ISSN: 2182-2069 / E-ISSN: 2182-2077

Hardware Design and Implementation of Secure Hash Algorithm Based on FPGA

M.F. Al-Gailani[1*]

[1*]Cybersecurity Engineering Department, College of Information Engineering, Al-Nahrain University, Baghdad, Iraq. m.falih@nahrainuniv.edu.iq, https://orcid.org/0000-0003-4307-941X

Received: January 24, 2025; Revised: March 04, 2025; Accepted: April 21, 2025; Published: May 30, 2025

Abstract

The Secure Hash Algorithm is an essential part of data security. It has numerous applications, the most important of which is data integrity, which ensures that the received data matches the transmitted data and has not been tampered with during transmission or storage. Achieving data security often requires the use of a combination of algorithms and protocols to ensure data confidentiality, integrity, and authenticity. Therefore, a hardware implementation is the optimal solution to meet data security requirements and implement these algorithms in a coordinated and time-efficient manner, ensuring high execution speed compared to software implementation. In this research paper, hardware is designed for SHA-2 and implemented on a Xilinx XC7VX330T-3FFG1761 device using Xilinx ISE 14.7. The architecture is designed for message digest lengths of 256 and 512 bits. It is also suitable for other SHA-2 families by truncating the extra bits. The design aims to optimize performance with minimal complexity to suit resource-constrained devices and applications. Therefore, an iterative looping architecture was initially chosen to achieve this goal. Implementation results of the proposed architecture were satisfactory, with a throughput of 1.021 Gbps for SHA-256 and 1.331 Gbps for SHA-512. In addition, the number of slice registers was 333 for SHA-256 and 699 for SHA-512. The total power consumption for implementing the architecture was 0.394 W for SHA-256 and 0.608 W for SHA-512. A hybrid architecture for SHA-256 was then designed by replicating the hardware of a single round two and four times, allowing for the processing of two and four rounds per iteration. This reduces the number of iterations by half and a quarter, improving performance and increasing throughput by approximately 63% and 87% while increasing the number of slices LUTs and the number of occupied slices by only 5% and 30% (for unfolded 2), and 12% and 34% (for unfolded 4).

Keywords: FPGA, SHA, System Generator, Throughput.

1 Introduction

NIST, the National Institute of Standards and Technology, developed the Secure Hash Algorithm (SHA) for the first time in 1993. Since then, it has undergone several revisions to address identified vulnerabilities and add additional fixed-length output for message digests.

The first version, known as SHA-0 (NIST, 1993), was based on the MD4 hash function (Dobbertin, 1998; Rivest, 1991) published as the Federal Information Processing Standard (FIPS 180) in 1993. The next revised version, SHA-1 (NIST, 1995), was released as FIPS 180-1 in 1995. The second revision,

Journal of Internet Services and Information Security (JISIS), volume: 15, number: 2 (May), pp. 209-226.
DOI: *10.58346/JISIS.2025.I2.015*

*Corresponding author: Cybersecurity Engineering Department, College of Information Engineering, Al-Nahrain University, Baghdad, Iraq.

FIPS 180-2, released in 2002 and called SHA-2 (NIST, 2002), was based on the same underlying architecture and processes and produced only three different message digest lengths. The algorithms were named based on the length of the hash. Another hash was added in 2008 and published as FIPS 180-3 (NIST, 2008), and two more algorithms were added in 2015 and published as FIPS 180-4 (NIST, 2015). Table 1 shows a comparison between the different algorithms.

Each algorithm is a one-way recursive hash function that processes a variable-length message to generate a fixed-size hash value (digest), a condensed representation of it. These algorithms ensure message integrity since any modification to the message, even a single bit, will likely produce a distinct message digest (William Stallings, 2023). This feature is valuable for random numbers generation, digital signatures generation and verification, and message authentication codes generation (Rogaway & Shrimpton, 2004).

The NIST intended to gradually switch from SHA-1 to SHA-2 over a period of five years. However, the transition occurred in a shorter period as some researchers were able to describe an attack that could find a collision, where two distinct messages are hashed to an identical message digest (Barker & Roginsky, 2019; Polk et al., 2011; Stevens et al., 2017).

Table 1: Comparison of SHA Parameters (NIST, 2015)

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)	No. of Rounds	Year	FIPS
SHA-0	$< 2^{64}$	512	32	160	80	1993	180
SHA-1	$< 2^{64}$	512	32	160	80	1995	180-1
SHA-224	$< 2^{64}$	512	32	224	64	2008	180-3
SHA-256	$< 2^{64}$	512	32	256	64	2002	180-2
SHA-384	$< 2^{128}$	1024	64	384	80	2002	180-2
SHA-512	$< 2^{128}$	1024	64	512	80	2002	180-2
SHA-512/224	$< 2^{128}$	1024	64	224	80	2015	180-4
SHA-512/256	$< 2^{128}$	1024	64	256	80	2015	180-4

Field-programmable gate arrays (FPGAs) have recently gained widespread popularity in practical applications due to their advantages, including reconfigurability, design flexibility, performance, and power consumption (Orozco & Ttofis, 2025). However, deploying algorithms based on them faces several challenges, including:

- **Deployment:** Deploying algorithms on FPGAs in real-world applications presents hardware-software co-design issues, necessitating knowledge of HDLs/HLS to optimize performance, area efficiency, and power consumption. For instance, careful pipeline planning and parallelization are required to achieve maximum throughput while managing constrained logical resources.
- **Scalability:** Scalability is limited by the capacity of the FPGA, requiring multi-FPGA setups or dynamic partial reconfiguration for heavier workloads, while the extra expense of reconfiguration and synchronization makes the design more difficult.
- **Practical considerations:** Include robust debugging, maintenance strategies, security, and reliability.

The paper is organized as follows: Section 2 introduces the related work; Section 3 presents the methodology including a detailed description of the SHA algorithm, definition of the Field-Programmable Gate Array (FPGA) and system generator, and presentation of the proposed hardware design for the SHA algorithm; Section 4 discusses the implementation results; and the conclusion is presented in the last section.

2 Related Work

This section lists other research publications that address the practical part of the hardware implementation of the SHA.

In (Lee & Shin, 2018), the authors described the design and verification of an FPGA implementation of a SHA processor that performs three hash algorithms (SHA-512, SHA-512/224, and SHA-512/256). The design runs on a Virtex-5 FPGA device with a maximum clock frequency of 129 MHz and a throughput of 826 Mbps, occupying 1080 slices (Thomson. Yuvaraj et al., 2019) However, the authors designed the round operations with a 32-bit word length, unlike the NIST standard of 64-bit length, to reduce hardware complexity and gate count at the expense of the number of clock cycles and, consequently, throughput (Kumar & Nelakudit, 2021)

The authors in (Wong et al., 2018) proposed two solutions to improve the SHA-256's hardware implementation to achieve lightweight (low-cost area) and high-performance implementations. The first case is based on the Folded-5 architecture implemented on Virtex-6 (-3) and consumes 150 slices, with a maximum achieved frequency of 418 MHz, a clock cycles count of 321 per block, a throughput of 666.7 Mbps, and an efficiency ratio (TP/Slice) of 4.44. The second case uses the Folded-2 architecture with a 4-2 adder compressor implemented on Virtex-6 (-3). The required area is 197 slices, with a maximum frequency of 354 MHz, a clock cycle count of 129 per block, a throughput of 1.405 Gbps, and an efficiency ratio of 7.13. The optimization techniques followed by the authors were previously proposed by S. Dominikus (Dominikus, 2002) with the possibility of sharing resources between different algorithms implemented in the same system, and by Ling and Li in (Ling Bai & Shuguo Li, 2009) where they used a combination of 7-3 and 3-2 adder compressors instead of 4-2.

In 2020, the authors in (Kammoun et al., 2020) proposed two solutions for designing SHA-256 hardware. The first is a standard implementation without optimization. In contrast, the second optimizes the compression function by unrolling the external loops, improving the throughput by 87% compared to the first solution at the expense of hardware resources (Jain & Babu, 2024). The design was implemented on an XC7Z020 FPGA board, with a maximum achieved frequency of 181 MHz, resulting in a throughput of 917 MBit/s, consuming 6367 chips and 25190 FF. Although throughput improves compared to previous work, complexity has increased by a factor of five.

