge z) \tag{2}$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \tag{3}$$

$$\sum_{0}(x) = S^{2}(x) \oplus S^{13}(x) \oplus S^{22}(x) \tag{4}$$



```
void sha256_init(SHA256_CTX
*ctx)
{
ctx->datalen = 0;
ctx->bitlen = 0;
ctx->state[0] = 0x6a09e667;
ctx->state[1] = 0xbb67ae85;
ctx->state[2] = 0x3c6ef372;
ctx->state[3] = 0xa54ff53a;
ctx->state[4] = 0x510e527f;
ctx->state[5] = 0x9b05688c;
ctx->state[6] = 0x1f83d9ab;
ctx->state[7] = 0x5be0cd19;
}
```

```
void sha256_update(SHA256_CTX *ctx,const unsigned char
data[], size_t len)
{
Unsigned int i;
for (i = 0; i < len; ++i) {
ctx->data[ctx->datalen] = data[i];
ctx->datalen++;
if (ctx->datalen == 64) {
sha256_transform(ctx, ctx->state, ctx->data,ctx->state);
ctx->bitlen += 512;
ctx->datalen = 0;
                    }
            } }
```

```
void sha256_final(SHA256_CTX *ctx, unsigned char hash[])
{        unsigned int i;
         i = ctx->datalen;
         // Pad whatever data is left in the buffer.
         if (ctx->datalen < 56) {
                  ctx->data[i++] = 0x80;
                  while (i < 56)
                  ctx->data[i++] = 0x00;}
         else {   ctx->data[i++] = 0x80;
                  while (i < 64)
              ctx->data[i++] = 0x00;
           sha256_transform(ctx,ctx->state, ctx->data,ctx->state);
           memset(ctx->data, 0, 56);}
         // Append to the padding the total message's length in bits and transform.
         ctx->bitlen += ctx->datalen * 8;
         ctx->data[63] = ctx->bitlen;
         ctx->data[62] = ctx->bitlen >> 8;
         ctx->data[61] = ctx->bitlen >> 16;
         ctx->data[60] = ctx->bitlen >> 24;
         ctx->data[59] = ctx->bitlen >> 32;
         ctx->data[58] = ctx->bitlen >> 40;
         ctx->data[57] = ctx->bitlen >> 48;
         ctx->data[56] = ctx->bitlen >> 56;
         sha256_transform(ctx, ctx->state,ctx->data,ctx->state);
         // Since this implementation uses little endian byte ordering and SHA uses big endian,
         // reverse all the bytes when copying the final state to the output hash.
         for (i = 0; i < 4; ++i) {
         hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
         hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
         }
}
```

Fig. 2. Code description of the SHA-256 functions.

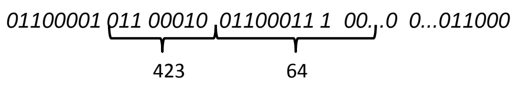


Fig. 3. Preprocessing of the SHA-256.

$$\sum_{1}(x) = S^{6}(x) \oplus S^{11}(x) \oplus S^{25}(x) \tag{5}$$

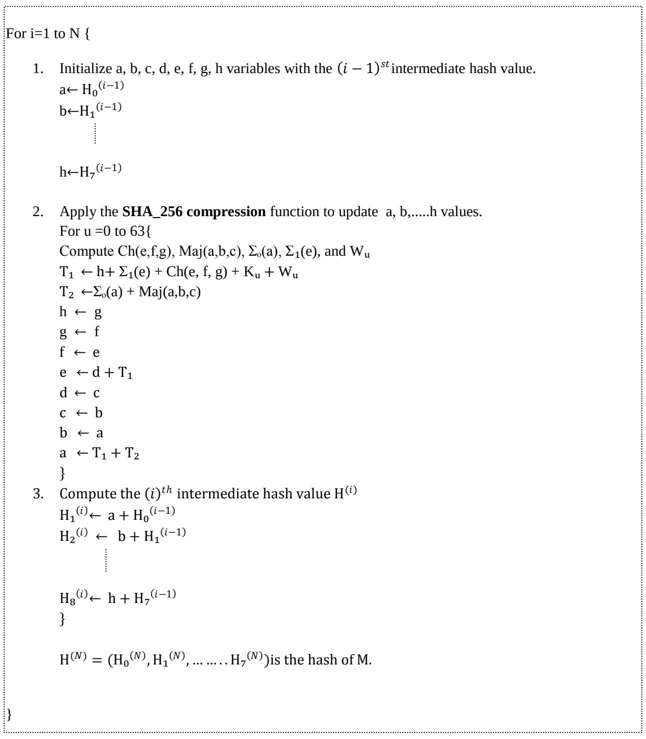
$$\delta_0 = S^{7}(x) \oplus S^{18}(x) \oplus R^{3}(x) \tag{6}$$

$$\delta_1 = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x) \tag{7}$$

where $\oplus$, $\wedge$ and $\sim$ are respectively the bitwise XOR, the bitwise AND and NOT while R and S represent the right shift and right rotation by *n* bits.

