Design & Validation of BLAKE 256 IP Core

Jinita Jose*  
Manjusha Maria Alex  
Chippy James  
Nandakumar R  
St. Joseph’s College of Engineering and Technology, Palai  
St. Joseph’s College of Engineering and Technology, Palai  
St. Joseph’s College of Engineering and Technology, Palai  
NIELIT Calicut

Abstract— The amount of sensitive data being transmitted over the internet has been increased due to the rapid development in communication industry. In this context, ensuring security has prime importance. One of the solutions for data security is cryptography. Hash function is an interesting and growing area in cryptography. Secure Hash Algorithms (SHA) is a family of cryptographic hash functions. BLAKE is also a hash function, which is the competitor of SHA 3. In this paper design, implementation and characterization of BLAKE 256 core is presented. Verilog HDL is used to model the hardware. Xilinx® Virtex®-6 FPGA (XC6VLX240T-1FF1156) is used to prototype the design.

Keywords— cryptography, hash functions, secure hash algorithms, SHA 3, BLAKE, IP core

I. INTRODUCTION

Hash functions form an important category of cryptography, which is widely used in a great number of protocols and security mechanisms. It is defined as computationally efficient function, which maps binary strings of arbitrary length to binary strings of fixed length. The last ones are the outputs of a hash computation and they are called hash values. Hash functions are applied to support digital signatures, data integrity, random number generators, authentication schemes, and data integrity mechanisms [1].

Hash functions are a main building block for numerous cryptographic applications. Hash functions are one-way functions and it provides integrity. A family of SHA (Secure Hash Algorithm) algorithms are designed by National Institute of Standards and Technology (NIST) and published as Federal Information of Processing Standards (FIPS). Some of the most widely used dedicated hash functions in real applications are message-digest algorithm MD5 [2] and SHA-1 [3]. Due to a series of attacks on the widely deployed SHA-1 hash function by Wang [4-5], the US National Institute for Standards and Technology (NIST) recommended the replacement of SHA-1 by the SHA-2 hash function family and announced a call for the design of a new SHA-3 hashing algorithm in 2007 [6]. Five candidates, BLAKE [7], Grostl [8], JH [9], Keccak [10] and Skein [11], made it to the third and final round of the competition. Among from the number of proposals KECCAK was selected as the winner of the NIST hash function competition. SHA-3 is not meant to replace SHA-2, as no significant attack on SHA-2[12] has been demonstrated. Because of the successful attacks on MD5 and SHA-0 and theoretical attacks on SHA-1 and SHA-2, NIST perceived a need for an alternative, dissimilar cryptographic hash, which became SHA-3. Like SHA 2, BLAKE also comprises of four variants 224, 256, 384 and 512. The hash function BLAKE-256 operates on 32-bit words and returns a 32-byte hash value. The BLAKE hash function is designed by Aumasson [7], and follows the HAIFA design methodology of Biham and Dunkelman [13].

II. PRINCIPLE OF OPERATION

HAIFA maintains the good properties of the Merkle-Damgard [14-15] construction while adding to the security of the transformation, as well as to the scalability of the transformation. HAIFA has several attractive properties: simplicity, maintaining the collision resistance of the compressions function, increasing the security of iterative hash functions against (second) pre-image attacks, and the prevention of easy-to-use fix-points of the compression function. HAIFA also supports variable hash size and has a built-in support for defining families of hash functions as part of the framework. HAIFA also possesses the online hashing property of the Merkle-Damgard construction. The computation of a HAIFA hash function requires one pass on the message, without keeping the entire message in memory, and while using a fixed amount of memory for the hashing of each block [13].