The research presented by (Gad et al., 2020) focused on applications that require low power consumption and area usage while delivering high efficiency and speed. Accordingly, they used various techniques to optimize their design, such as sharing of arithmetic resource, gated clock conversion, and structural modelling (Tang et al.,2024). Their design was implemented on a Virtex-7 FPGA, achieving a throughput of 0.637 Gbit/sec, a slices area of 275, and a power consumption of 13 mW. The authors believe that this achievement, by reducing complexity and area usage, will open up new avenues of applications.

In 2021, the authors in (Kieu-Do-Nguyen et al., 2021) implemented the SHA-256 computing core using hardware description languages. Their optimization technique incorporates a two-stage pipeline architecture to reduce the area usage and power consumption. They achieved a maximum frequency of 139.04 MHz and a throughput of 1.04 Gbps using a Xilinx Artix-7 (xc7a200t) device that consumed 1310 lookup tables, 881 registers, and 327 slices.

In 2024, researchers (Santos et al., 2024) improved the performance of the hardware implementation of SHA-256 using a multi-core architecture and parallel implementation. Their design allows a trade-off between low power consumption and high-performance applications by employing a single core up to

16 cores, respectively. With 16 cores, a maximum throughput of 1.4025 Gbit/sec is achieved, making their design suitable for IoT devices in blockchain environments. Although their design offers flexibility between power consumption and performance efficiency, there is a noticeable increase in resource consumption when choosing the highest core for maximum throughput from 1993 to 12618 slice.

By reviewing the related work and examining the methods used to improve performance and optimize resource utilization, the importance of the researcher's expertise in selecting the appropriate device, technology, and processes to achieve the research goal becomes clear. The results are generally good, with some observations and potential for improvement.

3 Methodology

This section comprehensively explains the secure hash algorithm for which the device is designed, followed by the hardware design of the algorithm's operations. The Xilinx System Generator is used in the design and implementation of the algorithm's hardware architecture on the target device, a Virtex-7 FPGA (XC7VX330T-3FFG1761).

3.1. Secure Hash Algorithm (SHA)

SHA consists of two stages: preprocessing and hash computation. Preprocessing is the process that prepares the message for the computation stage, which consists of three steps. The first step is to pad the message to make its length a multiple of the block length. The padding process is performed by appending a bit '1' to the message's end, followed by several '0' bits, computed according to Equation 1, adapted from (NIST, 2015), followed by r bits, which record the message's initial length before padding.

$$L + 1 + z = (b - r) \; mod \; b \tag{1}$$

Where L is the message's length prior to padding, z is the minimum number of added zeros that make the equation true, b is the block size, and its value is determined based on the algorithm, which can be 512 or 1024 bits, as illustrated in Table 1, and r is either 64-bit for a 512-bit block size or 128-bit for a 1024-bit block size of.

The second step is to parse the padded message into N b-bit blocks. The final step in the preprocessing stage is to initialize a 256-bit or 512-bit buffer (H_t), depending on the algorithm, to hold and update the intermediate and final results of the hash computation.

In general, these algorithms are constructed by generating multiple words, each of 32 bits in a 512-bit block algorithm or 64 bits in a 1024-bit block algorithm, from the fractional portions of the prime numbers' square roots, with each word generated from a different prime number. However, the last two algorithms in Table 1 are generated differently using an *IV Generation Function*.

In contrast, the hash computation process generates a message schedule by taking the N b-bit blocks of the padded message and the buffer and using functions, constants, and word operations, which is then used to recursively generate a sequence of hash values (the output from each processed message block), and the message digest is represented by the final hash value (NIST, 2015).

The core of the algorithm is the function F, an n-round module, as illustrated in Figure 1. The inputs to each round include a buffer value H_{i-1}, whose content is updated each round, a word W derived from the message block M_i using a message schedule, and an additive constant K extracted from the first n prime numbers by taking the first 32-bit for a 512-bit block or the first 64-bit for a 1024-bit block, as a

fractional parts of the cubic root of each prime number. These invariants remove any regularities in the original data by adding random patterns.

The message digest H_i is generated by adding the input of the previous round H_{i-1} to the output of the n^{th} round. Depending on the algorithm, the Addition modulo 2^{32} (or 2^{64}) is carried out independently for each word in the buffer with every corresponding word in H_{i-1}.

After processing each N b-bit block, the N^{th} stage yields the message digest H_N.

The message schedule generates n words from each message block for the function F. The words are generated according to Equation 2 except for the first 16 words, which are fetched directly from the message block (NIST, 2015).

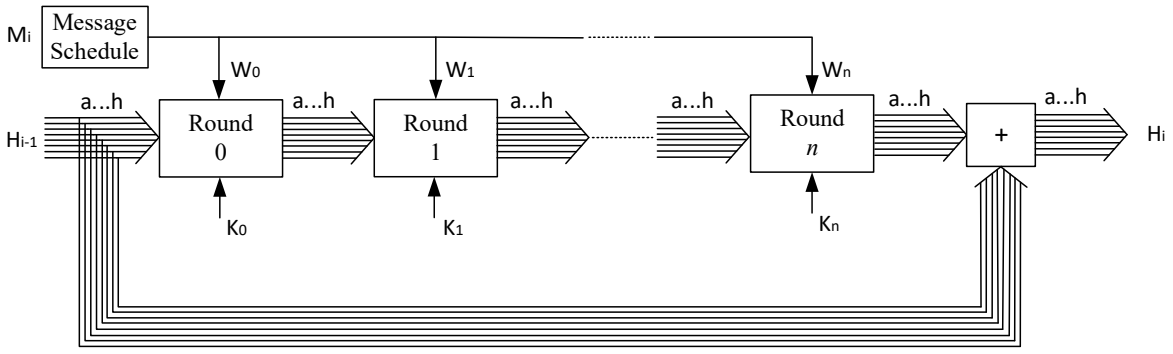

Figure 1: Processing of a Single b-Bit Block

$$W_t = \sigma_1^s(W_{t-2}) + W_{t-7} + \sigma_0^s(W_{t-15}) + W_{t-16} \quad (2)$$

where

$$\sigma_0^s(x) = ROTR^{u1}(x) \oplus ROTR^{u2}(x) \oplus SHR^{u3}(x) \quad (3)$$

$$\sigma_1^s(x) = ROTR^{v1}(x) \oplus ROTR^{v2}(x) \oplus SHR^{v3}(x) \quad (4)$$

$ROTR^n(x)$: Right rotation word x with n bits.

$SHR^n(x)$: Right shifts word x with n bits.

s = 256 for SHA-256 and SHA-224, 512 for SHA-512 and SHA-384

for s = 256 (u_1 = 7, u_2 = 18, u_3 = 3), for s = 512 (u_1 = 1, u_2 = 8, u_3 = 7)

for s = 256 (v_1 = 17, v_2 = 19, v_3 = 10), for s = 512 (v_1 = 19, v_2 = 61, v_3 = 6)

+ = addition modulo 2^{32} for SHA-256 (or 224) or 2^{64} for SHA-512 (or 384)

As is well known, to achieve a higher level of security, it is essential to apply Shannon's principles of confusion and diffusion. These principles are used through substitution and permutation. In each round, only the values of two words in the buffer change, while only the positions of the other six words change, as shown in Figure 2. The details of these functions are explained as follows (NIST, 2015).

$$T_1 = h + Ch(e,f,g) + (\sum_1^s e) + W_t + K_t \quad (5)$$

$$T_2 = (\sum_0^s a) + Maj(a,b,c) \quad (6)$$

$$h = g \quad (7)$$

$$g = f \quad (8)$$

$$f = e \quad (9)$$

$$e = d + T_1 \quad (10)$$

$$d = c \quad (11)$$

$$c = b \quad (12)$$

$$b = a \quad (13)$$

$$a = T_1 + T_2 \quad (14)$$

where

$$Ch(e,f,g) = (e\ AND\ f) \oplus (NOT\ e\ AND\ g) \quad (15)$$

$$Maj(a,b,c) = (a\ AND\ b) \oplus (a\ AND\ c) \oplus (b\ AND\ c) \quad (16)$$

$$\sum_0^s a = ROTR^{w1}(a) \oplus ROTR^{w2}(a) \oplus ROTR^{w3}(a) \quad (17)$$

$$\sum_1^s e = ROTR^{y1}(e) \oplus ROTR^{y2}(e) \oplus ROTR^{y3}(e) \quad (18)$$

t is the round number, between (0 – 63) for SHA-256, (0 and 79) for SHA-512

s = 256 for SHA-256 and SHA-224, 512 for SHA-512 and SHA-384

for s = 256 (w_1 = 2, w_2 = 13, w_3 = 22), for s = 512 (w_1 = 28, w_2 = 34, w_3 = 39)

for s = 256 (y_1 = 6, y_2 = 11, y_3 = 25), for s = 512 (y_1 = 14, y_2 = 18, y_3 = 41)