Fig. 4 illustrates the different steps proceeded in order to hash a message M which contains N blocks: Where W_u is the expanded message blocks (W_0 , W_1 , W_{63}) which are computed as given in equations 8 and 9, while K_u is a sequence of 64 constants used to initialize hash values:

$$W_u = M_u; (u = 0..15) \tag{8}$$



```
For i=1 to N {
    1. Initialize a, b, c, d, e, f, g, h variables with the $(i-1)^{st}$ intermediate hash value.
       a← $H_0^{(i-1)}$
       b←$H_1^{(i-1)}$
          ⋮
       h←$H_7^{(i-1)}$

    2. Apply the SHA_256 compression function to update a, b,.....h values.
       For u =0 to 63{
       Compute Ch(e,f,g), Maj(a,b,c), $\Sigma_0$(a), $\Sigma_1$(e), and $W_u$
       $T_1 \leftarrow h + \Sigma_1(e) + Ch(e, f, g) + K_u + W_u$
       $T_2 \leftarrow \Sigma_0(a) + Maj(a,b,c)$
       h ← g
       g ← f
       f ← e
       e ← $d + T_1$
       d ← c
       c ← b
       b ← a
       a ← $T_1 + T_2$
       }
    3. Compute the $(i)^{th}$ intermediate hash value $H^{(i)}$
       $H_1^{(i)} \leftarrow a + H_0^{(i-1)}$
       $H_2^{(i)} \leftarrow b + H_1^{(i-1)}$
          ⋮
       $H_8^{(i)} \leftarrow h + H_7^{(i-1)}$
       }

       $H^{(N)} = (H_0^{(N)}, H_1^{(N)}, \ldots\ldots H_7^{(N)})$is the hash of M.
}
```

Fig. 4. The steps proceeded to hash a message.

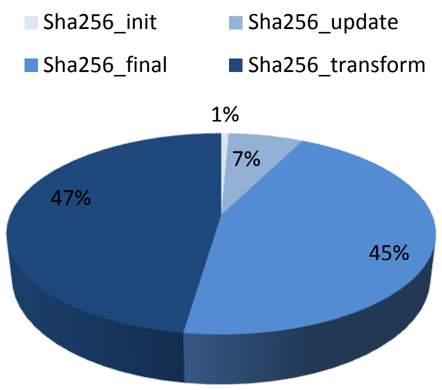


Fig. 5. Time distribution in (%) on ARM Cortex A9 processor for SHA-256 functions.

$$W_u = \delta_1(W_{u-2}) + W_{u-7} + \delta_0(W_{u-15}) + W_{u-16}; (u = 16..63) \tag{9}$$

3) Complexity Analyzes: In order to identify which part of SHA-256 algorithm is most consuming, a profiling is carried on SHA-256 software code using 'xtime 1.h' library under Standalone hosted on the ARM Cortex A9 processor operating at 667 MHz. Fig. 5 reports the time distribution in percentage for SHA-256 functions.

According to results supplied in Fig. 5, it is evident that the 'SHA256_Transform' function is the most complex and

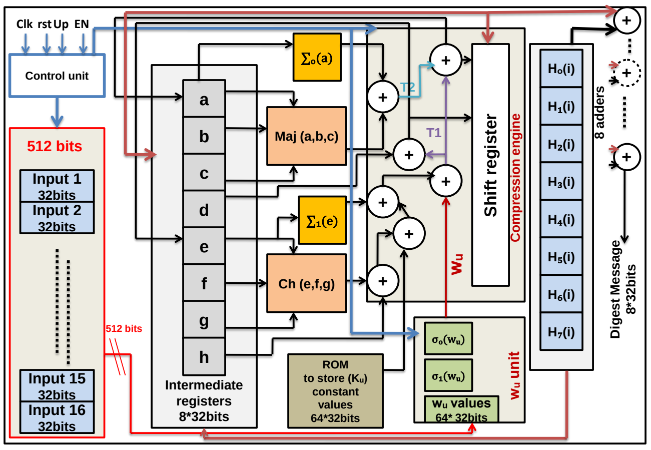


Fig. 6. The block diagram of the proposed architecture.

consumes about 47% of the total execution time required for SHA-256 algorithm. Thus, it is enough to design efficient hardware architecture for 'SHA256_Transform' function in order to ensure a trade-off between flexibility and performances.

B. Hardware exploration of 'SHA256_Transform' block

As the 'SHA256_Transform' function is the most time consuming function in the SHA-256 algorithm, we present in this section two hardware implementations based on low level architecture and high level architecture in order to find the optimal solution in term of throughput and power consumption.

1) Low level proposed architecture: The block diagram of the proposed low level architecture is detailed in Fig. 6. The proposed hardware architecture is dedicated to support 'SHA256_Transform' function.

This design supports a set of components designed as follow:

- *Input/output registers:* A 512 bits register per block which is organized in form of 16 inputs coded within 32 bits. Then, the 8×32 bits digest message generated in the output constitutes the hash value which is the concatenation of ($H_0^{(i)}$, $H_1^{(i)}$,........$H_7^{(i)}$).
- *The compression engine:* this unit is responsible for computing intermediate hash values *(a, b,...g, h)*. Given the expanded message *Wu*, the constant values *Ku* and the 32-bits initialized registers (a to h), compute T1 and T2 values used to update A and E registers. Afterwards, the shift registers is used to update the remaining registers in each clock cycle. To accomplish this task, we exploit the pre-calculated functions $\sum_0(a)$, $\sum_1(e)$, *Maj(a,b,c)* and *Ch(e,f,g)*.
- W_u *unit:* presented in Fig. 7, generates the W_u used by the round computation. For the first 16 rounds (W_0 to W_{15}), they are transmitted to the compression engine as *input1, input2,...., input16* to provide the first values of W_u. After that, W_u is computed recursively using its previous values W_{u-2}, W_{u-7}, W_{u-15} and W_{u-16}. This calculation requires the estimation of $\delta_0(W_u)$ and $\delta_1(W_u)$ values.

