ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

Design and Implementation of SHA-512 Hash Function IP Core

Sen Li, Jilin Tang, Jian Tang

Automation Research Institute Co.,Ltd. of China South Industries Group , Mianyang, 621000, China

Abstract

To meet the requirements of verifying the correctness, integrity and whether the configuration file has been tampered with during the remote upgrade process of the FPGA configuration file, a SHA-512 IP core is designed to implement the digital signature verification of the configuration file on the FPGA. To facilitate interaction with other modules, the IP core adopts the AXI-stream interface to receive message data. A pipeline design approach is used to make message padding, message expansion and message compression proceed in sequence, reducing the processing delay of the IP core. No third-party IP cores are used in the design, making it easy to transplant and deploy on other domestic FPGAs. Finally, the designed IP core is tested on the AMD XC7K325TFFG676 chip. The test results show that the IP core can perform hash operations on messages and has stable performance.

Keywords

Hash Function; SHA-512; FPGA; IP Core; Message Padding; Message Compression.

1. Introduction

During operation, FPGA chips in underwater and aerospace equipment often require remote upgrades of the programs running within them. These upgrades typically utilize wireless transmission methods such as wireless networks. Since wireless transmission is employed for program upgrades, data integrity, correctness, and security must be ensured throughout the process. Digital signatures [1] are a common method for verifying data integrity and correctness. Common hash functions include four major categories: MD5, SHA-1, SHA-2, and SHA-3[2]-[3]. The SHA-2 series comprises SHA-224, SHA-256, SHA-384, and SHA-512. Due to its enhanced security, SHA-512 is widely adopted.

The SHA-512 hash function is widely employed to verify data integrity and correctness, ensuring data remains unaltered during transmission. It finds extensive application in password storage, blockchain technology, and digital signatures[4]-[5]. In digital signatures, SHA-512 generates a hash value for the message. This hash is then encrypted using a cryptographic algorithm and appended to the message. Upon receiving the message, the recipient calculates a hash value for the new message and compares it with the hash value sent by the sender to verify the message's integrity and correctness. To address the demand for SHA-512 hash function implementation in FPGAs while reducing algorithm development complexity for designers, shortening equipment development cycles, and enhancing system stability, this paper designs an SHA-512 IP core to fulfill the requirement for hash encryption operations within FPGAs.

2. Hash Function SHA-512

The SHA-512 function is a widely used hash function capable of mapping data of any length to a 512-bit hash value. Its overall structure is shown in Figure 1. First, the input message undergoes formatting to adjust its length to a multiple of 1024 bits-a process called message padding. The message is then divided into blocks of 1024 bits each. During each operation, a message block and a

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

hash vector (the initial vector for the first iteration) are sequentially extracted from the message and fed into the round operation and summation module (comprising message expansion and message compression sub-modules). After completing one iteration of SHA-512 rounding and summation, the hash value from this operation is output and reused with the next message block for repeated iteration. This continues until the hash value from the final block (block N) is output, representing the SHA-512 digest for the message.

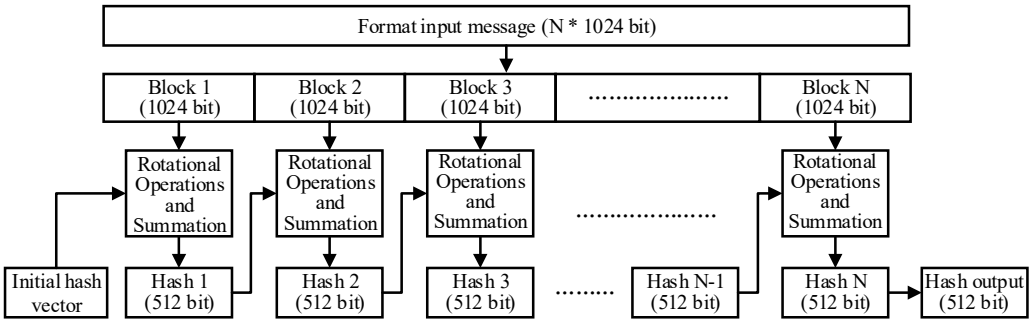

Figure 1. Overall Structure of SHA-512 Message Processing

2.1 Message Padding

The SHA-512 function processes messages in 1024-bit units. Therefore, when the input message length is not an integer multiple of 1024 bits, the SHA-512 function cannot directly process the input message. Padding is required to make the length a multiple of 1024 bits before performing the operation[6]-[7]. Message padding consists of two parts:

Append a single binary "1" bit to the end of the input message, followed by k binary "0" bits. Let the original message length be L bits. The padded message length must satisfy: $L+1+k \equiv 896 \pmod{1024}$.

Building upon the first step, append a 128-bit length block whose value is the original length of the input message in bits. After completing both padding steps, the entire message length becomes an exact multiple of 1024 bits.