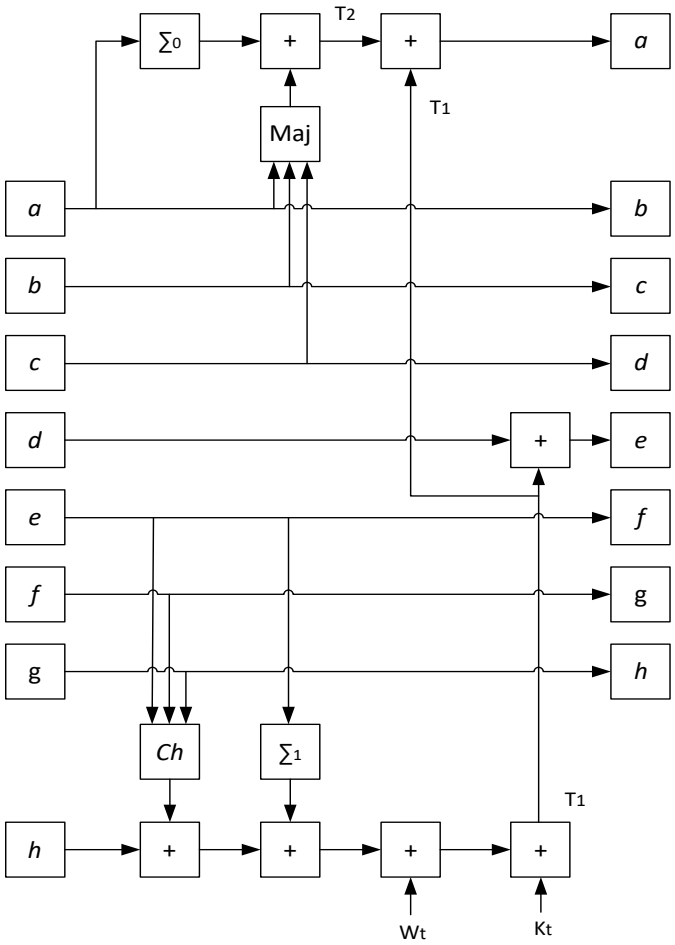

Figure 2: Round Operations

FPGA and System Generator

FPGA is a general-purpose integrated circuit that a designer can program to carry out a specific task without communicating with the manufacturer of the device. An FPGA has the ability to be reprogrammed even after it has been implemented, in contrast to an application-specific integrated circuit (ASIC), which can carry out a comparable function in an electronic system (Golshan, 2007, 2024). Designers can perform a wide range of logic and arithmetic operations by utilizing an FPGA, giving them access to a two-dimensional array of programmable resources (Bruno, 2024; Raj, 2017; Woods et al., 2017).

Compilation tools transform a designer's high-level abstractions into a low-level executable form, producing a bitstream file. This bitstream file, or configuration program, is downloaded into the device on the chip's static random-access memory (SRAM) to program it.

The Xilinx System Generator was the first to introduce the idea of generating an FPGA program from a high-level Simulink model. A system-level modelling tool called System Generator facilitates the design of FPGA hardware. This tool provides a high-level abstraction that is automatically compiled into the FPGA. Furthermore, this tool offers low-level abstractions that facilitate access to FPGA resources at the hardware level, enabling the creation of highly efficient FPGA designs (Xilinx, 2012).

FPGA Design Choices

FPGA design options can generally be classified into the following categories.

a. Iterative Looping Architecture
b. Loop Unrolling Architecture
c. Hybrid Architecture

The appropriate category is chosen based on the type of application, its internal design, and resource availability. Typically, it is a trade-off between throughput and area (Hwang, 2006; Maxfield, 2004). For example, the first category is appropriate for resource-constrained applications. Therefore, the focus is on optimal resource utilization rather than throughput. Where the operations of a single round are designed, followed by a register, and the data is then reprocessed iteratively through this piece of hardware for the number of rounds of the algorithm. The total number of cycles needed to process the algorithm usually equals the number of rounds since each round is typically processed in a single cycle.

The second category focuses on achieving optimal throughput, of course, at the expense of resource consumption. The hardware of a single round is replicated to the number of rounds, and the entire algorithm is typically executed in a single cycle. While the area expands by the number of rounds, i.e. multiplied by 64 for SHA-224 and SHA-256 or by 80 for SHA-512 and others.

The third category combines the two previous types to increase throughput while allowing for an acceptable increase in area. For example, instead of replicating the hardware of a single round by the number of rounds, it is duplicated two or four times, reducing the total number of clock cycles needed to execute the algorithm and thus increasing the throughput.

In addition to the categories mentioned above, there are two other important categories: parallel architecture and pipeline architecture. These two categories are functional in applications requiring high throughput and flexible area usage. Naturally, not all algorithms fit into these categories, depending on the algorithm's internal design and available resources. In general, these two types of architecture often overlap in design with the categories mentioned previously.

Hardware Design and Implementation

The hardware architectures of the SHA-256, SHA-256 ×2, SHA-256 ×4, and SHA-512 algorithms are designed and implemented on an FPGA using the Xilinx System Generator (XSG). The necessary netlist file is synthesized and simulated using Xilinx ISE Foundation 14.7 and ModelSim, following its generation by the XSG. The Xilinx XC7VX330T-3FFG1761, the target machine, is chosen because it met the design requirements.

The following describes the hardware design of the algorithms using the Xilinx System Generator:

a. ***Ch* and *Maj* functions:** The *Ch* function in Equation 15 and the *Maj* function in Equation 16 use the AND and XOR operations, while the *NOT* operation is only used in the *Ch* function. These operations can be implemented using *AND* (Xilinx logical block), *XOR* (Xilinx logical block), and *NOT* (Xilinx invertor), as shown in Figures 3 and 4. The *AND* and *XOR* functions are configured by setting the number of inputs to 2, the precision of the output to a 32- or 64-bit unsigned number (32-bit for SHA-256 and 64-bit for SHA-512), and a binary point to 0.

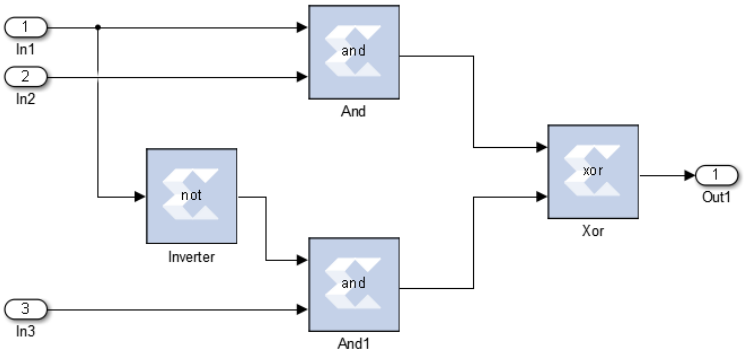

Figure 3: *Ch* Function

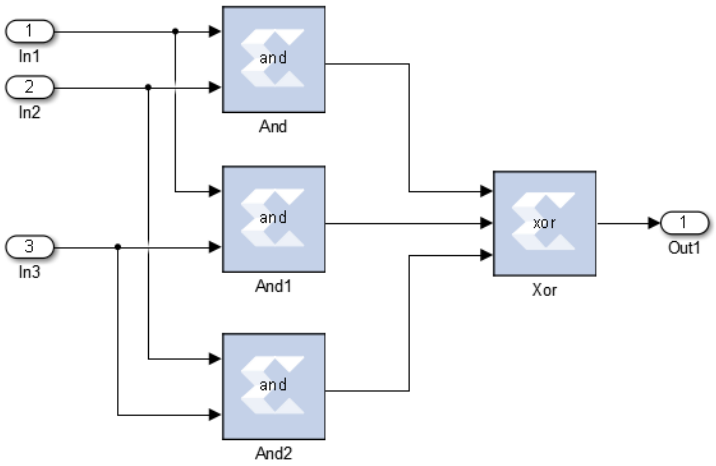

Figure 4: *Maj* Function

b. $\sum_0^s a$ **and** $\sum_1^s e$**:** $\sum_0^s a$ is represented in Equation 17 and $\sum_1^s e$ in Equation 18, they use the right-rotating circular shift register $ROTR^n(x)$ and *XOR*. $ROTR^n(x)$ can be implemented in the system generator using the Xilinx BitBasher block. It is configured by writing the expression (a = {b[n-1:0], b [31: n]}), where n is the number of offsets, and setting the output type to an unsigned number with a word length of 32 or 64 bits. The illustration of $\sum_0^s a$ is shown in Figure 5. For $\sum_1^s e$ it has the same design but with a different offset.