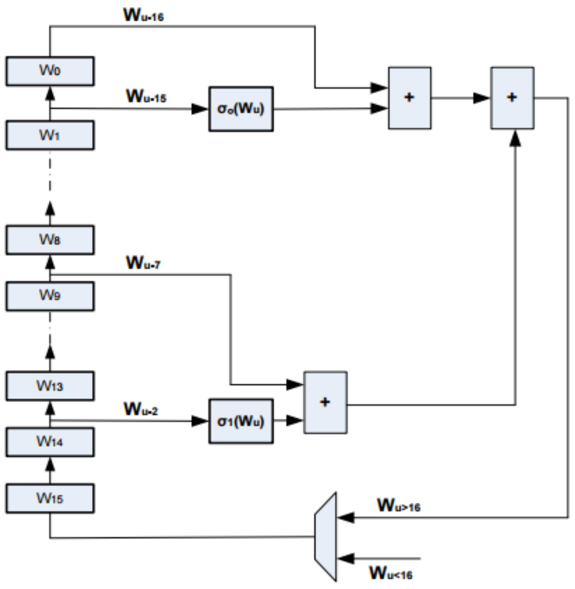


Fig. 7. Implementation of W_u unit.

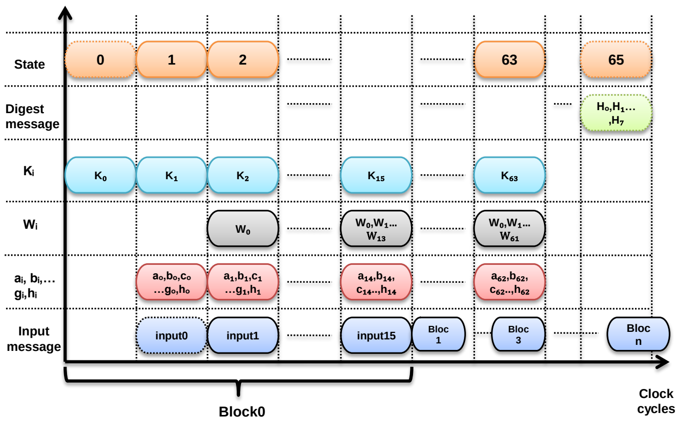


Fig. 8. The total clock cycles required for the hash operation.

- *ROM memory:* It is used to store 64 K_u constant values where the total size of the ROM is about 64×32 bits.
- *The control unit:* The whole operation of the control unit is explained in Fig. 8. In fact, the 16 words representing the 512 bits input message require 16 clock cycles to be transferred to the W_u unit. Simultaneously, the calculation of the intermediate registers *(a, b, c...h)* are carried on which values are updated in each clock cycle. Furthermore, the pipeline process is applied between the compression engine that is responsible for loading *(a, b, c ... and h)* values and the W_u computing unit. At all, 65 clock cycles are enough to load the digest message which is the concatenation of $H_0 \parallel H_1 \parallel H_2 \parallel H_3 \parallel H_4 \parallel H_5 \parallel H_6 \parallel H_7$.

The maximum throughput achieved by the proposed design is computed as given in equation 10

$$\delta = MB \times freq/C \tag{10}$$

where δ is the maximum throughput, *MB* is the message block

TABLE II
COMPARISON STUDY WITH PREVIOUS WORKS.

Designs	FPGA	Area (Slice)	Freq. (MHZ)	Throughputs (Mbps)
Ignacio et al. [13]	Virtex-2 XC2VP-7	1274	115.46	909.48
Mihai et al. [14]	Virtex 5 XC5VFX100T	-	125	420
Sang − H. et al. [15]	Virtex-5	1080	129	826
Ryan et al. [16]	Virtex 200/400XCV	1306	77	308
Nicolas et al. [17]	Virtex v200pq240	2384 (CLBs)	74	291
Rommel et al. [18]	Virtex-4 LX Virtex-5 VLX	422 139	50.06 64.45	91 117.8
Harris et al. [19]	Virtex -5 XC5VLX330	9012	112	3880
Proposed solution	XC7Z020	1305	135	1063

size, *freq* is the operating frequency and *C* is the total clock cycles.

Table II summarizes the FPGA resources, the operating frequency and the throughput results of the proposed architecture which was implemented in XC7Z020 Zynq [30] device and simulated using ModelSim tool.

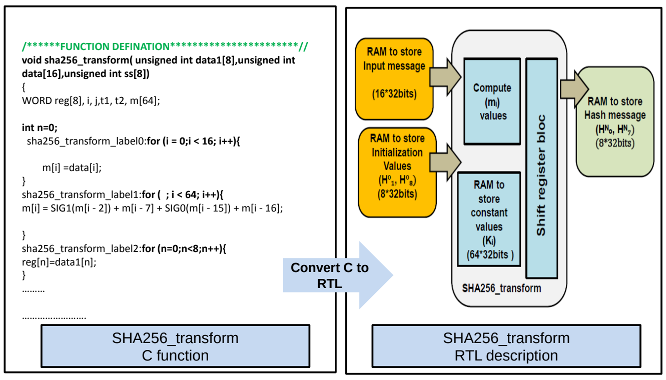
The synthesis results show that our design offers a throughput factor 14% to 88% better than [13], [14], [15], [16], [17] and [18]. Further, the area cost of our design presents an enhancement of about 85% relative to [19].

