An Efficient and Low-Power Design of the SM3 Hash Algorithm for IoT

Xin Zheng, Xianghong Hu, Jinglong Zhang, Jian Yang, and Shuting Cai

1 School of Automation, Guangdong University of Technology, Guangzhou 510006, China; xinzheng9209@gmail.com (X.Z.); xianghong_hu@163.com (X.H.); zjl_jesse@163.com (J.Z.); shutingcai@gdut.edu.cn (S.C.); xmxin@gdut.edu.cn (X.X.)

2 Company of Chipeye Microelectronics Foshan Ltd., Foshan 528200, China

Received: 5 August 2019; Accepted: 11 September 2019; Published: 14 September 2019

Abstract: The Internet-of-Things (IoT) has a security problem that has become increasingly significant. New architecture of SM3 which can be implemented in IoT devices is proposed in this paper. The software/hardware co-design approach is put forward to implement the new architecture to achieve high performance and low costs. To facilitate software/hardware co-design, an AHB-SM3 interface controller (AHB-SIC) is designed as an AHB slave interface IP to exchange data with the embedded CPU. Task scheduling and hardware resource optimization techniques are adopted in the design of expansion modules. The task scheduling and critical path optimization techniques are utilized in the compression module design. The proposed architecture is implemented with ASIC using SMIC 130 nm technology. For the purpose of comparison, the proposed architecture is also implemented on Virtex 7 FPGA with a 36 MHz system clock. Compared with the standard implementation of SM3, the proposed architecture saves the number of registers for approximately 3.11 times, and 263 Mbps throughput is achieved under the 36 MHz clock. This design signifies an excellent trade-off between performance and the hardware area. Thus, the design accommodates the resource-limited IoT security devices very well. The proposed architecture is applied to an intelligent security gateway device.

Keywords: the cryptographic algorithm; SM3; AHB-SIC; optimization

1. Introduction

Information security plays a very important role [1,2]. Although IoT technology provides many benefits, it also brings in various security threats such as the attack vulnerability of the hard-coded security key and the user privacy information leaking problem [3,4]. A lot of previous work has been done to mitigate and address potential security threats. For example, the static or dynamic analysis of the firmware and the source code running on IoT devices is used to discover and cope with the potential vulnerabilities for IoT devices [5]. An Interference Mitigation Risk Aware (IMRA) framework is proposed to settle the problem of mitigating the interference imposed by the intruders so as to protect the proper operation of passive RFID networks [6]. There are also studies focusing on designing new lightweight algorithms or optimizing the original cryptography algorithms [7,8]. A novel tiny symmetric encryption algorithm (NTSA) providing enhanced security for the transfer of text files through the IoT network by introducing additional key confusions dynamically for each round of encryption is proposed in [9]. In [10], a Function-based Access Control scheme in IoT (IoT-FBAC) is presented, and it uses an Identity-based Encryption (IBE) scheme. The data masking and the
encryption algorithm are utilized in many proposed solutions to protect sensitive information, but these solutions reduce the availability of original data and increase the time delay [4].

In all the above-mentioned work, the hash function, which belongs to the one-way encryption algorithms that compress the message with arbitrary length to the digest with fixed-length [11], is usually applied. The typical hash algorithms include MD5, SHA-0, SHA-1, SHA-2, SHA-3 and SM3 [12–16]. Since the SHA-0, SHA-1 and MD5 have not been secure enough, we will not discuss them here. Table 1 shows a comparison between SHA-2, SHA-3, and SM3 [17]. In order to ensure the security of the hash algorithm, the digest size should not be too short, as shown in Table 1.

Table 1. Comparison of Hash Algorithms (sizes and security are specified in bits).

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>SHA-224</th>
<th>SHA-256</th>
<th>SHA-384</th>
<th>SHA-512</th>
<th>SHA3-224</th>
<th>SHA3-256</th>
<th>SHA3-384</th>
<th>SHA3-512</th>
<th>SM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message size</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{128}$</td>
<td>$&lt;2^{128}$</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>$&lt;2^{64}$</td>
</tr>
<tr>
<td>Block size</td>
<td>512</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
<td>1152</td>
<td>1088</td>
<td>832</td>
<td>576</td>
<td>512</td>
</tr>
<tr>
<td>Digest size</td>
<td>224</td>
<td>256</td>
<td>384</td>
<td>512</td>
<td>224</td>
<td>256</td>
<td>384</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>Security</td>
<td>112</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>112</td>
<td>128</td>
<td>192</td>
<td>256</td>
<td>128</td>
</tr>
</tbody>
</table>

SM3 is a hash algorithm published by the Security Commercial Code Administration Office of China in 2010 and recognized by the ISO/IEC international standard [11,18]. SM3 can be used for a digital signature and identity authentication in different applications, and for the IoT application, SM3 should be designed to possess the characteristics of high performance, low power and low costs. The structure of SM3 is similar to that of SHA256. Recently, low power and efficient implementation of the hash algorithm have become an active topic. A software implementation of SHA-3 on the Intel Core-i5 and Cavium Networks Octeon embedded platform is presented in the work of [19]. It has more flexibility and uses less resources, but runs with a lower rate. Instead of commonly used shift registers, a compact hardware implementation of SM3 using SRAM to do message expansion is proposed in article [12]. This method has higher security and speed, but uses more hardware resources and has low flexibility. In article [16], four new different hardware architectures are proposed to improve the performance of SHA-256, reducing the critical path by reordering some operations required at each iteration of the algorithm and computing some values in advance. This work has a significant improvement in performance, but it is not suitable for IoT devices because huge hardware resources are required, and recently, the software/hardware co-design methods are adopted to implement the hash function, which can achieve an excellent trade-off between performance and costs. For instance, in article [13], authors adopt the software/hardware co-design method to implement SM3, which is applied to the financial IC card. Moreover, the different optimized SM3 architecture and methods, such as the 3-stage pipeline approach, the parallel implementation strategy, the optimized architecture adopted the CSA adder, the implementation with the embedded ARM core, etc., are proposed [20–25]. Previous work promotes the development of SM3 VLSI architecture which has the characteristics of small hardware area and low power consumption.