The main ideas behind HAIFA are the introduction of number of bits that were hashed so far and a salt value into the compression functions. In HAIFA the chaining value $h_i$ is computed as

$$h_i = C(h_{i-1}, m_i, #bits, salt)$$

where #bits is the number of bits hashed so far and salt is a salt value. Thus, to hash a message $M$ using $C(\cdot)$ and the salt and obtaining $m$ bits of digest value (as long as $m \leq m_s$), the following operations are performed:

1. Pad $M$ according to the padding scheme
2. Compute $IV_m$ the initial value for a digest of size $m$
3. Iteratively digest the padded message using $C(\cdot)$, starting from the initial value $IV_m$ and using the salt. In that case an additional block is padded to the message, the compression function is called on this block with $#bits = 0$.
4. Truncate the final chaining value if needed
The hash function BLAKE-256 operates on 32-bit words and returns a 32-byte hash value. It uses eight 32-bit same initial values which is used in SHA 256. BLAKE-256 uses 16 constants \{c0, c1...c15\}. Ten permutations of \{0,1,...,15\} are used by all BLAKE functions [7]. For hashing, a message \(m\) of bit length \(\ell < 2^{64}\) is taken as input. The message is first padded then it is processed block per block by the compression function.

- **Padding**

  First the message is extended so that its length is congruent to 447 modulo 512. Length extension is performed by appending a bit 1 followed by a sufficient number of 0 bits. At least one bit and at most 512 are appended. Then a bit 1 is added, followed by a 64-bit unsigned big-endian representation of \(\ell\). Padding can be represented as

  \[ m \leftarrow m|1000...0001<\ell>_{64} \]

  This procedure guarantees that the bit length of the padded message is a multiple of 512.

- **Iterated hash**

  To proceed to the iterated hash, the padded message is split into 16-word blocks \(m_0, m_1, ... , m_{15}\). We let \(l_i\) be the number of message bits in \(m_0, m_1, ... , m_i\), that is, excluding the bits added by the padding. The salt \(s\) is chosen by the user, and set to the null value (i.e. \(s_0 = s_1 = s_2 = s_3 = 0\)). The hash of the padded message \(m\) is computed as follows:

  \[
  h_0 \leftarrow \text{IV} \\
  \text{for } i=0,...,N-1 \\
  h_{i+1} \leftarrow \text{compress}(h_i, m_i, s, l_i) \\
  \text{return } h_N
  \]

  The compression function of BLAKE-256 takes as input four values:

  - chain value \(h = h_0, \ldots, h_7\)
  - message block \(m = m_0, \ldots, m_{15}\)
  - salt \(s = s_0, \ldots, s_3\)
  - counter \(t = t_0, t_1\)

  These four inputs represent 30 words in total (i.e., 120 bytes = 960 bits). The output of the function is a new chain value \(h' = h'_0, h'_1, \ldots, h'_7\) of eight words (i.e., 32 bytes = 256 bits). The compression of \(h, m, s, t\) to \(h'\) is written as \(h' = \text{compress}(h, m, s, t)\).

  ➤ **Initialization**

  A 16-word state \(v_0, v_1, ..., v_{15}\) is initialized such that different inputs produce different initial states. The state is represented as a 4×4 matrix, and filled as follows:

  \[
  \begin{pmatrix}
  v_0 & v_1 & v_2 & v_3 \\
  v_4 & v_5 & v_6 & v_7 \\
  v_8 & v_9 & v_{10} & v_{11} \\
  v_{12} & v_{13} & v_{14} & v_{15}
  \end{pmatrix}
  \begin{pmatrix}
  h_0 & h_1 & h_2 & h_3 \\
  h_4 & h_5 & h_6 & h_7 \\
  s_0 & c_0 & s_1 & c_1 \\
  s_2 & c_2 & s_3 & c_3 \\
  t_0 & c_4 & t_1 & c_5 \\
  t_2 & c_6 & t_3 & c_7
  \end{pmatrix}
  \]