The next section will be devoted to develop the high level proposed architecture. The main objective of this study is to determine which design method among LLS and HLS provides better performances in term of area cost, throughput and power consumption.

2) High Level Proposed Architecture: In this section, the HLS synthesis is adopted as a design method in order to improve design performances in term of area cost and throughput factor. At this stage, the Vivado HLS tool is exploited as a tool to develop several optimized hardware solutions for 'SHA256_Transform' function using Zynq 7000 FPGA platform. In the top level function, we use two 32 bits input vectors which are data1[8] and data[16] to store respectively the first eight initial values ($H_0^{(0)}$, $H_1^{(0)}$, $H_2^{(0)}$, $H_3^{(0)}$, $H_4^{(0)}$, $H_5^{(0)}$, $H_6^{(0)}$, $H_7^{(0)}$) and the 16 words representing the 512 bits input message. In the output, the hash message is loaded into $8{\times}32$ bits RAM memory block. The block diagram of the proposed HLS architecture is illustrated in Fig. 9.

To improve design productivities, two hardware solutions are elaborated by adding optimized pragmas incrementally to the design.

- *#Solution 1:* In this first experiment, the SW code of 'SHA256_Transform' function is kept in initial condition without adding any optimization. Besides, the hardware design is synthesized under Zynq 7000 FPGA using Vivado HLS tool. Indeed, the experimental results supplied in table III, prove that FPGA implementation is shared between 339 SLICE, 1036 LUT, 5 BRAM and 1322 FF with an operating frequency equal to 222 MHz. Otherwise, the maximum throughput achieved by this solution can reach 96 Mbps.
- *#Optimized solution:* In the second experiment, the UNROLL directive is applied to the external loops of the computation and scheduling equations. This optimization leads to reduce latency time and improve maximum throughput achieved by the hardware design. Although this technique has reduced the Latency with a profit up to 87% compared to #Solution 1, it comes with a penalty in area cost. On the other hand, we conclude, from Table III, that the manual solution possesses 15% to 90% higher throughput compared to #Optimized solution and #Solution 1 respectively and also it uses 80% fewer area resources relative to #Optimized solution.



```
/******FUNCTION DEFINATION**********************//
void sha256_transform( unsigned int data1[8],unsigned int
data[16],unsigned int ss[8])
{
WORD reg[8], i, j,t1, t2, m[64];

int n=0;
  sha256_transform_label0:for (i = 0;i < 16; i++){

      m[i] =data[i];
}
sha256_transform_label1:for (  ; i < 64; i++){
m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

}
sha256_transform_label2:for (n=0;n<8;n++){
reg[n]=data1[n];
}
.........

.........................
```

SHA256_transform
C function
Convert C to RTL
RAM to store Input message (16*32bits)
RAM to store Initialization Values (H^0_1, H^0_8) (8*32bits)
Compute (m_i) values
RAM to store constant values (K_i) (64*32bits)
Shift register bloc
SHA256_transform
RAM to store Hash message (H^N_0, H^N_7) (8*32bits)
SHA256_transform
RTL description

Fig. 9. The HLS proposed architecture.

On the other hand, we evaluate the power consumption of the different proposed solutions. From table III, it is evident that our proposed solution consumes less power than #Solution 1 and #Optimized solution. This is explained by the fact that the operating frequency of the manual approach is fewer than the HLS solutions. Consequently, it can be noticed that the proposed manual architecture is more efficient than HLS cases as it provides the best trade-off between FPGA resources and throughput.

After achieving all hardware acceleration tasks, section 4 will be reserved for the study of the developed solutions in such an SW/HW environment. In fact, a deeply evaluation of design performances in terms of FPGA resources, execution time, and power consumption will be detailed.

IV. SW/HW SOLUTIONS EXPLORATION AND PERFORMANCE EVALUATION

To highlight the influence of hardware acceleration in terms of time, area cost and power consumption, the SW/HW concept is adopted as a design method. This solution exploits the Zynq SoC architecture which incorporates an ARM Cortex A9 [31] processor operating at 667 MHz and a programmable logic (PL). The communication between the processor and PL is ensured through an AXI4-Stream protocol controlled by the TDATA, TVALID, TLAST and TREADY signals [32]. In case of HLS design method, the streaming interface is developed based on #Optimized solution which provides

TABLE III
IMPLEMENTATION RESULTS OF THE HLS SOLUTIONS UNDER FPGA PLATFORM.