To balance the performance and hardware resources of a cryptographic system for IoT devices, we adopt the software/hardware co-design implementation method to accommodate both the high complexity of computation and flexibility of the algorithm. The optimized algorithm implementation and the AHB-SIC interface IP are also proposed to make the design more efficient and easier-to-use in IoT applications.

In this paper, the followings are made:

- New overall implementation architecture is proposed, which includes the embedded CPU, AHB-SIC, the SM3 circuit module and other modules. The AHB-SIC is designed to easily convert SM3 modules with the non-standard interface into the standard AHB slave interface. The SM3 circuit module is implemented by the software/hardware co-design method to enhance flexibility and reduce the hardware resource.
• Task scheduling and hardware resource optimization methods are applied in the expansion process to reduce the hardware area and power consumption so as to improve overall performance of SM3 implementation. The task scheduling and critical path optimization techniques are also applied in compression module design to reduce time delay and improve efficiency. The identical controller is shared in both expansion and compression modules to simplify the control circuit and reduce the hardware overheads.

• The proposed architecture is implemented on FPGA and ASIC, and also applied to an intelligent gateway. Combined with the other cryptographic modules, our proposed SM3 module can realize the digital signature and identity authentication to protect the security of user data. This framework can also be integrated into other IoT devices.

The rest of this paper is organized as follows: The mathematical background of SM3 is described in Section 2. The system architecture of the design is presented in Section 3. The theoretical analysis and experiment results are shown in Section 4. An example of IoT application of proposed design is given in Section 5. There is a conclusion about this paper in Section 6.

2. SM3 Background

Given the message with length of \( l \) (\( l < 2^{64} \)) bits, the SM3 hash algorithm maps it to produce a message digest of 256 bits. The procedure consists of message padding and parsing, expansion and compression [11].

2.1. Padding and Parsing

Message padding and parsing extend the length of the original message to an integer multiple of 512, and then parse the message into 512-bit blocks.

Suppose the length of message \( m \) is \( l \) bits. First, add the bit ‘1’ to the end of the message, and then add \( k \) bits ‘0’, where \( k \) is the smallest non-negative integer that satisfies \( l + 1 + k = 448 \mod 512 \). Then, add a 64-bit bit string which is binary representation of length \( l \). If \( k = 0 \), we simply need to pad a bit ‘1’ and the binary length \( l \). The length of the padding message \( m' \) is a multiple of 512. After that, the padding message is then parsed into \( n \) 512-bit blocks which are denoted as \( B^{(0)}, B^{(1)}, ..., B^{(n-1)} \). The processes of padding and parsing on disparate lengths of message are shown in Figure 1.

![Figure 1. Padding and parsing on different lengths of messages.](image)

2.2. Expansion

This operation expands each block message \( B^{(i)} \) to generate 132 words \( W_0, W_1, ..., W_{67}, W'_0, W'_1, ..., W'_{63} \) for the compression function (CF) as follows:

• Divide the message \( B^{(i)} \) into 16 words \( W_0, W_1, ..., W_{15} \). The size of each word is 32-bit.

• FOR \( i = 16 \) to 67

\[
W_i = P_1(W_{i-16} \oplus W_{i-9} \oplus (W_{i-3} \ll 15)) \oplus (W_{i-13} \ll 7) \oplus W_{i-6}
\]  

(1)
In the Equation (1), ⊕ and ◯ represent the bitwise XOR and ROL operations, respectively; 
\( P_1(X) = X \oplus (X \ll 15) \oplus (X \ll 23) \) is the permutation function.

- FOR \( i = 0 \) to 63

\[
W'_i = W_i \oplus W_{i+4} \tag{2}
\]

ENDFOR

2.3. Compression

The compression is the core operation of SM3. The grouped message \( m' = B(0), B(1), ..., B(n-1) \), where \( n = (l + k + 65)/512 \). Iterate \( m' \) as follows.

- FOR \( i = 0 \) to \( n - 1 \)

\[
V^{(i+1)} = CF(V^{(i)}, B^{(i)}) \tag{3}
\]

ENDFOR

In the Equation (3), \( V^{(0)} \) is the 256-bit initial value \( IV \), and the result of iterative compression is \( V^{(n)} \). Eight registers denoted as \( A, B, C, D, E, F, G, \) and \( H \) are used in the compression function \( CF \). The size of each register is a word. Let \( SS1, SS2, TT1, \) and \( TT2 \) be the middle variables, and the iteration of the \( CF \) process is described as follows.

\[
ABCDEFGH = V^{(i)}
\]