As illustrated in Figure 2, the padding process is demonstrated using the input message "SIECK". The ASCII code value for "SIECK" totals 40 bits. Following the padding rules: Step 1: First append 1 bit of binary 1, then append k bits of binary 0 (1024 - 40 - 1 - 128 = 855); Step 2: Append a 128-bit binary length field. This 128-bit length value represents the binary form of the original input message length (L).

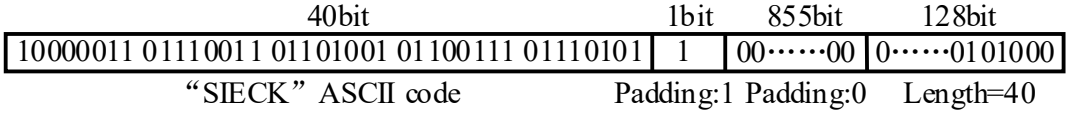

Figure 2. Message "SIECK" Padding Example

2.2 Round Operation and Summation

The Round Operation and Summation module comprises message expansion and message compression. The message compression consists of 80 rounds (Round 1 to Round 80) of compressed iterative operations, structured as shown in Figure 3. The input to the entire round operation and summation module is the message block t and the hash value output from the previous round operation and summation ($Hash_{t-1}$). The output is the hash value from the current operation ($Hash_t$).

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

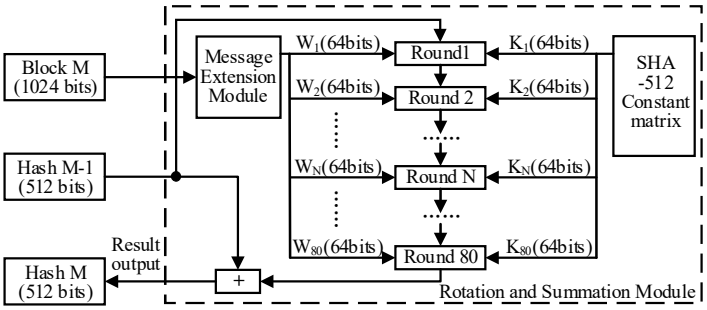

Figure 3. Round Operation and Summation Module

2.2.1 Message Expansion

The purpose of message expansion is to process the padded input message in units of 1024 bits (message blocks), expanding it through expansion operations into 80 data words (W_0 to W_{79}), each 64 bits in length. The message expansion process is as follows: (1) First, the 1024-bit message block is divided into 16 sub-blocks (M_0 to M_{15}), each 64 bits long. M_0 is the first sub-block, M_{15} is the last sub-block, and they correspond sequentially to W_0 to W_{15}. (2) Using W_0–W_{15} as the foundation, iterative calculations are performed according to formula (1) [8]-[10] to generate W_{16}–W_{79}. Upon completion of W_{79}, the expansion calculation for the current message block concludes. The structure of the message block expansion iteration is illustrated in Figure 4.

$$W_t = \begin{cases} M_t, (0 \le t < 16) \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, (16 \le t < 80) \end{cases} \quad (1)$$

Where:

(1) $\sigma_0(x) = \mathrm{POTR}^1(x) \oplus \mathrm{POTR}^8(x) \oplus \mathrm{SHR}^7(x)$;

(2) $\sigma_1(x) = \mathrm{POTR}^{19}(x) \oplus \mathrm{POTR}^{61}(x) \oplus \mathrm{SHR}^6(x)$;

(3) $\mathrm{ROTR}^n(x)$: Represents rotating x clockwse by n positions, placing the shifted-out bits in the high-order positions;

(4) $\mathrm{SHR}^n(x)$: Represents shifting x right by n positions, discarding the shifted-out bits and padding the high-order positions with zeros;

(5) $\oplus$: Represents the bitwise exclusive OR operation.

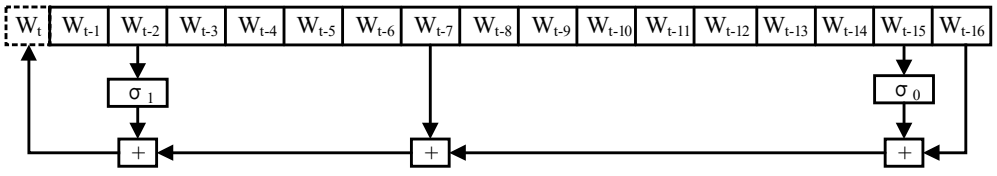

Figure 4. Message Block Extension Iterative Computation Structure

2.2.2 Message Compression

Message compression consists of 80 rounds of compression iterations performed by the compression function. During each iteration, the compression function takes as input the hash value output from the previous round's operation, the message word W_t, and the SHA-512 constant K_t. Its output is the iteration result for the current round. The computational structure of each compression function iteration is illustrated in Figure 5. Registers A through H represent eight 64-bit registers [11] used to store the 512-bit hash values required for iterative operations. During initialization, they are assigned the hash function's initial values H_0 through H_7[12] (as shown in Table 1). Register A is initialized with H_0, and register H is initialized with H_7.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