Designs	FPGA	Area (Slice)	BRAM	FF	Freq. (MHz)	Through. (Mbps)	Power (Watt)
#Solution1	XC7Z020	339	5	1322	222	96	0.052
#Optimized solution	XC7Z020	6367	0	25190	181	917	0.041
#Proposed solution	XC7Z020	1305	-	2583	135	1063	0.022

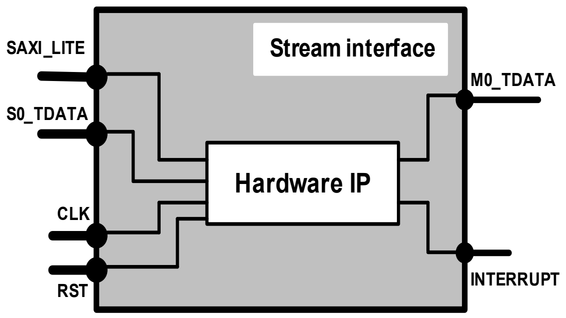


Fig. 10. Connection between hardware IP and AX4-Stream interface.

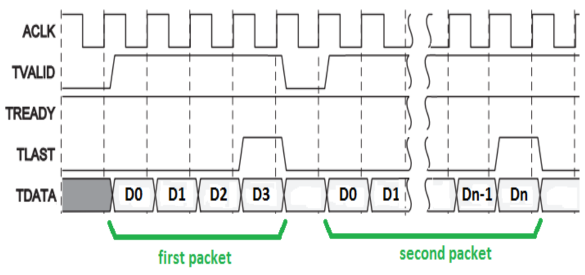


Fig. 11. The AXI4-stream behavior.

the best trade-off between throughput and area cost. The different signals used to control the transfer status between the processor and the stream interfaces are detailed in Fig. 10. The AXI4-stream uses between 2 and 9 signals to ensure the communication between master and the slave protocols. The main used signals are TVALID which indicates the presence of data, the TREADY flag is equal to 1 when it is ready to receive the data and the TLAST signal notifies the end of the frame. This behavior is explained in Fig. 11. On the other hand, the work presented in [33] proved that the AXI4-Lite bus performances are limited since it provides a sequential transfer of only 32-bit data. This generates a huge communication time and makes the transfer a bit slow. However, In case of the streaming interface, it is just enough to fix the maximum packet size depending on input message. Afterwards, the 8 bits input data are concatenated within 32 bits registers and then communicated in form of train to the DDR memory.

A synthesis results of the proposed hardware IPs with HLS (#SHA256_Transform_HLS) and LLS (#SHA256_Transform_LLS) design methods connected

TABLE IV
IMPLEMENTATION RESULTS OF THE PROPOSED STREAMING INTERFACES.

Designs	FPGA	LUT	BRAM	FF	Freq. (MHz)
$\#SHA$256 _Transform_HLS	XC7Z020	21197	2	19212	175
$\#SHA$256 _Transform_LLS	XC7Z020	3358	0	3400	135

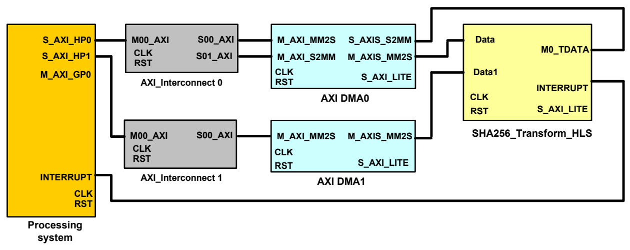


Fig. 12. The block diagram of the SW/HW LLS design.

to the streaming interface is presented in table IV.

From this implementation results, we confirm that the #SHA256_Transform _LLS solution provides a gain of about 84% in LUT compared to #SHA256 _Transform_HLS solution while the operating frequency of the #SHA256_Trans form_HLS is 22% better than the #SHA256_Transform_LLS.

Furthermore, the full SW/HW designs of the SHA-256 algorithm are carried out in Standalone execution mode using the Xilinx ZC702 board. Indeed, the SW/HW LLS design, given in Fig. 12, includes one DMA channel configured in read/write mode connected to #SHA256_Transform_LLS IP.

This routing provides a direct access to the DDR external memory in order to read and write data. However, since the #SHA256_Transform_HLS stream interface requires two input vectors, the SW/HW HLS design uses two DMA channels to connect the #SHA256_Transform_HLS stream interface. The first DMA is configured in read/write mode while the second is operating in write mode only as detailed in Fig. 13 below.

The next stage of this project consists of comparing the different proposed design methods (SW, SW/HW HLS, SW/HW LLS) in term of time and power consumption. As a first experiment, we study the impact of only 'SHA256_transform' accelerators on execution time without including the whole SHA-256 chain. Then, the experimental results are compared to the SW implementation as presented in Fig. 14. The general formula followed to evaluate the execution time "DTime" in microsecond is provided in equation 11:

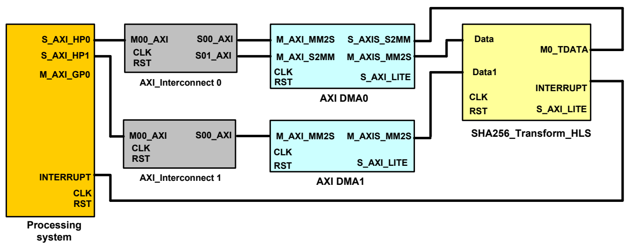


Fig. 13. The block diagram of the SW/HW HLS design.