FOR \( j = 0 \) to 63

\[
\begin{align*}
SS1 &= ((A \ll 12) + E + (T_j \ll j)) \ll 7, \quad SS2 = SS1 \oplus (A \ll 12) \\
TT1 &= FF_j(A, B, C) + D + SS2 + W'_j, \quad TT2 = GG_j(E, F, G) + H + SS1 + W_j \\
D &= C, \quad C = B \ll 9, \quad B = A, \quad A = TT1, \quad H = G, \quad G = F \ll 19, \quad F = E, \quad E = P_0(TT2)
\end{align*}
\]

ENDFOR

\[
V^{(i+1)} = ABCDEFGH \oplus V^{(i)}
\]

The registers \( A \) to \( H \) are initialized with \( IV \) for the first message block, where \( IV = 7380166f, 4914b2b9, 172472d7, dao6000, a96f30bc, 163138aa, e38de4d, b0f80c4e. \) \( \lor, \land \) and \( \neg \) represent the bitwise OR, AND and NOT operations, respectively. The permutation function is represented as \( P_0(X) = X \oplus (X \ll 9) \oplus (X \ll 17) \). The constant \( T_j \) and the Boolean functions \( FF_j \) and \( GG_j \) are given in Table 2.

<table>
<thead>
<tr>
<th>Value Range</th>
<th>( T_j )</th>
<th>( FF_j )</th>
<th>( GG_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; ( j ) ≤ 15</td>
<td>79cc4519</td>
<td>( X \oplus Y \oplus Z )</td>
<td>( X \oplus Y \oplus Z )</td>
</tr>
<tr>
<td>16 &lt; ( j ) ≤ 63</td>
<td>7a879d8a</td>
<td>((X \land Y) \lor (X \land Z) \lor (Y \land Z))</td>
<td>((X \land Y) \lor (\neg X \land Z))</td>
</tr>
</tbody>
</table>

After computing the result of the last group, we obtain \( ABCDEFGH = V^{(n)} \), and the final 256-bit hash value is \( y = ABCDEFGH \).

3. Proposed Architecture

3.1. Overall Implementation Architecture

In this section, overall implementation architecture, which includes the embedded CPU, AHB-SIC, the SM3 circuit module and other modules as shown in Figure 2, is proposed. The embedded CPU is
the master of this IoT SoC in which the software design is implemented. The structure of AHB-SIC is simple and easy to use. The non-standard interface of the cryptographic module can be quickly converted to the AHB slave interface, so as to realize the SoC design effectively. As the slave module of proposed architecture, the SM3 circuit module consists of the expansion module, the compression module, as well as the controller and the result module. In our design, the controller and the data path are separated. The former is chiefly used to control the execution process of the circuit and provide control signals. Composed of several sub-circuit modules, the latter mainly implements SM3 encryption functions, and the result module reads and outputs the result in the form of 32-bit. Besides, the other modules cover RAM, ROM, and other cryptographic modules or interface modules.

![Figure 2. The overall architecture of IoT SoC.](image)

### 3.2. The AHB-SM3 Interface Controller

The data interaction between software and hardware is controlled by the AHB-SIC we proposed. The AHB-SIC is composed of AHB Bus Interface Control Logic (BICL) and four function registers. In our design, the master is the embedded CPU, and the slave is the SM3 circuit modules. Data transfer between the master and the slave is through the AHB-SIC. The BICL is made up of control logic, the address decoder, the data distributor and multiplexer. According to the different types of signals, we design four kinds of function registers (the input register, the output register, the control register and the status register) to realize the data interaction with SM3. The input signals of SM3 consist of write control signal \( W \), read control signal \( R \) and 512-bit input data \( DIN \). The output signals of SM3 incorporate status signal \( STATE \), finish signal \( FINISH \), and 256-bit hash value \( DOUT \). The block diagram of AHB-SIC structure is shown in Figure 3.

![Figure 3. The block diagram of AHB-SIC structure.](image)
The signals of the AHB bus include HADDR, HWDATA, HRDATA, HSEL and so on [26]. The AHB protocol sequence under the basic transmission mode is shown in Figure 4. In the read mode, once the master drives the address and HWRITE signals after the first rising edge, the slave will send data on the second rising edge, and the master will sample the data on the third rising edge. While in the write mode, the master drives the address and HWRITE signals after the first rising edge and outputs the corresponding data on the second rising edge; the slave will sample the address and data on the second and third rising edge, respectively.

![Figure 4. The AHB protocol sequence under the basic transmission mode. (a) Read; (b) Write.](image)

BICL is designed based on AHB bus timing, which mainly implements the data transfer between the master and four sorts of function registers. As seen from the Figure 3, the control logic reads disparate signals on the AHB bus and generates control signals through internal logic to control the address decoder, the data distributor and multiplexer. The four function registers are primarily used for calculation, control, data interaction and the status flag, which can flexibly control the cryptographic hardware modules and obtain their current status for software debugging. Finally, combining a 32-bit low-power embedded CPU, we adopt the AHB-SIC to integrate the SM3 module into an SoC.

### 3.3. SM3 Implementation

#### 3.3.1. Software/Hardware Co-Design

Before the design of the SM3 circuit, we firstly use C/C++ to implement SM3 on the platform of CPU I7-8700k@3.2 GHz. Figure 5 shows the CPU time consuming of each part of SM3, and it is the basic hotspot analysis result which is obtained via using the VTune Amplifier XE tool to list the most active (most time-consuming) functions in our application [27]. As shown in Figure 5, the expansion and compression operations occupy over 90% of the total run time. If we implement the expansion and compression module in software, the encryption efficiency will be very low. Besides, considering the variable length of message, we find it difficult to store whole message, and it will also consume a lot of hardware resources if padding and parsing are designed in hardware. Thus, in our design, the padding and parsing operations are implemented in software, and expansion and compression operations are implemented in hardware. After the software/hardware partitioning, the resulting architecture of the SM3 circuit can be determined.