Table 1. Hash Function Iteration Initial Values

Symbol	Numeric	Symbol	Numeric
H_0	64'h6a09e667f3bcc908	H_4	64'h510e527fade682d1
H_1	64'hbb67ae8584caa73b	H_5	64'h9b05688c2b3e6c1f
H_2	64'h3c6ef372fe94f82b	H_6	64'h1f83d9abfb41bd6b
H_3	64'ha54ff53a5f1d36f1	H_7	64'h5be0cd19137e2179

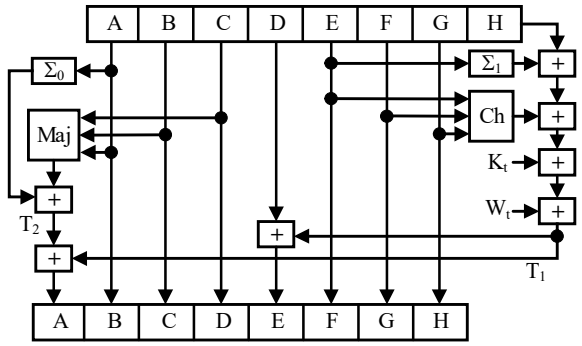

Figure 5. Iterative Structure of the SHA-512 Compression Function

$\sum_0$, $\sum_1$, Maj and Ch are function operators, whose calculation expressions are shown in Equations (2) to (5).

$$\mathrm{Maj}(x,y,z) = (x\&y) \oplus (x\&z) \oplus (y\&z) \tag{2}$$

$$\mathrm{Ch}(x,y,z) = (x\&y) \oplus (\sim x\&z) \tag{3}$$

$$\sum_0(x) = \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x) \tag{4}$$

$$\sum_1(x) = \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x) \tag{5}$$

The computational process for message compression is as follows:

(1) If processing the first 1024-bit message block, first use eight initial hash values H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7 (each 64 bits) as the initial values for the first iteration of the 80-round process, using $\mathrm{Hash}_{t-1}(H_0^{t-1} \sim H_7^{t-1})$ as input. Initialize registers A, B, C, D, E, F, G, H with H_0, H_1, H_2, H_3, H_4, H_5, H_6, H_7 to the 8 registers A, B, C, D, E, F, G, H for initialization. Otherwise, the hash values computed from the previous message block through 80 rounds of iterative processing are used as the initial values for the first iteration of Hasht-1 in the current 80-round compression iteration.

(2) After initialization, perform 80 iterative cycles on A, B, C, D, E, F, G, H. The pseudocode is as follows:

for t = 0 to 79

H = G; G = F; F = E; E = D+T1； D = C; C = B; B = A; A =T_1 + T_2;

endfor

where:

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

$$T_1 = h + \sum_1(E) + \mathrm{Ch}(E,F,G) + K_t + W_t \quad (6)$$

$$T_2 = \sum_0(A) + \mathrm{Maj}(A,B,C) \quad (7)$$

(3) After completing 80 cycles of computation, add the resulting values A, B, C, D, E, F, G, H to the iterative initial values used in Step (1) to obtain the hash values for this operation ($H_0^t \sim H_7^t$). As shown below:

$$H_0^t = A + H_0^{t-1},\ H_1^t = B + H_1^{t-1},\ H_2^t = C + H_2^{t-1},\ H_3^t = D + H_3^{t-1} \quad (8)$$

$$H_4^t = E + H_4^{t-1},\ H_5^t = F + H_5^{t-1},\ H_6^t = G + H_6^{t-1}, H_7^t = H + H_7^{t-1} \quad (9)$$

Repeat the operations described in steps (1) to (3) above until all message blocks have been processed. The compressed output of the final message block is the SHA-512 hash value of the entire message, which is the message digest.

3. IP Core Design and Implementation

3.1 Overall IP Core Design

The overall structure of the SHA-512 IP core is shown in Figure 6. It consists of two submodules: the "Message Padding and Expansion Module" and the "Message Compression Module." The AXI-stream interface inputs the message byte stream with an 8-bit data width. The "Message Padding and Expansion Submodule" comprises a message padding state machine and a message expansion function. It pads the input message to a multiple of 1024 bits. Upon completing processing of a message block, the message expansion function block immediately latches the message block and sequentially outputs the expanded word W_t. The Message Compression Submodule comprises a compression iteration module and a SHA-512 constant matrix, performing message compression operations to output the computed hash value.