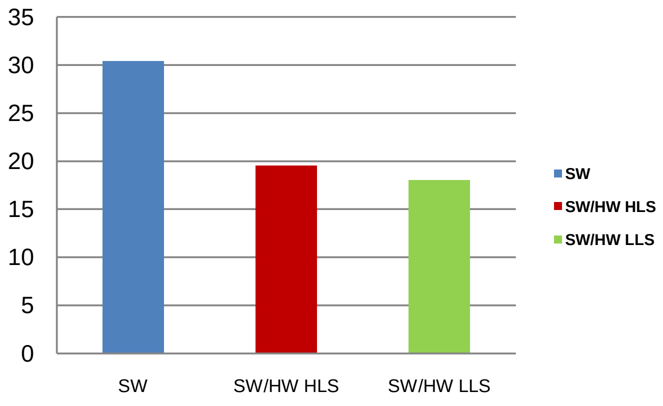


Fig. 14. Execution time of the SHA256_Transform implementations.

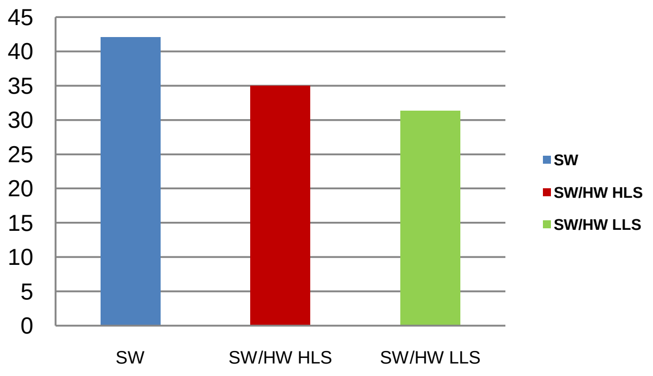


Fig. 15. Execution time results of the SHA-256 algorithm.

$$DTime = 1.0 \times (End - Start)/(CNT/1,000,000) \quad (11)$$

where *end* is the end time, *start* is the start time, and *CNT = 100,000,000/2*.

As shown in Fig. 14, the execution time for the developed SW/HW LLS and SW/HW HLS solutions is reduced by nearly 43% and 35% respectively compared to SW case.

In the second experiment, we evaluate the execution time of the whole SHA-256 chain after adding the 'SHA256_Transform' accelerators designed with LLS et HLS methods as illustrated in Fig. 15.

The experimental results prove that the execution time of the SW/HW LLS solution is 10% and 25% better than SW/HW HLS and SW designs respectively.

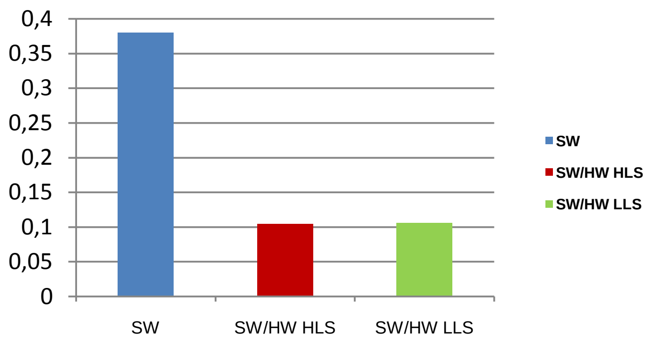


Fig. 16. Power consumption comparison.

To evaluate the performance, we study of the influence of the proposed design methods on the power consumption constraint as detailed in Fig. 16. In fact, the Xilinx fusion digital power designer is adopted as a tool to measure accurately the consumed power of the different proposed solutions after connecting a USB interface adapter to the board.

From Fig. 16, the calculated results of the SW/HW solutions seem to be equal. In addition to that, they allow a gain up to 73% relative to SW implementation.

Regarding all obtained results including the FPGA area, the throughput factor, the execution time and the power consumption, we confirmed that the SW/HW LLS solution managed to report an efficient acceleration for the SHA-256 hash algorithm while maintaining low power consumption.

V. Conclusion

Hash functions are central to compute signatures and MACs that allow users to verify authenticity and integrity of data. In this paper, we focused on the implementation of the SHA-256 hash algorithm under ZC 702 platform. After profiling the SHA-256 software, the 'Sha256_Transform' function seemed to be the critical unit with 47% of the total execution time.

Two solutions were introduced for the algorithm acceleration, the first one by using the LLS design method and the second one by exploiting the HLS technique. The analysis of the hardware implementations showed that the LLS solution possesses a 15% higher throughput and up to 80% less area cost rather than the HLS case. In addition, the developed hardware solutions are incorporated in SW/HW environment using the ARM cortex A9 processor operating at 667 MHz and an AXI4-Stream connected to a Direct Memory Access channel. Then, the SW/HW HLS design is compared to the SW and the SW/HW LLS solutions in term of time and power consumption. The experimental results proved that the SW/HW LLS design provided the best compromise Area/ Performance while maintaining a low power consumption ratio relative to SW/HW HLS and SW cases. Indeed, the execution time is reduced by 25% with 73% lower power consumption compared to the SW case which makes it a suitable choice for hardware acceleration. Consequently, we conclude that the hardware acceleration is able to produce a remarkable gain in term of energy and time response. This contribution will underline the importance of such co-design optimizations in order to improve the embedded system architecture across the traditional boundaries of hardware and software. Also, it lets the developers to think about design in terms of a trade-off between performance and flexibility.