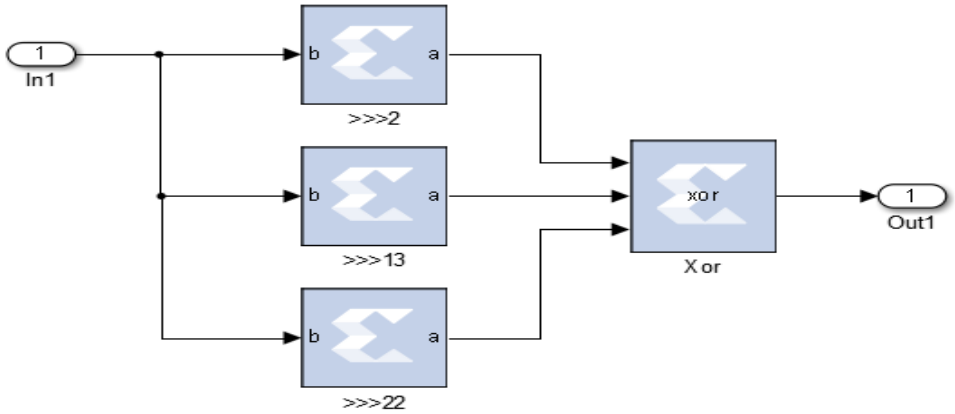

Figure 5: $\sum_0^s \boldsymbol{a}$ design

c. **Modular Addition:** This operation can be handled using Xilinx's Adder/Subtracter block. It is configured by specifying the addition operation and setting the output type to an unsigned number with the word length.
d. **Transposition or permutation:** It is achieved through hardwiring (equations 7, 8, 9, 11, 12, 13).
e. **The Constant K_t:** Consists of 64 constants of a 32-bit word for SHA-256 or 80 constants of a 64-bit word for SHA-512. These constants are precomputed and stored in Xilinx's single-port read-only memory (ROM). The ROM is initialized by writing the total number of constants in the memory depth and their constant values into an initial value vector, with the output type set to unsigned fixed point (32 or 64 bits) and the binary point set to 0.
f. **The Constants W_t:** They are generated according to Equation 2, as shown in Figure 6, which includes two other Equations 3 and 4, and are represented by $\sigma_0^s(x)$ **(**see Figure 7**)** and $\sigma_1^s(x)$. They use $ROTR^n(x)$, XOR, and the right shift register $SHR^n(x)$. $SHR^n(x)$ is implemented in the system generator using a Xilinx Binary Shift Operator block and is configured by setting the shift direction and the number of bits to be shifted, as well as setting the output type to an unsigned number with a word length. A multiplexer is also required to select the correct constant based on the round number, which is entered into the multiplexer selector, where multiplexer inputs represent the different round constants in order. The multiplexer block is called a Xilinx bus multiplexer and is configured by entering the number of inputs and the type of the outputs.

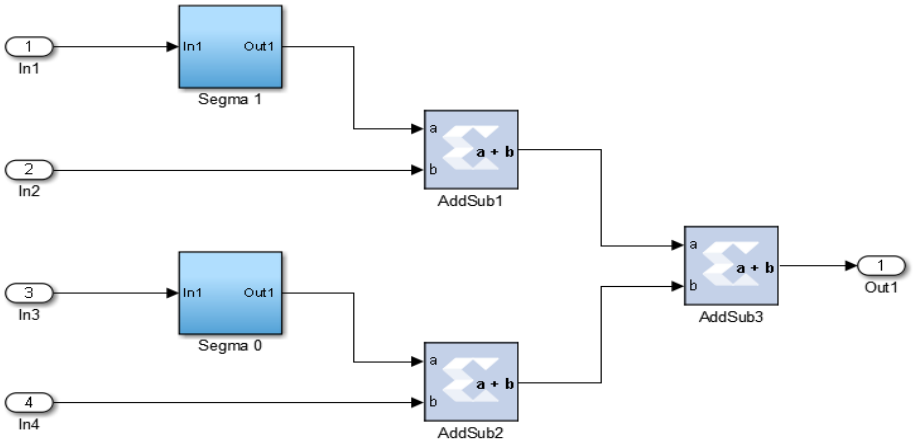

Figure 6: W_t Design

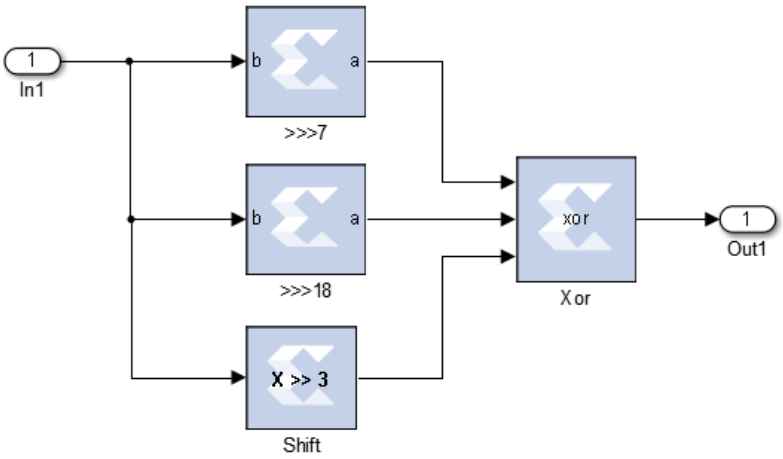

Figure 7: $\sigma_0^s(x)$ Design

g. **Registers:** Eight registers are used to store the output values from each round.

h. **Selector:** It consists of eight two-input multiplexers, used to feed the round inputs either with the initial values of H_i for the first round or with the outputs of the processed round for the other rounds, as partially shown in Figure 8.

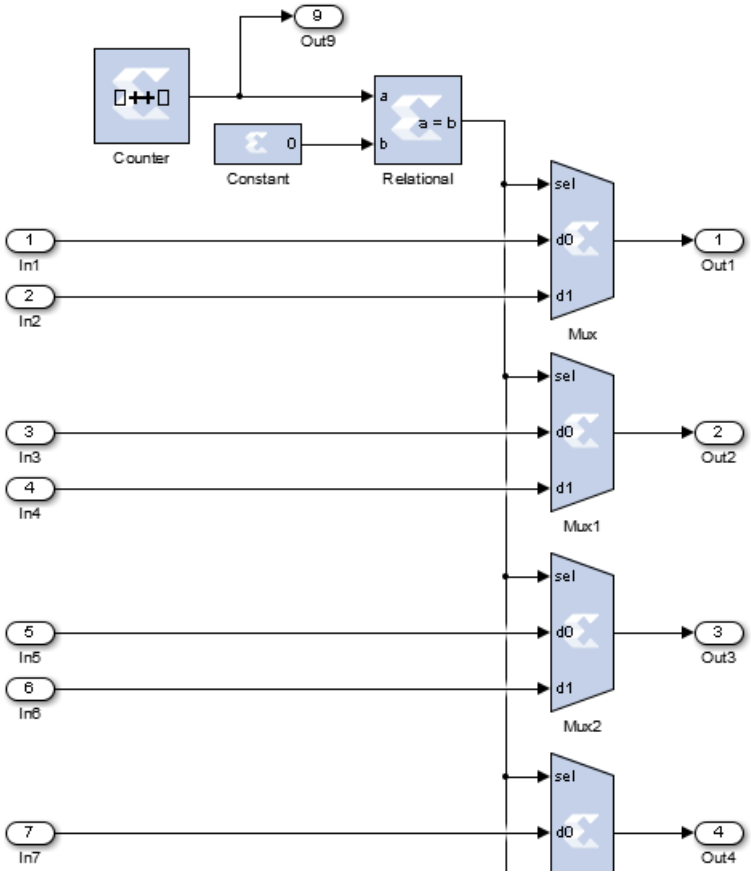

Figure 8: Selector Design

i. The designs of the main function, T_1 in Equation 5 and T_2 in Equation 6, are shown in Figures 9 - 11.