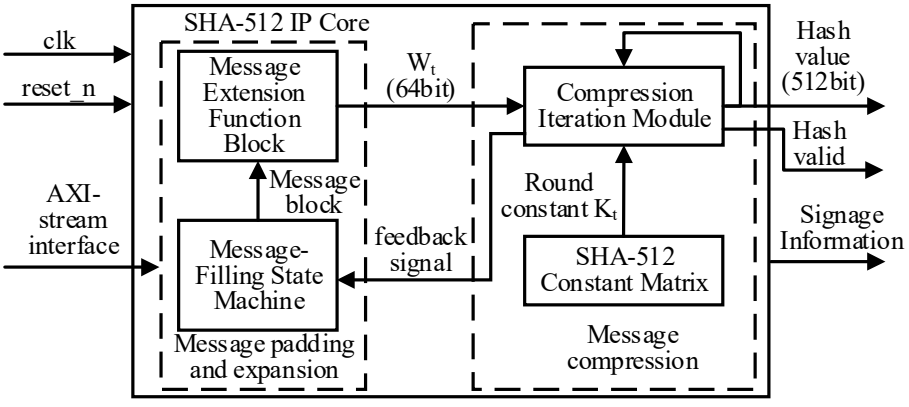

Figure 6. Overall Structure of the SHA-512 IP Core

3.2 Message Padding and Extension Module Design

3.2.1 Message Padding State Machine

During message padding, a state machine is employed to process the input message byte stream from the AXI-stream interface. The state transition diagram for the entire state machine is shown in Figure 7. Upon power-up, the state machine enters the idle state. When the last data flag is detected as valid, it transitions to the "Receive Message Byte -1" state. Upon detecting a valid data flag, it transitions to the "Receive Message Byte -2" state. After transitioning to "Receive Message Byte -1," it receives

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

one byte of message data, then enters the “Fill 0x80” state, followed by the “Fill 0x00” state. Upon completing the 0x00 fill, it transitions to the “Fill Length Field” state. After filling the length field, it enters the “End State” and subsequently returns to the “Idle State.” Upon transitioning to the “Receive Message Byte -2” state, the device continuously receives the message byte stream in this state until the final data flag becomes active, completing message data reception and entering the “Fill 0x80” state.

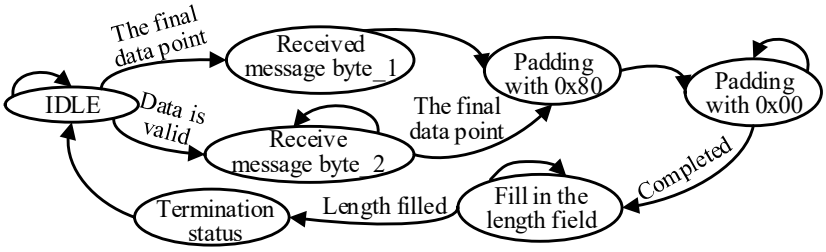

Figure 7. Message Filling State Machine

3.2.2 Message Expansion Function Design

The message expansion function receives messages in units of message blocks (1024 bits). Upon each complete message block output by the message padding state machine, the expansion function immediately latches the message block and simultaneously divides it into the first 16 expansion words W_0 to W_{15}. The correspondence between the message block and expansion words is illustrated in Figure 8, where byte0 represents the first byte input in the message stream. Subsequently, the extension words are output, the extension iteration calculation is performed, and the extension words are shifted. At the rising edge of each clock cycle, one extension word is output from W0 while simultaneously shifting. The shift relationship is: $W_t = W_{t+1}$, $(0 \le t \le 14)$, W_{15} = extension calculation expression, continuing until all 80 extension words are shifted and output.

Unlike the iterative structure in Figure 4, the actual design employs only 16 registers to complete the expansion function iteration. The function expression's value is directly assigned to the W_{15} register. Since the function values from the iterative computation are temporarily stored by combinational logic during processing, these 16 registers suffice to represent the storage of 17 variable values.

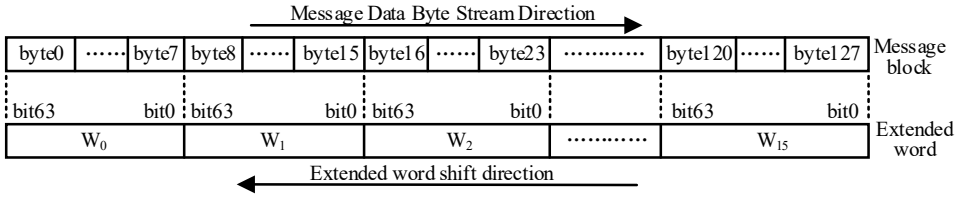

Figure 8. Correspondence Between Message Block Byte Stream and Extension Words W0 to W15

3.2.3 Message Padding and Extension Module Simulation

To validate the correctness of the “Message Padding and Extension Module” design, simulations were conducted on this module. The results are shown in Figures 9-11. Figure 9 displays the overall simulation results for an input message length of 129 × 8 bits (1032 bits). When `axi_tvalid = 1`, the IP core receives the input message byte stream. Upon receiving the first byte of the message, the module outputs the Hash vector initialization pulse (‘hash_vector_init = 1’) to initialize the Hash vector's starting value. Upon completing reception of a message block, the module immediately outputs the extended words, pulls word_valid_out high, and outputs the message extended words. After completion, it waits for the next message block. Figure 10 shows simulation details of the message input process, while Figure 11 displays simulation details of the extension word output process. It can be observed that after 80 64-bit extension words are fully output, word_valid_out is

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

immediately pulled low. Upon awaiting the next message block, the extension words for that block are computed and output.