![Figure 5. The CPU time consuming of each part of SM3.](image)
3.3.2. Controller Design

A standard three-stage Finite State Machine (FSM) structure is adopted in the controller module to generate and provide control signals (\texttt{ctrl}, \texttt{finish} and \texttt{countout} signals) [28]. The three-stage FSM structure can remove the glitch, which is more conducive to optimize the code and facilitate the user to set appropriate timing constraints. The \texttt{ctrl} is the process state signal which contains four states: \textit{idle}, \textit{write}, encryption and \textit{read}.

In the \textit{idle} state, if the input signal \texttt{W} is set to 1, SM3 enters the \textit{write} state and writes data from \texttt{DIN} to the data register. When the data encryption is completed or the last 32-bit data is read, \texttt{finish} will be set to 1. In this case, if \texttt{R} equals to 1, the module enters the \textit{read} state and outputs the result of the current operation in a 32-bit format. It should be noted that each time 512-bit data has been encrypted or once the 256-bit result has been read, the state returns to be \textit{idle}, and the initialization operation is performed to ensure the precision of the data in the next encryption process. If \texttt{R} is equal to 0, the module retains the current operation result and enters the \textit{idle} state to wait for the next block of data. Besides, \texttt{countout} is the output control signal of the counter which is used to control the process of other modules, such as expansion, compression and the entire encryption iteration process. The FSM state transition diagram is shown in Figure 6.

![Figure 6. The FSM state transition diagram.](image)

3.3.3. Expansion Module Design

For the part of expansion, the length of padding message \textit{m}' is assumed to be 512 bits. We analyze the expansion module in the standard method first. Conventionally, the 32 bits divided message \texttt{W}_0, \texttt{W}_1, ..., \texttt{W}_{15} are sequentially stored in 16 registers \texttt{W}_0, \texttt{W}_1, ..., \texttt{W}_{15} in 16 clock cycles. When the rising edge of each cycle comes, a set of data is written into the corresponding register. Starting with the 17th clock, we calculated Equation (1) given in Section 2 when the rising edge of the clock comes, and a new register \texttt{W}_{16} would be set to store the calculated value. That is, the new register \texttt{W}_{16} is set at the 17th clock and its value is calculated. Until the 68th clock, \texttt{W}_{67} is updated. After completing the calculation of Equation (1) given in Section 2, we obtain the 68 words \texttt{W}_0, \texttt{W}_1, ..., \texttt{W}_{67} are obtained, which takes 68 clock cycles. If we calculate \texttt{W}'_j after calculating all the \texttt{W}_i (i = 0, 1, ..., 67) in order, 64 new registers are used to store \texttt{W}'_0, \texttt{W}'_1, ..., \texttt{W}'_{63} and 64 clock cycles are consumed. At this point, a total of 68 + 64 = 132 clock cycles and registers are adopted.

In order to reduce the hardware resources and enhance the performance of the SM3, task scheduling and register optimization are adopted in this work. The proposed design of expansion is shown in Figure 7. In line with the principle analysis of the SM3 algorithm, it can be known that we can get the data required for the second 64-round iterations directly instead of waiting until the first 68-round iterations are completed. It only needs to start computing after the 4th round iteration of the first 68-round iterations. That is to say, the second 64-round iterations can be nested into the first 68-round iterations and begin with its fifth iteration. In this way, merely 68 rounds of iterations or 68 cycles are required. Since the register will bring a lot of areas and power consumption overheads,
in order to further reduce the power consumption, the register multiplexing method is adopted to optimize and reduce the number of registers.

Figure 7. The proposed design of expansion.

For updating $W_i$, we still use the registers $W_0, W_1, ..., W_{15}$ to store the expanding message $m'$, and then set a register $W$ to store the current value $W_i$. The value of $W_i$ is then assigned to $W_{i-1}$ after each round of calculation. For example, after calculating $W_{16}$, the result is directly assigned to $W$. Then, the values of all the registers are assigned to the previous register when $W_{17}$ is computed. The value of $W$ is assigned to $W_{15}$; the value of $W_{15}$ is assigned to $W_{14}$; the value of $W_0$ is discarded. This method can reduce the original 68 registers to merely 17 registers ($W_0, W_1, ..., W_{15}, W$) without affecting the calculation results. Since not all the $W_i$ is necessary to be retained, some intermediate variables are not saved. $Y$ and $P_1$ are directly involved in the calculation as intermediate variables, and do not need to be saved in the register. For updating $W'_j$, it can be seen from the Equation (2) given in Section 2 that the calculation result of $W'_j$ is only related to $W_i$. After the registers are simplified in the previous step, $W_i$ simply keeps the current calculated value and value of 16 adjacent words. Therefore, the calculation of $W'_j$ must be performed when the corresponding $W_i$ is reserved. For example, when the value of $W'_0$ is calculated, $W_0$ and $W_4$ are required. The value of $W_4$ is assigned at the 4th cycle, and in the next cycle, the value of $W_5$ is read, and $W'_1$ can be calculated. At the 15th cycle, the values of $W_0, W_1, ..., W_{15}$ are all read, and the value of $W'_1$ is calculated. Starting from the 16th cycle, we find that $W_i$ begins to store the result in the register of $W$. The calculation of $W'_j$ only needs to use the value of $W_j$ and $W_{j+4}$ calculated by the current clock cycle. In this case, the calculated value ($W_{j+4}$) of the current clock cycle is always stored in the register $W$. Thus, we adopt the register reduction method to obtain $W_j$. Starting from the 16th clock cycle, we find that the values of $W_0, W_1, ..., W_{15}$ will shift to the left every cycle. From the beginning of $W'_12$, each subsequent calculation of $W'_j$ is $W'_j = W_{12} \oplus W (j \geq 12)$.