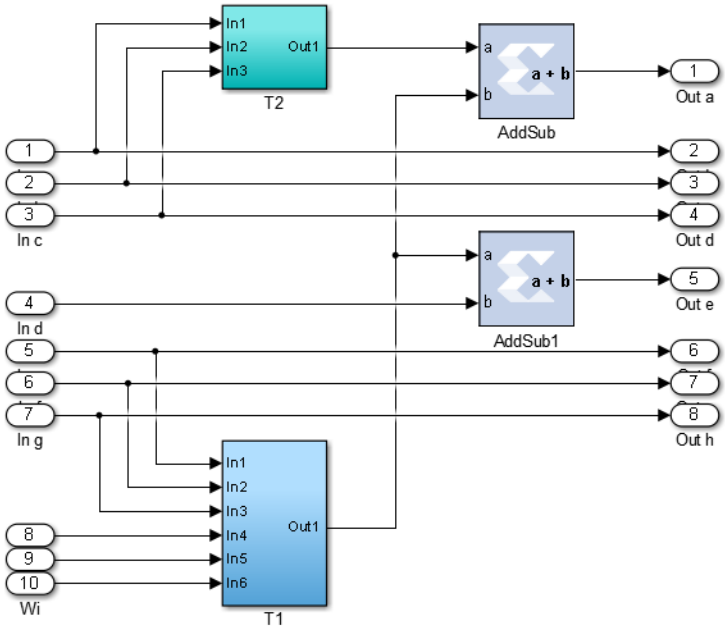

Figure 9: The main Function Design

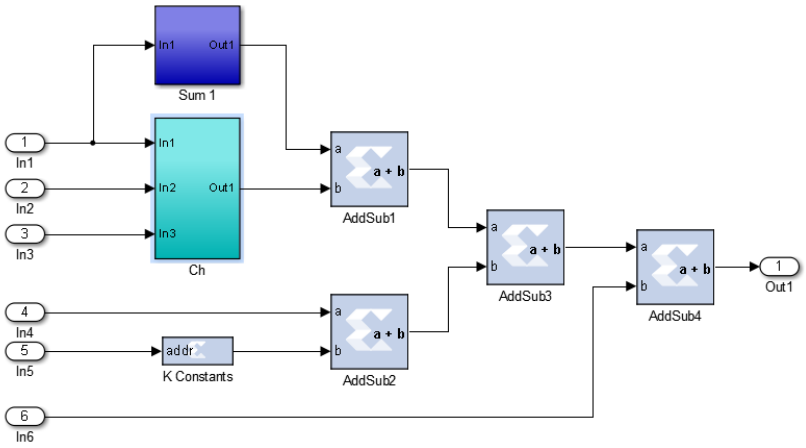

Figure 10: T_1 Design

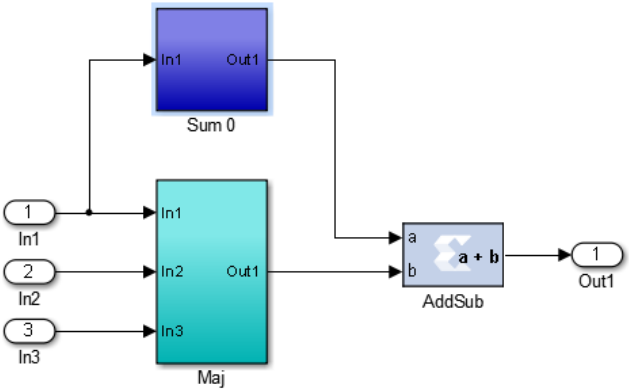

Figure 11: T_2 design

4 Results and Discussion

Table 2 and Figures 12-19 show the results of the hardware implementation of the SHA-256 and SHA-512 algorithms. Both designs share the same architecture, which is based on iterative looping architecture, making them suitable for resource-constrained applications. The other two algorithms, SHA-256 × 2 and SHA-256 × 4, are based on a hybrid architecture that combines iterative looping architecture and partial loop unrolling architecture by replicating two and four rounds in a single iteration. This architecture maintains a balance between improved performance and efficient resource utilization. All data words are executed in parallel for each operation in the iteration, and each iteration is executed in one clock cycle. Therefore, the total number of clock cycles needed to implement the proposed design equals the entire number of iterations of the SHA-2 algorithm.

SHA-256 performance improved when using a hybrid architecture that allows processing two rounds per iteration and improves even further when processing four rounds per iteration. This reduces the clock cycle count by half and a quarter.

Table 2: Implementation Results

Basic criteria		SHA-256	SHA-256 ×2	SHA-256 ×4	SHA-512
Performance	Throughput (Gbps)	1.021	1.667	1.905	1.331
	Maximum Frequency (MHz)	125.581	104.210	59.542	102.690
	Minimum Period (ns)	7.963	9.596	16.795	9.738
	Efficiency (Mbps / Slice)	3.066	5.021	5.755	1.904
Area Usage	Number of Slice Registers	333	332	331	699
	Number of Slice LUTs	7,027	7,363	7,852	17,619
	Number of occupied Slices	1,852	2,417	2,490	5,813
	Number of bonded IOBs	257	257	257	513
Supply power (W)	Total	0.394	0.413	0.415	0.608
	Dynamic	0.216	0.235	0.237	0.429
	Quiescent	0.178	0.178	0.178	0.179

The throughput and efficiency are calculated according to Equations 19 and 20, respectively.

$$Throughput = \frac{Number\ of\ Processed\ Bits\ \times Maximum\ Frequency}{Number\ of\ Clock\ Cycles} \quad (19)$$

$$Efficiency = \frac{Throughput}{No.of\ Slices} \quad (20)$$

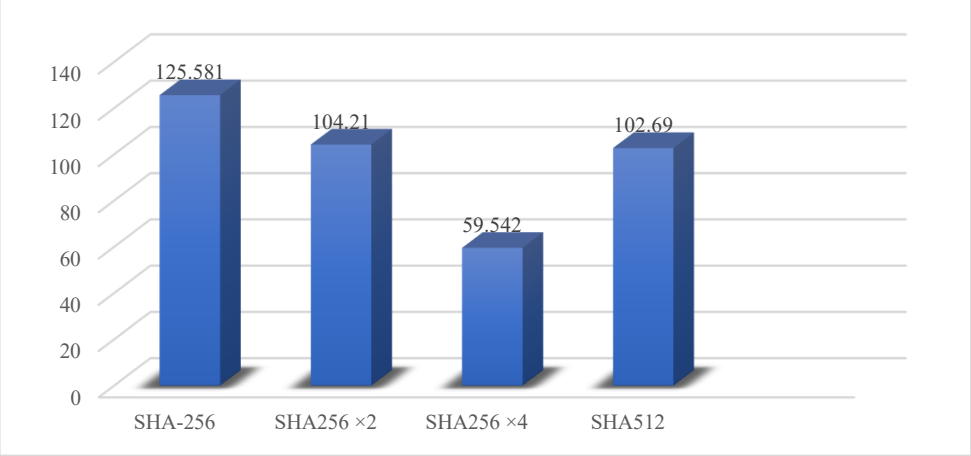

Figure 12: Maximum Frequency (MHz)

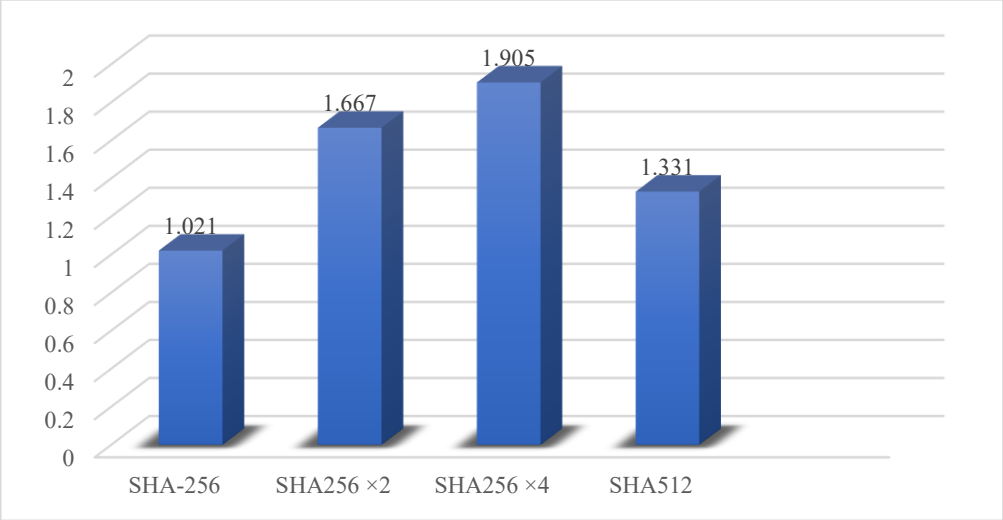

Figure 13: Throughput (Gbit/Sec)