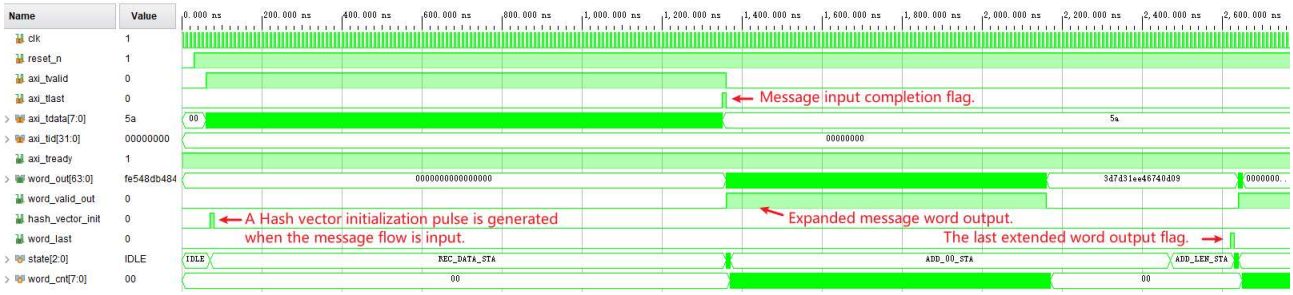

Figure 9. Overall Simulation Results of the Message Padding and Expansion Module

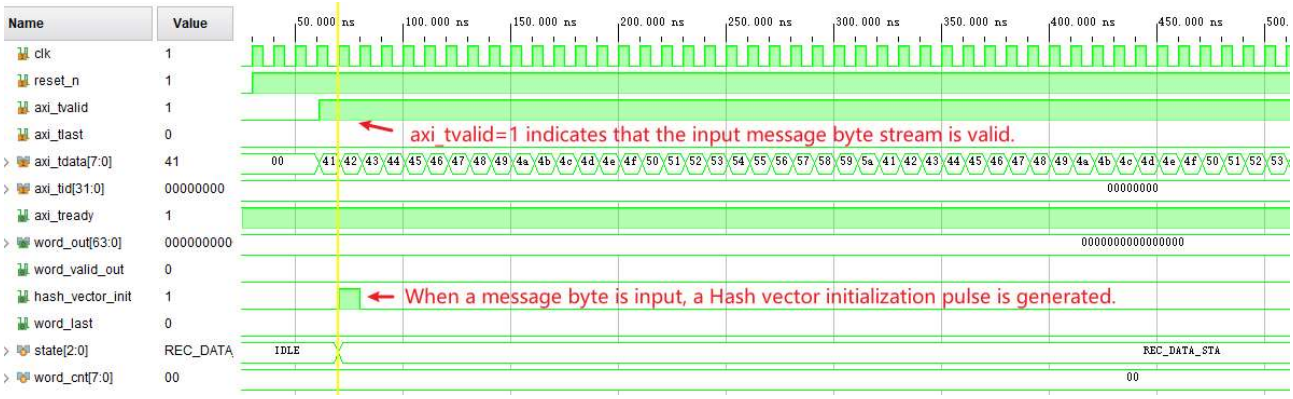

Figure 10. Message Input Process Details

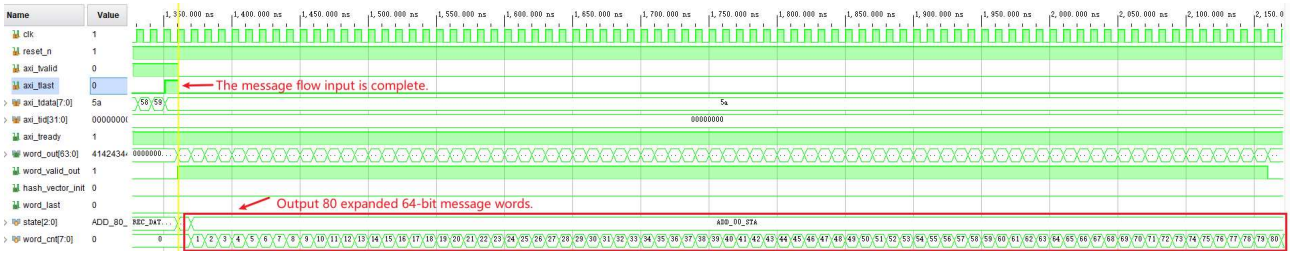

Figure 11. Extended Character Output Simulation Details

3.3 Message Compression Module Design

The message compression module implements 80 rounds of iterative hash function calculations, outputting the final iteration result (hash value). The port design of the message compression module is shown in Figure 12. When hash_init = 1, the values of the internal A–H registers are initialized to H0–H7. When word_valid = 1, the expanded word input to the word_in port is latched, and a constant is retrieved from the SHA-512 constant matrix to perform one round of hash function compression iteration. After completing 80 rounds of compression iteration, the module waits for the next expanded word generated from the data block input to initiate a new 80-round compression iteration. When `word_last = 1` and the iteration counter reads 79, `hash_valid` outputs a single-cycle high pulse. The value at the `hash_out` port represents the message digest generated by applying the SHA-512 function to the input message.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

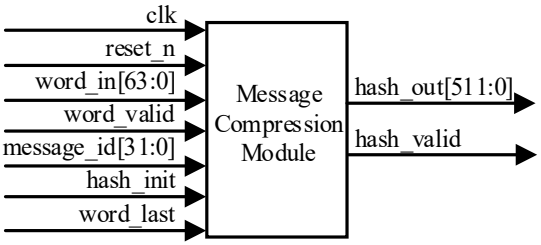