3.3.4. Compression Module Design

The compression module is the core of SM3, and its main functions are the implementation of the iteration and compression which produce most of the critical path delay. The iterative process of the SM3 covers logical operations such as XOR, ROL, and addition operations (ADD). The delay of XOR or ROL is almost negligible, while the addition operation has a great delay effect [29]. Hence, during
each iteration, the maximum number of serial additions determines the size of delay. We define the
calculation path with the largest number of serial addition operations as the critical path. In order
to reduce the critical path delay and the number of registers, we adopt the task scheduling and critical
path optimization. The proposed design of compression is shown in Figure 8.

As seen from Figure 8, the calculation paths of $A$ and $E$ require five additions to complete. Therefore, the calculation paths of $A$ and $E$ are the critical paths. The total delay of one iteration
is the sum of the delays of five additions. Obviously, the compression process requires a 64-round
loop iteration, which consumes another 64 cycles and the total delay will be large. In our design,
two novel registers $M$ and $N$ are added to reduce one addition operation of $A$ and $E$, and we
let $M_j = D_j + W'_j + 1$, $N_j = H_j + W_j + 1$. Set the initial value of $A$, $B$, $C$, $D$, $E$, $F$, $G$, $H$, $M$, and $N$ to be $A_{-1}$, $B_{-1}$, $C_{-1}$, $D_{-1}$, $E_{-1}$, $F_{-1}$, $G_{-1}$, $H_{-1}$, $M_{-1}$, and $N_{-1}$, respectively. Since the intermediate variable $M$
and $N$ are introduced, the initial addition operations of $M_{-1} = D_{-1} + W'_0$ and $N_{-1} = H_{-1} + W_0$ need to
be implemented before the start of the iterative process. Thus, one clock cycle is consumed before the
compression iteration. It is noteworthy that the calculation of $M_j$ and $N_j$ is parallel with that of $A_j$ and $E_j$ in the critical path. This parallel computing will expedite the compression in SM3. The compression
optimization algorithm is shown in Figure 9.

Through analyzing Equation (4) given in Section 2, we figure out that the calculation of $TT_1$
only needs the value of $W'_j$ in the current clock cycle. Thus, $W'_j$ and $TT_1$ can be simultaneously
calculated. In other words, compression can be nested within the expansion process in advance and it
can start with the sixth iteration of expansion, which finally adds just two more clock cycles. Besides,
by task scheduling optimization, the expansion and compression modules can share the control signals
generated by the same controller, which reduces the complexity and overheads of the control circuit.
The result module mainly realizes the parallel conversion output from the 256-bit to 32-bit format and
controls the conversion process by the counter, which needs 8 cycles; after completing the conversion,
we find that $finish = 1$, indicating that the reading is completed and the next message can be encrypted.
In the end, it takes only 70 cycles to complete an expansion and compression operation.
The compression optimization algorithm

**Input:** IV : The 256-bit initial value of V(0); W0, W1, ..., W67: The 68-round generated words; W0', W1', ..., W63': The 64-round generated words; D1, H1: The initial value of register D, H.

**Output:** The final hash value y.

**Procedure:**
- Initialize the register ABCDEFGH to IV;
- Calculate the initial value of M and N;
- **For** i = 0 to n-1
  - Update the value of V(i) to ABCDEFGH;
- **For** j = 0 to 63
  - Calculate the middle variables through ROL, XOR and Boolean functions;
  - Calculate and update the value of ABCDEFGHMN.
- **Endfor**
- Update the value of ABCDEFGH to V(n);

**Endfor**

The final hash value y = ABCDEFGH.

Figure 9. The compression optimization algorithm.

4. Analysis and the Experiment Result

In this section, we conduct the experiments and analyze our proposed implementation including time and resource consumption analysis, computation amount analysis, analysis of comparison between pure software and other existing work.

4.1. The Setting and Implementation of the Experiment

In the experiment, the proposed architecture of SM3 described above is implemented with the C and Verilog-HDL language. The environment of Vivado 2016 and the platform of Xilinx Virtex 7 xc7vh580 are adopted to implement the architecture of SM3. Besides, the hardware module of SM3 is also synthesized by DC with the SMIC 130 nm technology process. The performance evaluation of SM3 is based on the area, power and throughput.

Firstly, we verify the function of proposed architecture on FPGA. The input message vector is set to be 'abc' which is the published example test case in [11]. The performance, power and the area of this design on FPGA are achieved in Section 4.2.

Secondly, through using the Verilog Compiled Simulator, we can obtain the result data and the waveform to check the validity of this design. The simulation results are shown in Figure 10 and Figure 11, showing the correctness of this design.

![Figure 10. The simulation result of this design.](image-url)
Thirdly, we process the synthesis on DC with SMIC 130 nm technology. The slack, performance, power and the area of this design are acquired and shown in Section 4.2.