Although the amelioration performed by our work, the results still average, which push us to think of better way to optimize the accelerator to cost less resource. As future works, we will migrate to a more sophisticated processor from RISC-V to have better results. In addition to that, we will study other recent hash algorithms using HW/SW design method in order to estimate implementation constraints and compare results with the SHA-256 proposed solution. Furthermore, we can also exploit reconfigurable methods as a solution to avoid memory overhead and decrease power consumption.

References

[1] A. Ben Atitallah,M. Kammoun, An FPGA comparative study of high-level and low-level combined designs for HEVC intra, inverse quantization, and IDCT/IDST 2D modules, international journal of circuit theory and applications, 1-17,2020, https://doi.org/10.1002/cta.2790.

[2] S. Lahti, P. Sjvall, Are we there yet? A study on the state of high-level synthesis, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5):898-911, 2019, DOI: 10.1109/TCAD.2018.2834439.

[3] M. Pelcat, C. Bourrasset, Design productivity of a high-level synthesis compiler versus HDL, IEEE SAMOS'16, 17-21, 2016, DOI : 10.1109/SAMOS.2016.7818341.

[4] P. Sjvall, High-level synthesis of HEVC intra prediction on FPGA, Master of Science Thesis, 2015.

[5] X.S. Zhang, Tutorial for Vivado HLS. Washington University, St. Louis, Missouri, https: //www.ese.wustl.edu/ xuan.zhang/ ese566files/ tutorials/.pdf, 2007.

[6] CALYPTO, Catapult C Synthesis. http://www.calypto.com/catapult-c-synthesis.php.

[7] M. Elleuchi , M. Boujelben, Towards low power security mechanisms and architectures for Wireless Sensor Networks. Journal of Information Assurance and Security, 13: 066-079, 2018.

[8] T. Dipti, and M. Utsav Kumar, Low Power Implementation of Secure Hashing Algorithm (SHA-2) using VHDL on FPGA of SHA-256, International Journal for Research in Applied Science & Engineering Technology, 13(5), (2018), DOI: 10.22214/ijraset.2018.5376.

[9] National Institute of Standards and Technology, Secure Hash Standard. Federal information Processing Standards, 180-4, 2012.

[10] Z. Xiaoyong , W. Ruizhen . A High-Performance Parallel Computation Hardware Architecture in ASIC of SHA-256 Hash. International Conference on Advanced Communications Technology(ICACT), 17-20, 2019, DOI: 10.23919/ICACT48636.2020.9061457.

[11] K.T. Kurt, C.L.Y. Steve, An FPGA based SHA-256 processor. International Conference on Field Programmable Logic and Applications, 577-585, 2012, DOI: https://doi.org/10.1007/3-540-46117-5_60.

[12] Xilinx, Inc. ZC-702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC. (Available from: http://www.xilinx.com/support/documentation/boards and kits/zc702 zvik/ug850-zc702-eval-bd.pdf).

[13] I. Algredo-Badilloa, C. Feregrino-Uribe, FPGA-based implementation alternatives for the inner loop of the Secure Hash Algorithm SHA-256, Microprocessors and Microsystems, 37: 750-757, 2013, https://doi.org/10.1016/j.micpro.2012.06.007.

[14] T. Mihai, F. Adrian, Design and implementation of cryptographic modules on FPGAs. Proceedings of the Applied Mathematics and Informatics, 2010, 149154, 2010.

[15] L. Sang-Hyun, S. Kyung-Wook, An Efficient Implementation of SHA processor including three Hash Algorithms (SHA-512, SHA-512/224, SHA-512/256). International Conference on Electronics, Information, and Communication, 24-27, 2018, DOI: 10.23919/ELINFOCOM.2018.8330578.

[16] G. Ryan, I. Laurent, Multi-mode operator for SHA-2 hash functions. Journal of Systems Architecture, 53: 127-138, 2007, https://doi.org/10.1016/j.sysarc.2006.09.006.

[17] N. Sklavos, and O. Koufopavlou, Implementation of the SHA-2 Hash Family Standard Using FPGAs. The Journal of Supercomputing, 31: 227-248, 2005, https://doi.org/10.1007/s11227-005-0086-5.

[18] G. Rommel, , A. Ignacio, A compact FPGA-based processor for the Secure Hash Algorithm SHA-256, Computers and Electrical Engineering, 40: 194-202, 2014, https://doi.org/10.1016/j.compeleceng.2013.11.014.

[19] E.M. Harris , K. Apostolis, Hardware Implementation of the Totally Self-Checking SHA-256 Hash Core, International Conference on Computer as a Tool (EUROCON), 2015, DOI: 10.1109/EUROCON.2015.7313715.

[20] I. Gordon, F. Shane, Is high level synthesis ready for business a computational finance case study, in Int. Conf. Field-Programmable Technol. (FPT), 2014, DOI: 10.1109/FPT.2014.7082747.

[21] M. Kammoun, , B.A. Ahmed, Case study of an HEVC decoder application using high-level synthesis: intra prediction, dequantization, and inverse transform blocks, Journal of Electronic Imaging, 28, 2019, DOI: 10.1117/1.JEI.28.3.033010.

[22] A. Karim, B.A. Rabie, Exploring HLS optimizations for efficient stereo matching hardware implementation. in Int. Symp. Appl. Reconfigurable Comput., 168-176, 2017, DOI: 10.1007/978-3-319-56258-2.