Figure 12. Message Compression Module Port Diagram

The simulation results of the message compression module are shown in Figures 13 and 14. Figure 13 displays the overall simulation results for the message compression module when the input message is "ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXY " as the input message. Figure 14 shows simulation details, revealing that when hash_valid=1, the hash_out[511:0] port outputs the final hash calculation value, which is "85bdfc4308894e121e2e5699aa66b6540da6bd5151e8f0ca543747b4f8da337073ea8a8a428c03d4d4a15797547b8aee285bbde1d0db8752ee18082f8d78a13c", confirming the calculation result is correct.

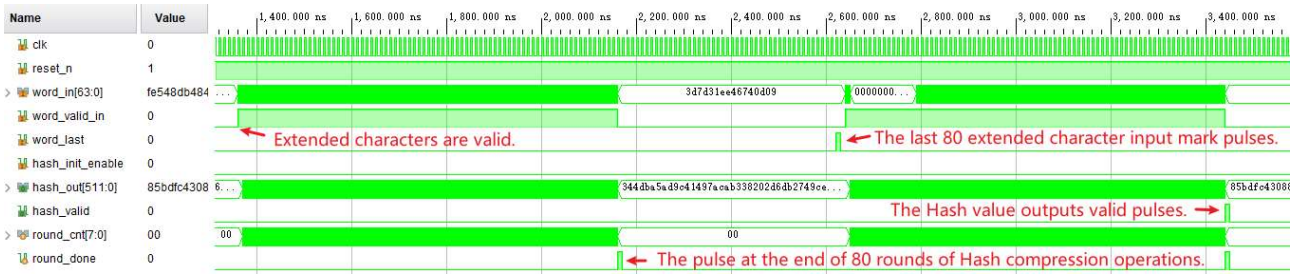

Figure 13. Overall Simulation Results of the Message Compression Module

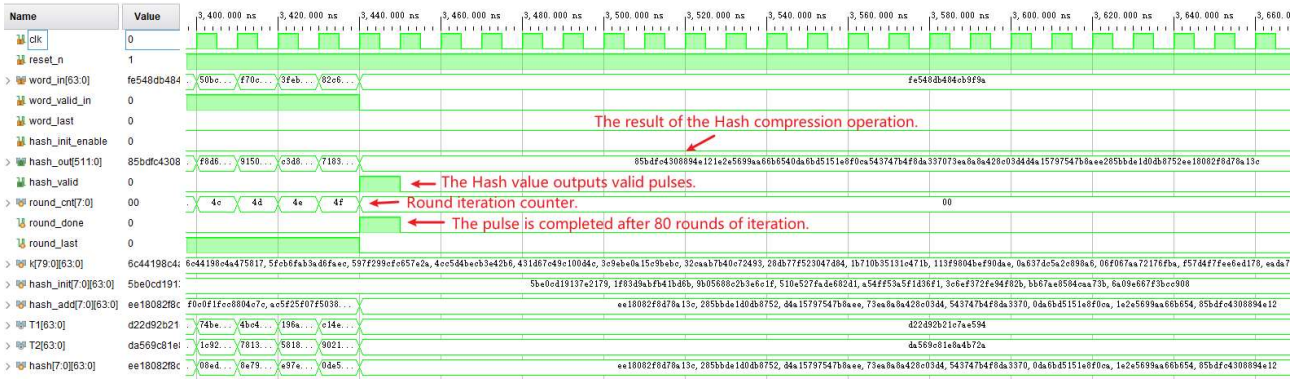

Figure 14. Message Compression Module Simulation Details

4. IP Core Integration and Packaging

4.1 IP Core Top-Level Design

The top-level module integrates all submodules, encapsulating them into a complete IP core. The port design of the top-level module is shown in Table 2, comprising three categories: system ports, AXI

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

ports, and hash operation output ports. The RTL source code undergoes synthesis, after which it is encapsulated as an IP core. The call interface of the encapsulated IP core is illustrated in Figure 15.