In order to further indicate the efficiency of our proposed design, we implement this design by pure software and the software/hardware co-design method on FPGA. Besides, the system clock is normalized to be 36 MHz, and the time consuming is shown in Figure 12.

After conducting the experiment and obtaining the result, we analyze the time and resource consumption, computation amount, comparison with pure software and other existing work to support our design.

4.2. Time and Resource Consumption Analysis

We analyze the time consumption and hardware resource consumption of our proposed design in theory first. Before task scheduling and register optimization, the expansion module requires 132 clock cycles to complete two iterations. The compression module requires a 64-round loop iteration and consumes another 64 cycles so that $132 + 64 = 196$ cycles are consumed. If the strategy of register multiplexing has not been adopted, the expansion module requires 132 32-bit registers to store the 132 intermediate variables, and the compression module requires a total of 16 32-bit registers. Eight registers are applied to store the initial value of $IV$ or the result of the previous 512-bit data block $B^{(i)}$, and the other 8 registers are used to store the currently obtained 256-bit data. Thus, 148 32-bit registers in total are required. After task scheduling and register optimization, only 70 cycles are needed to accomplish the expansion and compression. Besides, the expansion module needs 18 32-bit registers to store 16 32-bit intermediate data and the current two 32-bit data $W_i, W'_i$. The number of registers of the compression module is 18. The 16 32-bit registers are still needed, and the other 2 registers $M, N$ are added to reduce the critical path delay. Therefore, 36 32-bit registers in total are required.

Hence, if the time overhead of other circuits is not considered, the theoretical performance will be improved by nearly 2.8 times, and the number of registers will fall by around 3.11 times. The theoretical area overhead will decrease and eventually the power consumption will also be significantly reduced.

In addition, after critical path optimization, from the DC simulation, we set the clock period to 7 ns and the input/output delay to 2 ns. The data required time is 6.90 ns, and the data arrival time is 5.59 ns. The slack is improved by 1.31 ns. The consumption comparison between the optimized and non-optimized (standard) method is shown in Table 3.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Clock Cycles</th>
<th>Register</th>
<th>Additions of the Critical Path</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-optimized</td>
<td>196</td>
<td>148</td>
<td>5</td>
<td>0.0</td>
</tr>
<tr>
<td>Optimized</td>
<td>70</td>
<td>36</td>
<td>4</td>
<td>1.31</td>
</tr>
</tbody>
</table>

4.3. Computation Amount Analysis

Since the main operations of SM3 are concentrated on the compression function, we provide some theoretical analysis of the computational complexity. In this work, we set LOAD and STORE to represent the operations of the data load and the store, respectively. Before optimization, we need to load and store $W_0$ to $W_{15}$ one time severally. From $W_{16}$ to $W_{67}$, each $W_i$ consumes 5 LOAD, 1 STORE, 6 XOR and 4 ROL; then, we calculate each $W'_i (i = 0, 1, ..., 63)$, and find that 2 LOAD, 1 STORE and
1 XOR are needed. Eventually, each iteration of compression needs 3 LOAD, 12 STORE, 8 ADD, 3 XOR, 8 ROL, 1 \( FF_i \) and 1 \( GG_i \) function. Here the \( FF_i \) function executes 2 XOR \((i = 0, 1, ..., 15)\) or 3 AND and 2 OR \((i = 16, 17, ..., 63)\), and the \( GG_i \) function executes 2 XOR \((i = 0, 1, ..., 15)\) or 2 AND, 1 OR and 1 NOT \((i = 16, 17, ..., 63)\). Based on the above theoretical analysis, the computation amount of a complete compression function is listed in Table 4.

After optimization, since the calculation of \( W'_j \) and \( W_{j+4} \) is simultaneous, only 12 LOAD and 12 STORE are consumed in the iteration of \( W_0 \) to \( W_{15} \). From \( W_{16} \) to \( W_{67} \), each \( W_i \) consumes 5 LOAD, 1 STORE, 6 XOR and 4 ROL; \( W'_j \) is also simultaneously calculated with \( TT_1 \), and there is no extra LOAD and STORE consumption of \( W'_j \). \( T_j \) is a constant so that \((T_j \ll j) \ll 7\) can be calculated and stored in advance. Thus, an optimized compression of each iteration needs 2 LOAD, 2 STORE, 8 ADD, 3 XOR, 5 ROL, 1 \( FF_j \) and 1 \( GG_j \). Table 4 shows that the optimized compression function reduces the times of LOAD, STORE and ROL. In theory, the optimized algorithm can improve the run time by 28.9%.

<table>
<thead>
<tr>
<th>Operation</th>
<th>LOAD</th>
<th>STORE</th>
<th>XOR</th>
<th>LOAD</th>
<th>ADD</th>
<th>AND</th>
<th>OR</th>
<th>NOT</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-optimized</td>
<td>596</td>
<td>900</td>
<td>632</td>
<td>720</td>
<td>512</td>
<td>240</td>
<td>144</td>
<td>48</td>
<td>3792</td>
</tr>
<tr>
<td>Optimized</td>
<td>400</td>
<td>192</td>
<td>632</td>
<td>528</td>
<td>512</td>
<td>240</td>
<td>144</td>
<td>48</td>
<td>2696</td>
</tr>
</tbody>
</table>

4.4. Comparison with Pure Software

In order to test the performance of the SM3 and AHB-SIP, we compare the time consumption between the pure software design of SM3 and our software/hardware co-design of SM3. Let input messages have different lengths, and we conduct six comparative experiments. The time consumption on FPGA under the 36 MHz clock is shown in Figure 12, where the blue line indicates the time required by the pure software method, and the red line indicates the time required by the software/hardware co-design method.