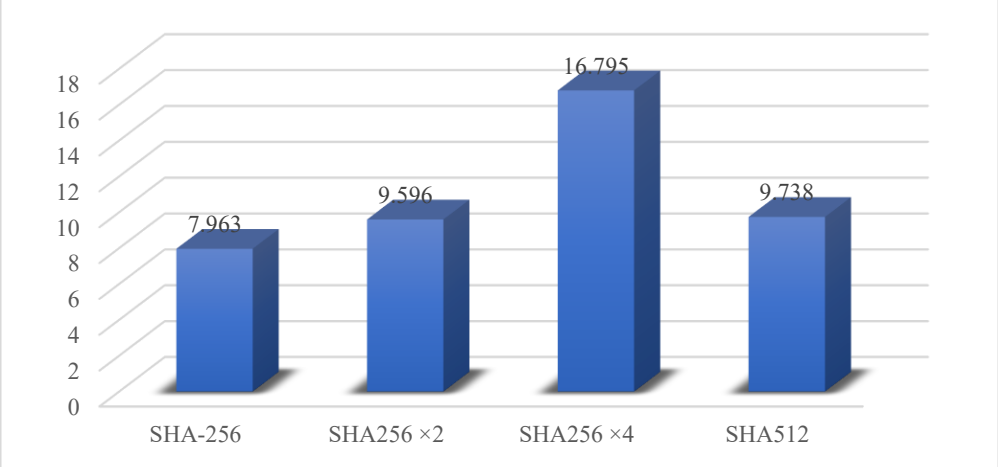

Figure 14: Min. Period (ns)

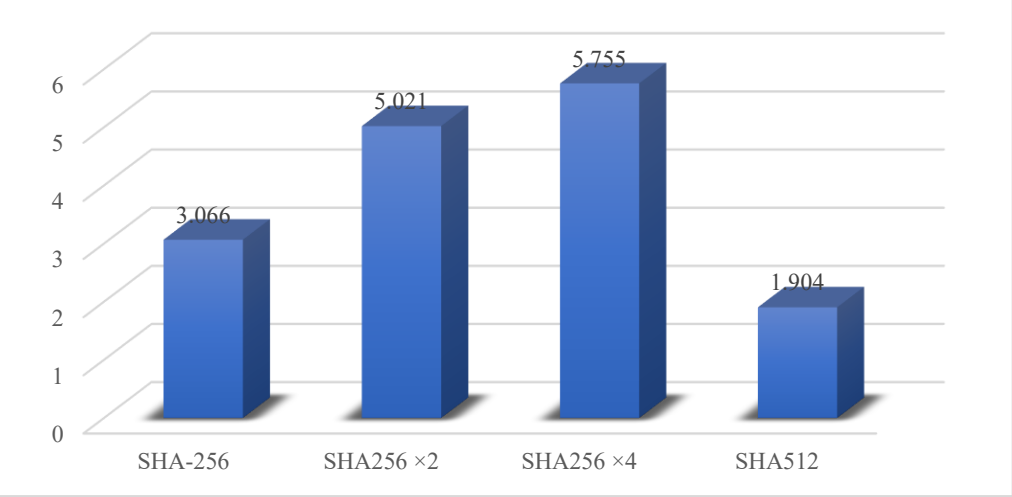

Figure 15: Efficiency (Mbps/Slice)

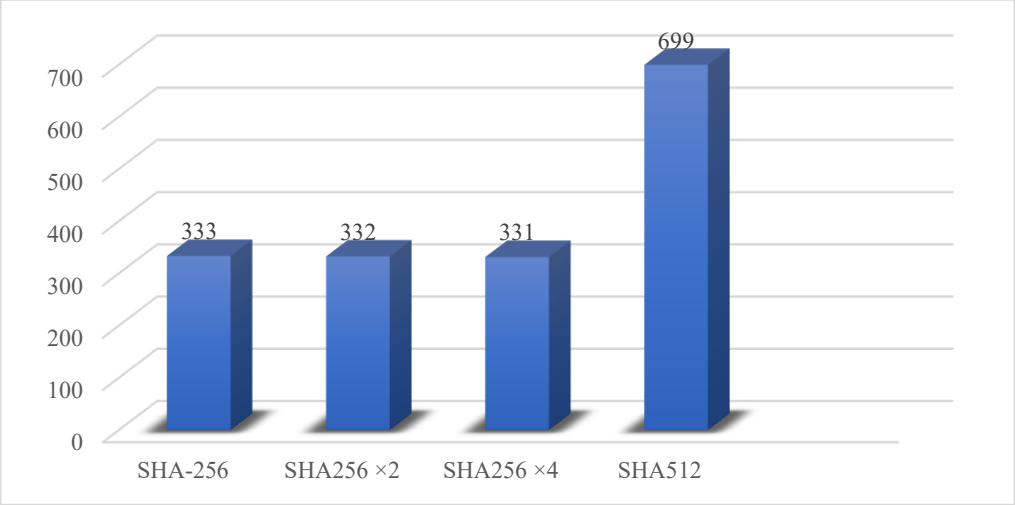

Figure 16: No. of Slice Reg.