  ➤ **Round function**

  Once the state \(v\) is initialized, the compression function iterates a series of 14 rounds. A round is a transformation of the state \(v\) that computes

  \[
  \begin{align*}
  G_0(v_0, v_4, v_8, v_{12}) & \quad G_1(v_1, v_5, v_9, v_{13}) \\
  G_2(v_2, v_6, v_{10}, v_{14}) & \quad G_3(v_3, v_7, v_{11}, v_{15}) \\
  G_4(v_0, v_5, v_{10}, v_{15}) & \quad G_5(v_1, v_6, v_{11}, v_{12}) \\
  G_6(v_2, v_7, v_8, v_{13}) & \quad G_7(v_3, v_4, v_9, v_{14})
  \end{align*}
  \]
where, at round $r$, $G_i(a, b, c, d)$ sets

\[
\begin{align*}
    a &\leftarrow a + b + (m_{\sigma_r}(2i) \oplus c_{\sigma_r}(2i+1)) \\
    d &\leftarrow (d \oplus a) \gg 16 \\
    c &\leftarrow c + d \\
    b &\leftarrow (b \oplus c) \gg 12 \\
    a &\leftarrow a + b + (m_{\sigma_r}(2i+1) \oplus c_{\sigma_r}(2i)) \\
    d &\leftarrow (d \oplus a) \gg 8 \\
    c &\leftarrow c + d \\
    b &\leftarrow (b \oplus c) \gg 7
\end{align*}
\]

The first four calls $G_0, G_1, G_2, \text{and } G_3$ can be computed in parallel, because each of them updates a distinct column of the matrix. The procedure of computing $G_0, G_1, G_2, \text{and } G_3$ is called as a column step. Similarly, the last four calls $G_4, G_5, G_6, \text{and } G_7$ update distinct diagonals thus can be parallelized as well, which is called as a diagonal step. At round $r > 9$, the permutation used is $\sigma_r \mod 10$.

- Finalization

After the rounds sequence, the new chain value $h'_0, h'_1, \ldots, h'_7$ is extracted from the state $v_0, v_1, \ldots, v_{15}$ with input of the initial chain value $h_0, h_1, \ldots, h_7$ and the salt $s_0, s_1, \ldots, s_3$:

\[
\begin{align*}
    h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
    h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
    h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
    h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
    h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
    h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
    h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
    h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{align*}
\]

III. PROPOSED DESIGN

A. Core I/O Diagram

![Core I/O Diagram for BLAKE 256](image-url)
B. Pin Table

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Mode</th>
<th>Type</th>
<th>Width</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>IN</td>
<td>Active high</td>
<td>1</td>
<td>Master clock</td>
</tr>
<tr>
<td>rst</td>
<td>IN</td>
<td>Active high</td>
<td>1</td>
<td>Master reset</td>
</tr>
<tr>
<td>wr</td>
<td>IN</td>
<td>Active low</td>
<td>1</td>
<td>Control signal to read the available data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin Name</th>
<th>Mode</th>
<th>Type</th>
<th>Width</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg_in</td>
<td>IN</td>
<td></td>
<td>8</td>
<td>Message data is input as 8-bit blocks through this port when ‘wr’ is low. After receiving each block (8-bit), ‘wr’ is asserted.</td>
</tr>
<tr>
<td>islast</td>
<td>IN</td>
<td>Active high</td>
<td>1</td>
<td>After the entire block of data is served, islast is asserted and then no input data can enter</td>
</tr>
</tbody>
</table>

H OUT 256 Message digest

IV. IMPLEMENTATION RESULT

The BLAKE 256 algorithm is simulated using Modelsim® SE and the obtained result is shown in Fig. 2. For software-hardware co-validation ‘00’ is taken for BLAKE 256.

![Fig. 2: Simulation result using Modelsim® BLAKE 256](image)

The designs are implemented on a Virtex®-6 FPGA (XC6VLX240T-1FF1156) included in the Xilinx® ML605 evaluation kit. Xilinx® ISE 14.3 is used as an FPGA development environment during the implementation process (i.e., synthesis, map and place & route). Table II below shows the overall resource consumption. Fig. 3 show the hardware test result using ChipScope™ ILA tool of Xilinx®.
V. CONCLUSION

FPGA implementation of BLAKE 256 is presented in this paper. The behavioural descriptions of the designs are written in verilog HDL and simulated using Modelsim® platform. The designs are successfully implemented on Xilinx® Virtex®-6 FPGA (XC6VLX240T-1FF1156). Design verification is performed with the help of ChipScope™ ILA tool of Xilinx®.

REFERENCES