Table 2. IP Core Top-Level Ports

Port Name	I/O	Description	Port Name	I/O	Description
clk	I	System Clock	axi_tid[31:0]	I	AXI Frame Sequence Number
reset_n	I	System Reset	axi_tready	O	AXI Data Ready Flag
axi_tvalid	I	AXI Input Data Valid	hash_out[511:0]	O	Hash Operation Message Digest Output
axi_tlast	I	AXI Frame End Marker	hash_valid	O	Hash Operation Message Valid Flag
axi_tdata[7:0]	I	AXI Data Port			

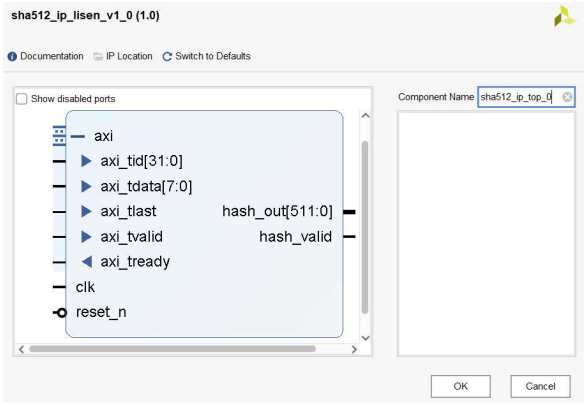

Figure 15. IP Core Call Configuration Window

4.2 Resource Consumption and Applicability

The resource consumption after synthes-izing the IP core is shown in Table 3. When the target device is set to XC7K325TFFG676-2 and the system clock frequency is constrained to 150MHz (6.6ns cycle), the minimum established margin after synthesis is 0.619ns. Therefore, the maximum clock frequency at which this IP core can operate is: f_{max} = 1 / (6.6ns - 0.619ns) = 167.19 MHz. The IP core is entirely implemented using pure System Verilog design without incorporating any third-party IP cores. Consequently, it is compatible with FPGA chips from internati-onal companies such as Intel, AMD, and Lattice, as well as domestic FPGA chips from companies including Guowei, Fudan Micro, Anlu Technology, and Zhongke Yihai Micro.

Table 3. IP Core Resource Consumption

Resource Name	Consumption Quantity/Unit	Resource Name	Consumption Quantity/Unit
Slice LUTs	2823	Muxes	2
Slice Registers	3276	Block RAM	0

4.3 IP Core Throughput Performance

The IP core receives input message byte streams via an AXI interface with an 8 bit data bus width. When continuous external message data byte streams exceed 1024 bits, the IP core processes input messages in real-time. Internally, the IP core incorporates a 128 × 8 bit buffer space. Upon filling this buffer, message data is immediately fetched in parallel from the buffer on the rising edge of the system clock (clk) for message expansion operations. Simultaneously, new message bytes can be instantly

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

stored into the buffer, enabling real-time continuous reception of the message byte stream. When the IP core operates at a system clock frequency of 150MHz, its maximum throughput during message byte stream reception is: 150MHz × 8 bits = 1.2Gbit/s. This capability satisfies the hash compression computation requirements for message data during Gigabit Ethernet transmission.

5. Actual Verification

To validate the correctness of the IP core, the designed IP core was instantiated into the verification program. A VIO (Virtual Input Output) module was used to input test stimuli (message byte stream data) to the IP core. The VIO was also employed to inspect the message hash values. Two test results were randomly selected from the test outcomes, as shown in Figures 16 and 17. When the input character message is "20250507", the message hash value is "29ead4225497b34bd0079f284a61989 54bbbcfc3b7d90ac21a73e18d41a4b7822bdc27b9ac2960cc7b13d65acb45b9d137d106334785d4a25 30d461d91c7fa3c''. The result is correct as verified by the calculation of hash_out[511:0] in Figure 16. When the input message is 'CAEPSWAI', the message's hash value is "742acdfe2881f0899b8eca79f2cd94f0df1ea1db22c8bfcceb639f22952ab0fb9e973ffe4640b9e7a582 9d2fcbf2bcc5c731d9e07197f94bbf063c0798f99312", and the result is correct as shown by the calculation of hash_out[511:0] in Figure 17.