[23] H S. Jacinto, L. Daoud , High Level Synthesis Using Vivado HLS for Optimizations of SHA-3. 60th International Midwest Symposium on Circuits and Systems (MWS-CAS), 6-9, 2017, DOI: 10.1109/MWS-CAS.2017.8052985.

[24] A. Imtiaz, A. Shoba Das, Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs, Computers and Electrical Engineering, 31: 345-360, 2005, https://doi.org/ 10.1016/ j.compeleceng. 2005.07.001.

[25] A. Radwa, M.M. Fouad, Design and implementation of new security hash algorithm based on MD5 and SHA-256. International Journal of Engineering Sciences &Emerging Technologies, 6: 29-36, 2013.

[26] W. Xiaoyun , L Y. Yiqun, Finding collisions in the full SHA-1, in: V. Shoup (Ed.), Advances in Cryptology CRYPTO'05, 3621: 17-36, 2005, DOI https://doi.org/10.1007/11535218_2.

[27] IWS, Descriptions of SHA-256, SHA-384, and SHA-512. http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf.

[28] K. Fatma , M. Hassen, Efficient FPGA hardware implementation of secure hash function SHA-256/Blacke-256, 12th International Multi-Conference on Systems, Signals & Devices (SSD15), 2015, DOI: 10.1109/SSD.2015.7348105.

[29] D. Rachmawati, J T. Tarigan, A comparative study of Message Digest 5(MD5) and SHA256 algorithm. 2nd International Conference on Computing and Applied Informatics, 2017.

[30] Xilinx Inc, Zynq-7000 all programmable SoC software developers guide, Version 9.0, http://www.xilinx.com/support/documentation/boardsand kits/zc702zvik/ug850-zc702-evalbd.pdf, 2014.

[31] ARM Inc, AMBA AXI and ACE protocol specification. http://infocenter.arm.com/ help/ index.jsp?topic=/ com.arm.doc.ihi0022e/ index.html, 2013.

[32] XILINX AXI reference guide , UG761 (v13.4), https://www.xilinx.com/ support/ documentation/ ip documentation/ axi ref guide/ v13 4/ ug761 axi reference guide.pdf, 2012.

[33] M. Kammoun, A. Ben Atitallah, Design exploration of effcient implementation on SoC heterogeneous platform. International Journal of Circuit Theory and Applications, 45: 2243-2259, 2016, DOI : 10.1002/cta.2308.

Manel Kammoun received her PhD in electronics from the National School of Engineering-Sfax (ENIS) in 2018 and her diploma of engineer in electronics from the University of Sfax in 2012. Currently, she is researcher in the Laboratory of Electronics and Information Technology within the C&S (Circuit & System) team. Her main research activities are focused on image and video signal processing, hardware implementation, and embedded systems.

Manel Elleuchi was born in 1985. Currently, she is a Doctor at the National Engineering School of Sfax. Her research activity is conducted within CES research unit. She has received a Ph.D. in computer systems engineering in December 2017. She has received the Engineering degree, from the National School of Engineers of Sfax (ENIS), Tunis, Tunisia in 2009 and the Master degree in New Technologies of Dedicated Computer Systems, from the National Engineering School of Sfax, in 2012. She has authored more than 10 papers. Her current research interests are in the field of Wireless Sensor Networks (WSNs) and the Internet of things (IoT). She focused on the security and routing protocols in WSNs and IoT in low power consumption.

Mohamed Abid is head of Computer Embedded System laboratory CES-ENIS, Tunisia. He is working now as a Professor at the Engineering National School of Sfax (ENIS), University of Sfax, Tunisia He received the Ph. D. degree from the National Institute of Applied Sciences, Toulouse (France) in 1989 in the area of Computer Engineering & Microelectronics. His current research interests include: hardware-software codesign, System on Chip, Reconfigurable System, and Embedded System, etc. He has also been investigating the design and implementation issues of FPGA embedded systems. Dr. Abid served in national or international conference organization and program committees at different organizational levels including Conference CoGeneral Chair, Technical program co-chair, organization co-chair and Member of several national and international conference Program Committees. He was Founding Member of several international conferences and school: SCS, SSD, GEI, and Sensor Net School. Recently, he is Vice General Co-chair of IDT'10. He was General Co-chair of SensorNetSchool'09, Vice General Co-chair of IDT'09, and Special Session Co-chair of ICECS'09. He is Member of technical committee of DASIP since 2007, ICM 2010 since 2006, ComNet 2010, and General Co-chair of IDT'08. He was also Joint Editor of Specific Issues in two International Journals and Joint editor of many conference's articles nationals and internationals: ICM'2004, GEI'2006-07, SCS'2004. He is a co-editor of the best paper in the international conference EDAC-ETC-EuroASIC'96.

Abdulfattah M. Obeid is deputy Director for Scientific Affairs, National Center for Electronics, Communications and Photonics at KACST. Founded and leading the Advanced Microsensors Division of ECP that is charged with stimulating the development of microelectronic development in the Kingdom of Saudi Arabia through provision of low cost manufacturing services in the region. Formed long-term strategic partnerships with a number of international private and public organizations and institutions for the implementation of various projects of national importance. He was a key member (Coordinator of the Electronics, Communications and Photonics strategic technology) of the implementation committee that drafted and the National Science, Technology and Innovation plan for Saudi Arabia. His research interests are Reconfigurable computing, VLSI architectures for DSP, Computer architecture, Wireless sensor networks and MEMS.