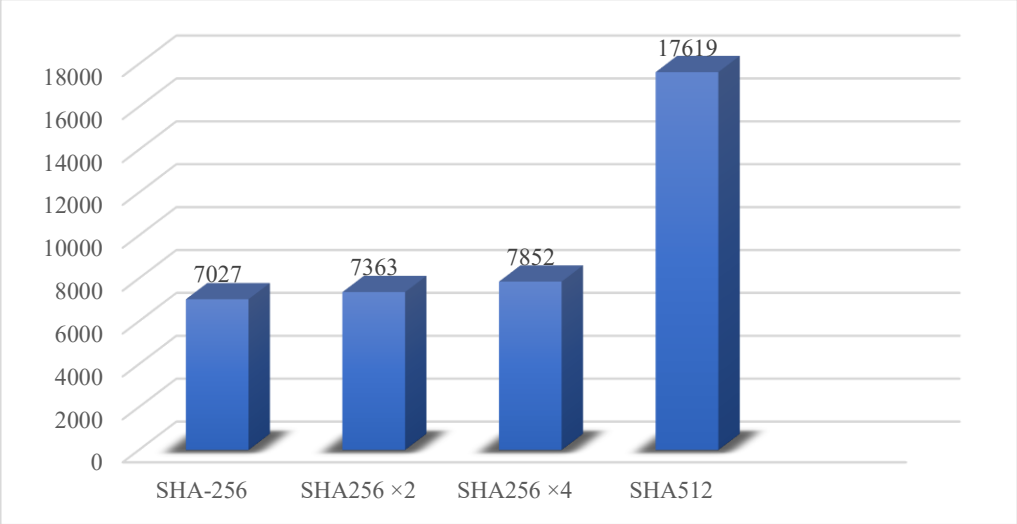

Figure 17: No. of Slice LUTs

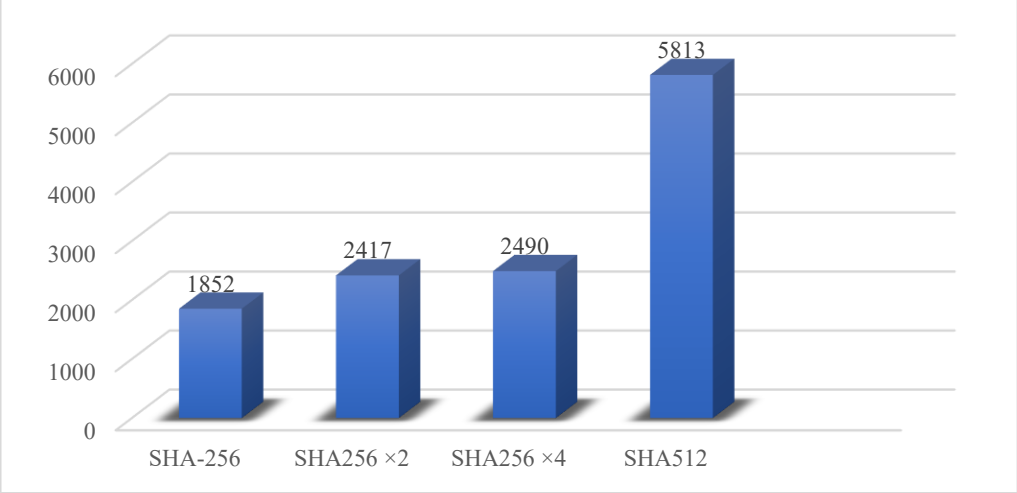

Figure 18: No. of Occupied Slices

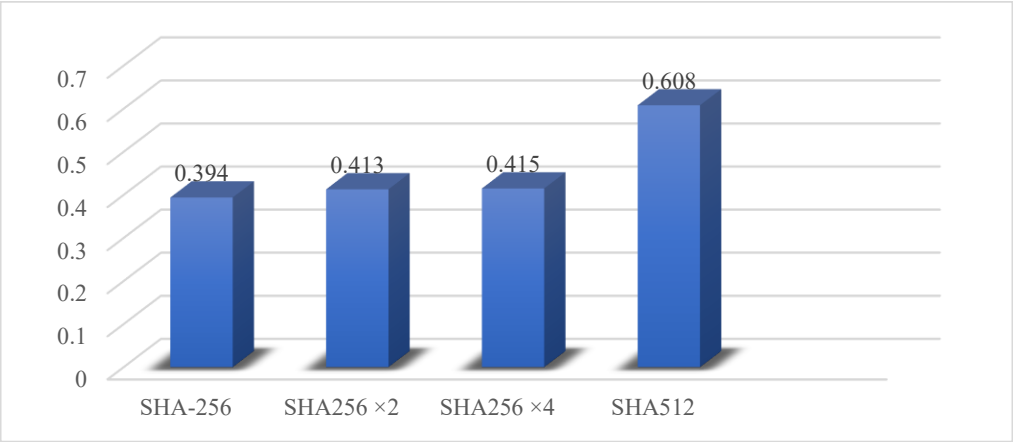

Figure 19: Supply Power (W)

Table 3 and Figures 20 and 21 show a comparison between SHA-256 and other related works. In general, the proposed design achieved the desired results for the intended application, which is resource-constrained applications. However, higher throughput can be achieved, as in (Bensalem et al., 2021; Chen & Li, 2020; Li et al., 2024) using different architecture and optimization techniques at the expense of resource utilization.

Table 3: Results Comparison

References	Throughput Gbit/sec	Max. Freq. MHz	Slice Area	D-Power W	Device	Optimization Technique
(Wong et al., 2018)	1.405	354	197	---	Virt6x-6 (-3)	Folded-2 Architecture with 4-2 adder compressor
(Kammoun et al., 2020)	0.917	181	6367	0.1	XC7Z020	Unrolling the external loops of the compression function
(Gad et al., 2020)	0.637	83.33	275	0.013	Virtex-7	Gated clock conversion, arithmetic resource sharing, structural modelling of small building blocks
(Kieu-Do-Nguyen et al., 2021)	1.04	139.04	327	0.072	Artix-7 (xc7a200t)	Two-stage pipeline architecture
(Santos et al., 2024)	1.4025	11.13	12618	0.503	Virtex-6 xc6vlx240t-1ff1156	16-core architecture for parallel implementation
Our design	1.905	59.542	331	0.237	Virtex-7 XC7VX330T-3FFG1761	4-Unfolded architecture

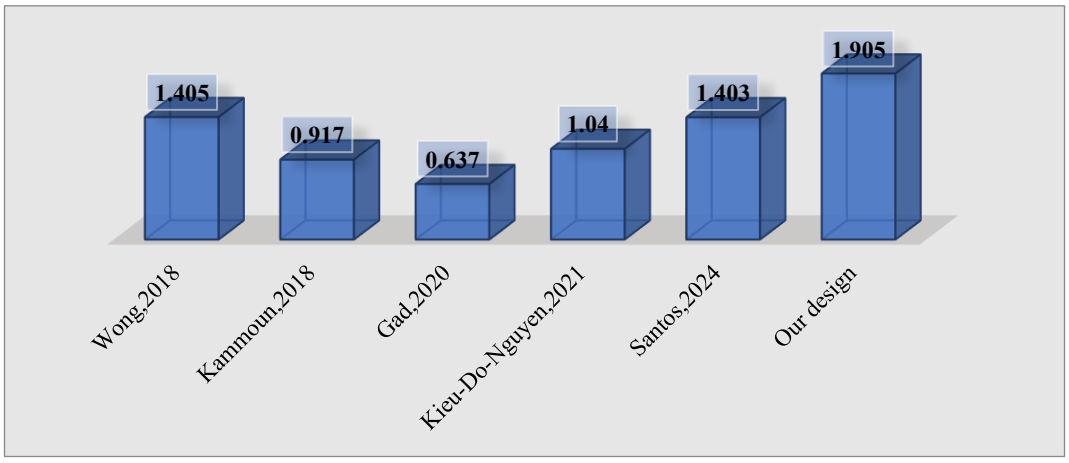

Figure 20: Results Comparison - Throughput (Gbps)

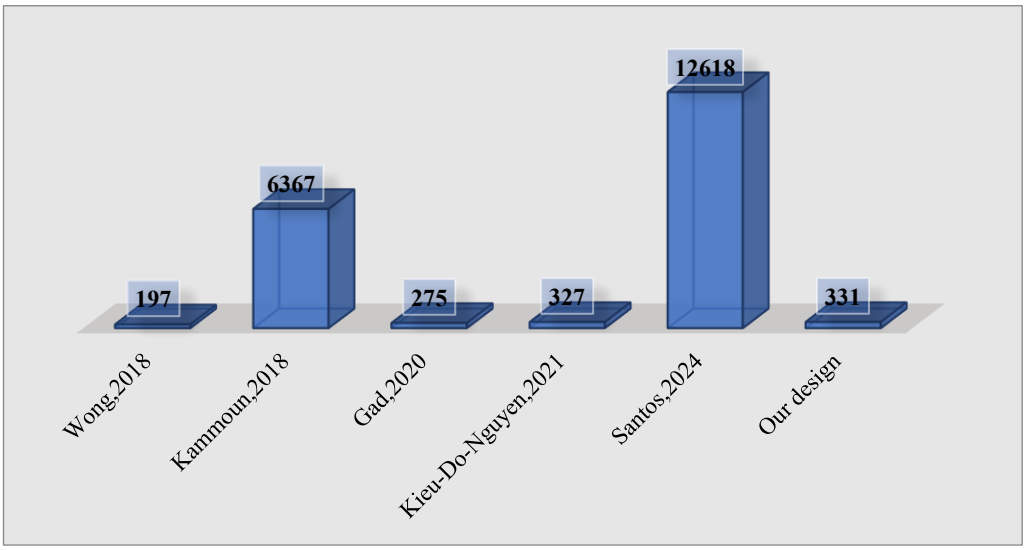

Figure 21: Results Comparison - Slice Registers

5 Conclusion

In this paper, hardware architectures for SHA-256 and SHA-512 algorithms are designed and implemented on a Xilinx XC7VX330T-3FFG1761 board using Xilinx ISE 14.7. The design also suits other SHA-2 families by truncating the redundant bits to the required digest lengths. The design aims to improve performance with minimal complexity for resource-constrained applications. Therefore, an iterative looping architecture is proposed. In addition, a hybrid architecture is also designed to enhance performance further by implementing more than one round per iteration, thus replicating the hardware of a single round. For power consumption optimization, a multi-layered strategy is needed, including user behavior, system-level design, intelligent software, and hardware efficiency.

References

[1] Barker, E., & Roginsky, A. (2019). *Transitioning the use of cryptographic algorithms and key lengths* (No. NIST SP 800-131Ar2; p. NIST SP 800-131Ar2). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-131Ar2

[2] Bensalem, H., Blaquiere, Y., & Savaria, Y. (2021). Acceleration of the Secure Hash Algorithm-256 (SHA-256) on an FPGA-CPU Cluster Using OpenCL. *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5. https://doi.org/10.1109/ISCAS51556.2021.9401197.