![Figure 12](image-url)  

Figure 12. Time consumption on FPGA.

We can see from Figure 12 that when the message is less than 512 bits, the speed of our method is 6.8 times faster than that of the pure software method. Since padding and parsing operations are implemented by the CPU, and these two steps only need to be done once, the computing efficiency
will be further increased with the length of message increasing. In theory, the computing speed can be 18.7 times faster than that of pure software.

In our experiment, the system clock frequency \( F = 36 \text{ MHz} \), which meets the requirement of low power applications. The throughput of SM3 is calculated by Equation (5):

\[
V_{SM3} = \frac{F \times L}{T}
\]

where \( L \) is the length of each block of the message and \( T \) is the clock period for one round of SM3 encryption. The ASIC implementation results are shown in Table 5. The DC synthesis results reflect that the area of SM3 hardware is 6036 gates and its throughput can be up to 263 Mbps.

4.5. Comparison with Other Related Work

We then compare our proposed architecture with other state-of-the-art hash architecture. We firstly discuss some state-of-the-art techniques. In [20], a SM3 IP design method with the high throughput rate is proposed. The 3-stage pipeline which shortens the critical path is adopted in this method, and the 64 round calculation is expanded to improve the performance. For the implementations reported in [21], the authors propose a parallel implementation strategy to reduce the delay. This method is easy to control, but throughput and efficiency are not very high. In [22], the Keyed-Hash Message Authentication Code and Secure Hash Algorithm 256 (HMAC-SHA256) is implemented, and a Trust-Based system that identifies the malicious nodes and differentiates them from trusted nodes is proposed to detect untrusted nodes. This solution is implemented in software and its throughput is not reported. In the work of [23], two different attacks on SM3 are introduced. The authors mainly studied the improvement of the resistance to attacks, but the internal details of the architecture were not reported, and thus we only discuss and compare the related implementation approaches. In [24], a Carry Save Adder (CSA) was adopted to optimize the critical path, and a dual-channel parallel adder was proposed to achieve higher throughput. In [25], authors combine the implementation with the ARM processor to enhance the throughput. Because of lack of detailed data in [21–23], no direct comparison can be made with the work.

As seen from Table 5, several different implement approaches of the hash algorithm are provided, and the results indicate that the SHA-3 design in [30] only requires 886 gates while the power and throughput are worse than ours. In [20], the proposed implementation has extremely high throughput while the area is also very large. The result shows that the throughput of our design is 1.14 times than that of design in [24] under the normalized frequency. Compared with the compact architecture implementation in [12], the saved areas with our proposed architecture are approximately 37\%. In [31], the shift registers instead of the SRL are used for message expansion since the SRL is not suitable to ASIC. The compact 8-bit SHA-256 architecture in [32] has a smaller area, but the power is higher and throughput is lower compared with our architecture. It is worth noting that, the area of AHB-SIC is only 0.072 mm\(^2\). The results indicate that our proposed implementation achieves an excellent trade-off between performance, the area and power under the normalized frequency.
Table 5. SM3 hardware performance comparison on the ASIC platform.

<table>
<thead>
<tr>
<th>Design</th>
<th>Platforms</th>
<th>Frequency (MHz)</th>
<th>Clock Cycles</th>
<th>Areas (Gates)</th>
<th>Power (mW)</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>0.13-µm</td>
<td>36</td>
<td>70</td>
<td>6036</td>
<td>1.24</td>
<td>263</td>
</tr>
<tr>
<td>SHA-3 [30]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM3 [20]</td>
<td>0.04-µm</td>
<td>28.8</td>
<td>3329</td>
<td>886</td>
<td>4.87</td>
<td>14</td>
</tr>
<tr>
<td>SM3 [24]</td>
<td>0.18-µm</td>
<td>200</td>
<td>1</td>
<td>8800k</td>
<td>-</td>
<td>105k</td>
</tr>
<tr>
<td>SHA-256 [32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHA-3 [30]</td>
<td>0.18-µm</td>
<td>216</td>
<td>68</td>
<td>9458</td>
<td>-</td>
<td>1619</td>
</tr>
<tr>
<td>SHA-256 [32]</td>
<td>0.13-µm</td>
<td>102</td>
<td>1120</td>
<td>9036</td>
<td>3.06</td>
<td>4</td>
</tr>
</tbody>
</table>

The proposed SM3 circuit structure is also implemented on the Vertex 7 FPGA platform, which shows a low power and efficient cryptographic processing performance. As shown in Table 6, there are seven different baselines compared with our proposed architecture. The SHA-256 architecture in [33] uses a three-stage pipeline implementation. The design without masking and with masking consumed 7219 and 10,918 logic cells respectively. The throughput is close to our method under the normalized frequency, but the area is larger than that of our method. In [25], the throughput of this design is only 1.09 times than that of our design while the area is 2.16 times than that of our proposed implementation. The architectures proposed in [31] are divided into the forms of compact and high throughput including the architecture of C-SM3, T-SM3 and Standard-SM3. The result shows that three architectures have a high performance, but 7.53 additional cycles are consumed. There are only 2 additional cycles consumed in our proposed method. In [34], a new SHA-3 architecture is proposed, and it simply consumes 56 slices of the area, but two extra BRAM are adopted, and the throughput of SHA-3 is not very efficient. Thus, compared with the above baseline architectures on FPGA, our proposed architecture also shows an excellent trade-off between performance and the area.