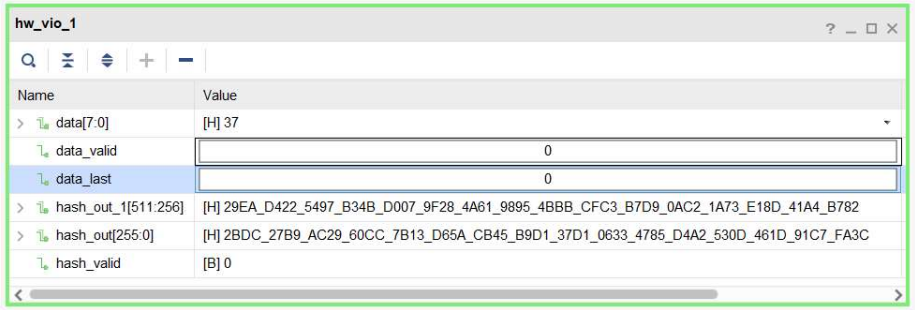

Figure 16. Message Hash Operation Result 01

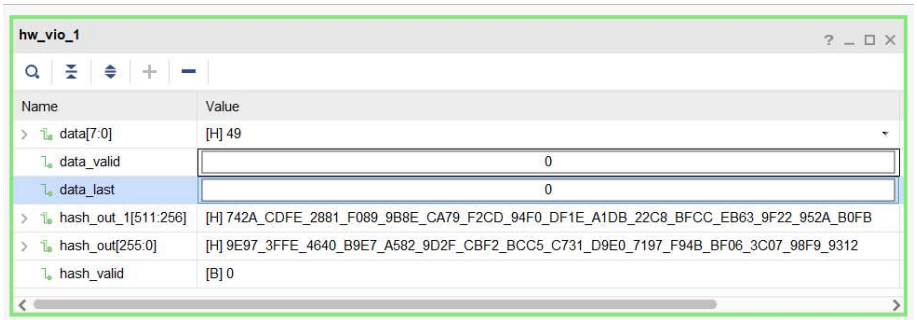

Figure 17. Message Hash Computation Result 02

6. Conclusion

This paper addresses the requirement for verifying data integrity and security during FPGA remote upgrades by designing a SHA-512 hash function IP core. This core fulfills the FPGA's need to perform hash operations on remote upgrade data. The integrated IP core design facilitates direct deployment in other engineering projects, shortening equipment development cycles and reducing coupling between program modules. This design provides a reference framework for implementing data integrity and security verification in FPGA systems.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202511_11(11).0013

Acknowledgments

The authors acknowledge the support provided by the Special Purpose Computer Division, China South Industries Group Automation Research Institute Co., Ltd.

References

[1] Nie Y. Implementation of a Hybrid Encryption Heterogeneous Computing System Based on FPGA [D]. Wuxi: Jiangnan University, 2022: 15-16.

[2] Lu S. Design and Implementation of an FPGA-Based MD Structure Hash Algorithm Accelerator [D]. Jinan: Shandong University, 2022: 8-10.

[3] Tian X. Optimization and System Design of FPGA-Based SHA-3 Algorithm Hardware Implementation [D]. Xi'an: Xidian University, 2019:8-13.

[4] Ge C. Fast Implementation and Application of GPU-Based SHA-2 Hash Algorithm [D]. Shanghai: Shanghai Jiao Tong University, 2018:23-25.

[5] Chen H D. Design of a Data Encryption System Based on AES Algorithm and SHA-512 [J]. Cybersecurity Technology and Application, 2023, (06): 26-29.

[6] Xi S X, Zhang W N, Zhou Q L, et al. High-Throughput Implementation of SHA512 Algorithm Based on Mimetic Computing [J]. Computer Engineering and Science, 2018, 40(08): 1344-1350.

[7] Xi S X. Optimized Implementation of SHA Series Functions on Reconfigurable Computing Platforms [D]. Zhengzhou: Zhengzhou University, 2017: 10-11.

[8] Xiao B H, Zheng Y N, Long J M, et al. Information Security Design of QR Codes Based on SHA512 Hash Function and Rijndael Encryption Algorithm [J]. Computer Systems Applications, 2015, 24(07): 149-154.

[9] Zheng B W. Design and Implementation of an FPGA-Based Hash Algorithm Accelerator [D]. Wuxi: Jiangnan University, 2022:12-13.

[10] Zhang X J. Research on Hash Functions and Boolean Functions Based on Quantum Computing [D]. Xuzhou: China University of Mining and Technology, 2023:18-20.

[11] Lin S Y. Full Pipeline Hardware Implementation and Optimization of SHA-2 and MD5 Algorithms Based on FPGA [D]. Xiamen: Xiamen University, 2018:47-49.

[12] Wang P. Research on Hardware Design of Post-Quantum Cryptographic Algorithms Based on SNTRUP [D]. Harbin: Harbin Engineering University, 2023:25-28.