[3] Bruno, F. (with Eschemann, G.). (2024). *The FPGA Programming Handbook: An essential guide to FPGA design for transforming ideas into hardware using SystemVerilog and VHDL* (1st ed.). Packt Publishing Limited.

[4] Chen, Y., & Li, S. (2020). A High-Throughput Hardware Implementation of SHA-256 Algorithm. *2020 IEEE International Symposium on Circuits and Systems (ISCAS),* 1–4. https://doi.org/10.1109/ISCAS45731.2020.9181065.

[5] Dobbertin, H. (1998). Cryptanalysis of MD4. *Journal of Cryptology*, *11*(4), 253–271. https://doi.org/10.1007/s001459900047

[6] Dominikus, S. (2002). A hardware implementation of MD4-family hash algorithms. 9th International Conference on Electronics, Circuits and Systems, 3, 1143–1146. https://doi.org/10.1109/ICECS.2002.1046454.

[7] Gad, A. H., Abdalazeem, S. E. E., Abdelmegid, O. A., & Mostafa, H. (2020). Low power and area SHA-256 hardware accelerator on Virtex-7 FPGA. *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, 181–185. https://doi.org/10.1109/NILES50944.2020.9257922

[8] Golshan, K. (2007). Physical design essentials: An ASIC design implementation perspective. *Springer.*

[9] Golshan, K. (2024). ASIC Design Implementation Process: A Complete Framework (1st ed. 2024). *Springer Nature Switzerland.* https://doi.org/10.1007/978-3-031-58653-8

[10] Hwang, E. O. (2006). Digital logic and microprocessor design with VHDL.

[11] Jain, A., & Babu, K. A. (2024). An Examination of Cutting-edge Design and Construction Methods Concerning Green Architecture and Renewable Energy Efficiency for Tier-II Cities of India. *Archives for Technical Sciences*, *2*(31), 57–69. https://doi.org/10.70102/afts.2024.1631.057

[12] Kammoun, M., Elleuchi, M., Abid, M., & BenSaleh, M. S. (2020). FPGA-based implementation of the SHA-256 hash algorithm. *2020 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS)*, 1–6. https://doi.org/10.1109/DTS48731.2020.9196134

[13] Kieu-Do-Nguyen, B., Hoang, T.-T., Pham, C.-K., & Pham-Quoc, C. (2021). A Power-efficient Implementation of SHA-256 Hash Function for Embedded Applications. *2021 International Conference on Advanced Technologies for Communications (ATC)*, 39–44. https://doi.org/10.1109/ATC52653.2021.9598264.

[14] Kumar, C. V. S. R., & Nelakuditi, U. R. (2021). Hardware/Software Co-Design Using ZYNQ SoC. *Journal of VLSI Circuits and Systems, 3*(1), 14–18. https://doi.org/10.31838/jvcs/03.01.03

[15] Lee, S.-H., & Shin, K.-W. (2018). An efficient implementation of SHA processor including three hash algorithms (SHA-512, SHA-512/224, SHA-512/256). *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, 1–4. https://doi.org/10.23919/ELINFOCOM.2018.8330578

[16] Li, T., Cheng, C., Wang, R., Wang, C., Zou, X., & Yu, D. (2024). A High Performance Hardware Implementation of SHA-256 Algorithm. *2024 IEEE 4th International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*, 477–481. https://doi.org/10.1109/ICIBA62489.2024.10869248

[17] Ling Bai & Shuguo Li. (2009). VLSI implementation of high-speed SHA-256. *2009 IEEE 8th International Conference on ASIC*, 131–134. https://doi.org/10.1109/ASICON.2009.5351591

[18] Maxfield, C. (2004). *The design warrior's guide to FPGAs: Devices, tools and flows*. Newnes Elsevier.

[19] NIST. (1993). *Secure hash standard* (No. NIST FIPS 180; pp. 1–22). National Institute of Standards and Technology (U.S.). https://doi.org/10.6028/NIST.FIPS.180

[20] NIST. (1995). *Secure hash standard* (No. NIST FIPS 180-1; pp. 1–28). National Institute of Standards and Technology (U.S.). https://doi.org/10.6028/NIST.FIPS.180-1

[21] NIST. (2002). *Secure hash standard* (No. NIST FIPS 180-2; pp. 1–76). National Institute of Standards and Technology (U.S.). https://csrc.nist.gov/files/pubs/fips/180-2/final/docs/fips180-2.pdf

[22] NIST. (2008). *Secure hash standard* (No. NIST FIPS 180-3; pp. 1–33). National Institute of Standards and Technology (U.S.). https://csrc.nist.gov/files/pubs/fips/180-3/final/docs/fips180-3_final.pdf

[23] NIST. (2015). *Secure hash standard* (No. NIST FIPS 180-4; pp. 1–36). National Institute of Standards and Technology (U.S.). https://doi.org/10.6028/NIST.FIPS.180-4

[24] Orozco, L., & Ttofis, H. (2025). Energy harvesting techniques for sustainable embedded systems: Design and applications. *SCCTS Journal of Embedded Systems Design and Applications, 2*(1), 67–78.

[25] Polk, T., Chen, L., Turner, S., & Hoffman, P. (2011). *Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms* (No. RFC6194; p. RFC6194). RFC Editor. https://doi.org/10.17487/rfc6194
[26] Raj, A. A. B. (2017). FPGA-Based Embedded System Developer's Guide (1st ed). *CRC Press.*
[27] Rivest, R. L. (1991). The MD4 Message Digest Algorithm. In A. J. Menezes & S. A. Vanstone (Eds.), Advances in Cryptology-CRYPT0' 90 (Vol. 537, pp. 303–311). *Springer Berlin Heidelberg.* https://doi.org/10.1007/3-540-38424-3_22
[28] Rogaway, P., & Shrimpton, T. (2004). Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. In B. Roy & W. Meier (Eds.), *Fast Software Encryption* (Vol. 3017, pp. 371–388). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-25937-4_24
[29] Santos, C. E. B., Silva, L. M. D. D., Torquato, M. F., Silva, S. N., & Fernandes, M. A. C. (2024). SHA-256 Hardware Proposal for IoT Devices in the Blockchain Context. *Sensors*, *24*(12), 3908. https://doi.org/10.3390/s24123908
[30] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017). The First Collision for Full SHA-1. In J. Katz & H. Shacham (Eds.), Advances in Cryptology – CRYPTO 2017 (Vol. 10401, pp. 570–596). *Springer International Publishing.* https://doi.org/10.1007/978-3-319-63688-7_19
[31] Sun, C. (2023). The Bit Query for Labels in a Binary Tree-Based Anti-Collision Recognition Algorithm. *Indian Journal of Information Sources and Services*, *13*(2), 68–75. https://doi.org/10.51983/ijiss-2023.13.2.3853
[32] Tang, U., Krezger, H., & LonnerbyRakob. (2024). Design and validation of 6G antenna for mobile communication. *National Journal of Antennas and Propagation, 6*(1), 6–12.
[33] Thomson. Yuvaraj, D., Saravanakumar, G., Prasath, J. S., & Sathish Kumar, S. (2019). Design and implementation of modeling and tuning of first-order process with dead time using PID controller. *International Journal of Communication and Computer Technologies, 7*(1), 1-6.
[34] William Stallings. (2023). Cryptography and network security: Principles and practice (Eighth edition, global edition). *Pearson.*
[35] Wong, M. M., Pudi, V., & Chattopadhyay, A. (2018). Lightweight and High Performance SHA-256 using Architectural Folding and 4-2 Adder Compressor. *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 95–100. https://doi.org/10.1109/VLSI-SoC.2018.8644825
[36] Woods, R., McAllister, J., Lightbody, G., & Yi, Y. (2017). FPGA-based implementation of signal processing systems (Second edition). *John Wiley & Sons Inc.*
[37] Xilinx. (2012). System Generator for DSP User Guide. https://www.origin.xilinx.com/support/documents/sw_manuals/xilinx14_7/sysgen_user.pdf

Author Biography

M.F. Al-Gailani is an associate professor at Al-Nahrain University, specializing in information security. He holds a bachelor's and master's degrees in computer engineering from the University of Technology and a Ph.D. degree in computer engineering from Newcastle University, UK. Dr. Al-Gailani has worked in academia and research for over twenty years, publishing more than twenty-five research articles in his field. He has held several administrative positions throughout his academic career, most recently as head of the Cybersecurity Engineering Department.