Table 6. SM3 hardware performance comparison on the FPGA platform.

<table>
<thead>
<tr>
<th>Design</th>
<th>Platforms</th>
<th>Frequency (MHz)</th>
<th>Clock Cycles</th>
<th>Areas (Gates)</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>Virtex-7</td>
<td>36</td>
<td>70</td>
<td>808 Slices</td>
<td>263</td>
</tr>
<tr>
<td>SHA-256 [33]</td>
<td>Cyclone II</td>
<td>116.24</td>
<td>68</td>
<td>7219 Logic Cells</td>
<td>875.22</td>
</tr>
<tr>
<td>SHA-256 [33]</td>
<td>Cyclone II</td>
<td>87.08</td>
<td>68</td>
<td>10,918 Logic Cells</td>
<td>655.66</td>
</tr>
<tr>
<td>SM3 [25]</td>
<td>ZYNQ-7020</td>
<td>167.8</td>
<td>64</td>
<td>1743 Slices</td>
<td>1342</td>
</tr>
<tr>
<td>Standard-SM3</td>
<td>Virtex-5</td>
<td>214</td>
<td>7.53</td>
<td>384 Slices</td>
<td>1611</td>
</tr>
<tr>
<td>C-SM3</td>
<td>Virtex-5</td>
<td>215</td>
<td>7.53</td>
<td>324 Slices</td>
<td>1619</td>
</tr>
<tr>
<td>T-SM3 [31]</td>
<td>Virtex-5</td>
<td>362</td>
<td>7.53</td>
<td>328 Slices</td>
<td>2726</td>
</tr>
<tr>
<td>SHA-3 [34]</td>
<td>Virtex-5</td>
<td>372</td>
<td>846</td>
<td>56 Slices + 2 BRAM</td>
<td>225</td>
</tr>
</tbody>
</table>

5. An Example of IoT Application of SM3

Recently, with the rapid growth of IoT devices, the related applications of IoT have been widely researched [35–37]. For the SM3 algorithm, few IoT applications have been proposed. In [13], the design of SM3 reportedly can be applied to the IC card, but it did not provide the implementation in given application. In this section, we give an example of our proposed architecture implementation in the intelligent gateway. Since the costs of different IoT devices and applications are disparate, it is difficult to set a cost evaluation criterion for various scenarios. The cost considered here is that of proposed design of SM3 only. Using the integrated SM3 module, we implemented the digital signature and identity authentication to avoid the attacks and to protect the user data from being tampered with. The IP network topology of the given intelligent security gateway is shown in Figure 13.
In Figure 13, the intelligent security gateway is a KNXnet/IP router. Generally, the security of IoT communication is guaranteed by a special physical medium and the modulation mode, while the IP is the conventional communication medium. When IoT devices use the IP to conduct large-scale networking, security issues are particularly important. The data transferring between gateways and devices is often the plaintext. It is easy to obtain the control commands of IoT devices through network capture, so as to conduct intrusion control. The function of the intelligent security gateway is to connect all IoT devices through the IP and provide secure communication between gateways. A crucial measure to ensure the security in this gateway is embedding the SM3 module. The AHB-SIC is a very convenient data transfer medium of master and slave devices.

In this application, the security chip embedded in the gateway also integrated with the asymmetric cryptographic module, which can be combined with our proposed module to realize the digital signature and verification. Our proposed design mainly achieves the message digest during the process of the signature. As shown in Figure 14, if the device connected with the gateway needs to authenticate the identity, we can launch the SM3 module to generate the message digest as the ciphertext and input the signature module. If there is no need for authentication, we only use the SM3 module to encrypt the plaintext and transfer the data to ensure the communication channel security.

Figure 14 shows the application scenario of this intelligent security gateway. All devices need to be authenticated when we access or communicate with the gateway, and all data is encrypted to guarantee non-repudiation and integrity.
6. Conclusions

The SM3 with efficient hardware and low-power architecture is proposed in this paper. The AHB-SIC is designed to quickly convert the non-standard interface of the cryptographic module into the AHB slave interface. The software/hardware co-design implementation approach is adopted to improve the overall performance and to reduce the hardware consumption. Specifically, the padding and parsing operations are implemented in software, while the expansion and compression operations are implemented in hardware. In the expansion module, task scheduling hardware resource optimization techniques are applied to cut down hardware consumption and to increase overall performance of SM3. In the compression module, the task scheduling and critical path optimization techniques are adopted to reduce delay time and enhance overall performance. Compared with the standard implementation of SM3, the proposed architecture reduces approximately 3.11 times of the total registers with 263 Mbps throughput achieved. This design shows an excellent trade-off between performance and the area when compared with previous work. The proposed architecture can be readily applied to the IoT devices.

Author Contributions: All authors contributed to this work. Investigation, X.Z, J.Z.; Methodology, X.Z, X.H.; Supervision, J.Y., S.C.; Writing-original draft, X.Z.; Writing-review and editing, J.Y., S.C. and X.X.

Funding: This research was supported by Science and Technology Planning Project of Guangdong Province of China (2017B010124003, 2017B090909001 and 2017B090909004.).

Conflicts of Interest: The authors declare no conflict of interest.

References


© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).