TECHNISCHE UNIVERSITÄT WIEN
Vienna | Austria

Institut für Computertechnik
Institute of Computer Technology

Benedikt TUTZER
Dinka MILOVANCEV

Digital Integrated Circuits Lab (LDIS)

384.088, Summer Term 2018

Supervisors:

Christian Krieg, Martin Mosbeck, Axel Jantsch

BLAKE2 cryptographic hash and message authentication code (MAC)

An implementation in VHDL

Abstract

The task of our group was to implement *BLAKE2b* hash function as specified in [2]. The *BLAKE2b* algorithm computation was implemented using state machines, the implementation was syntactically correct and synthesizable. The functional correctness was verified by using the reference implementation given in C. The test bench compared the output of our entity with the reference hash value for the same message input and reported the result in terminal. For the message input we used the random data generated by the Task 1 implementation. The *BLAKE2b* hash function entity is to be used as a component of *Argon2* memory hard function which should generate cryptographically secure keys from passwords as specified in [1]. The *Argon2* function should be implemented targeting *Nexyx 4 DDR* board.

1 Overview

Include a block diagram of the top-level entities here!

The *BLAKE2* hash function comes in two variants: *BLAKE2b* for 64-bit platforms and *BLAKE2s* for smaller architectures. We implemented both *BLAKE2b* and *BLAKE2s* based on RFC 7693 [2]. The input to *BLAKE2* is the message data (message length goes from 1 to 2^{128} bytes) and the output is the message digest or simply hash value (hash length goes from 1 to 64 bytes). According to the reference implementation [**REF**], the input message is divided into N 128 byte message blocks m_i; if a secret key is used, it is used as the first message block. In our design specification, the secret key is not used. Is it possible to enable key usage? This would be very cool, because then this core can be used to create message authentication codes (MACs).

The hash value is iteratively computed as in the following pseudo-code:

ALGORITHM 1: *BLAKE2b* algorithm

$h^0 = IV$: Initialize the hash state vector with initialization vector IV. Obtain the IV from the first 64 bits of the fractional parts of the first eight prime numbers' square roots. For creating a MAC, use a secret key instead.

for $i := 0$ ***to*** $N - 2$ **do**

$h^{i+1} = \text{compress}(h_i, m_i, t_i, f = \text{'FALSE'})$

The compression function completely compresses one data block; it takes as an input the previous hash state, the current message block m_i (divided into 16 words with length $w = 64$ bits), $2 \cdot w$ bits wide offset counter t_i that counts how many bytes have been compressed, and flag indicator f for the last message block. The input message block is mixed into the current hash states.

end

$h^N = (h_{N-1}, m_{N-1}, t_{N-1}, f = \text{'TRUE'})$

Compute the final message block. The output is the first $hash\_len$ bytes of little-endian state vector h. The input hash length parameter $hash\_len$ is in the range of 1 to 64 bytes. This sentence is not quite clear to me... Is the information regarding $hash\_len$ relevant here?

return h^N

The main challenge in implementing the algorithm is the compression function which is called for each message block. The message block is mixed in 12 rounds, and in each round, the message word schedule is defined by 10 possible permutations $\sigma_0 \ldots \sigma_9$ (hard coded into design as two dimensional *SIGMA* array). Mixing the messages requires an additional mixing function that mixes two 64-bit words from message m_i into hash state h_i. We an auxiliary local 4×4 working vector $v[0..15]$ in the mixing function:

$$v = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix}$$

Hash functions are used in various security protocols to ensure the integrity of transmitted data. Transmitted data is mapped to a hash value h of fixed size. It should be computationally impossible that two sets of data result in the same hash value. Also, a small change in message data must result in a large change in h, and it should be computationally impossible to reconstruct the input data from the hash value. We verify the quality of our *BLAKE2b* implementation by comparing the outputs to thte outputs produced by the reference implementation, available as C code in RFC 7693

[2, pp. 16-26].

2 Implementation

The *BLAKE2b* hash algorithm is split into eight main operations which are implemented as a state machine in very-high-speed integrated circuits hardware description language (VHDL). The state machine implements the following states (and state transitions):

1. STATE_IDLE

   Initialize the *BLAKE2b* hash function. Set initial values for the hash state vector $h[0..7]$ (array of 8 64-bit values). If used as a hash function, use the pre-defined set of prime numbers' square roots. If used as a MAC, use a secret key instead, which is provided as input to the function. If there is a new message block, set *valid_in* to 'HIGH'. The next state is STATE_PREPARE. The counter which counts the number of compressed bytes (*compressed_bytes*) is incremented by the double base width (which is 128 for BLAKE2b). For the last message block, the *compressed_bytes* counter is set to the length of the message.

2. STATE_PREPARE

   Initialize the compress function. Local state vector $v[0..15]$ is initialized with hash state vector $h[0..7]$, mixed with the number of received bytes. The counter which counts the number of mixing rounds *ci_done* is reset to zero (there should be 10 or 12 rounds mixing a message). The next state is STATE_COMPRESS.

3. STATE_COMPRESS

   Resets the counter for the mixing function *mi_done* to zero (its maximum value is 7). The next state is the first mixing state, STATE_MIX_A. At the final mixing state (which one is it? This is not quite clear to me...), all the columns and diagonals of the working vector $v[0..15]$ are mixed with words of the current message block.

4. STATE_MIX_A

   The first mixing state is one part of mixing function G specified in RFC 7693 [2, p. 7]. G requires word additions (Does it mean to add words, or to perform the arithmetic operation 'addition' on words?). In each cycle, one value of the working vector $v[0..15]$ is computed depending on the *SIGMA* permutation constant, the current mixing round, and the 2-bit wide tag *mio_left* which serves to internally codify the mixing equations. The next mixing state is STATE_MIX_B.

5. STATE_MIX_B

   The second mixing state implements the second part of mixing function G, and requires XOR operations and bit shifting. Again, in each cycle one value of the working vector $v[0..15]$ is computed depending on the *SIGMA* permutation constant, the current mixing round, and internal the 2-bit wide tag *mio_left* to internally codify the mixing equations.

   mio_left is updated after each operation in state STATE_MIX_B. If there are still pending mixing operations in a given round (*mi_done* $\neq 7$), then the next state is STATE_MIX_A.

   In total, there are eight mixing operations (*mio_done*) divided into states A and B (Hmmm, the term "state" seems to be used ambiguously here), and four operation codes for AB pairs. When all the codes are computed in a given order, the counter which tracks the mixing code, *mio_left*, is incremented. If the last mixing operation is done but there are still mixing

rounds pending, the next state is STATE_COMPRESS, and the counter that keeps track of the computed rounds, *ci_done*, is updated.

If all the mixing rounds are computed (*ci_done* $= 11$), and the last message bit is sent ($seen\_last$ is 'HIGH') the next state is STATE_DONE.

If the last block is not yet sent but all the mixing rounds are already computed, the next state is STATE_MIX_H.

6. STATE_MIX_H

   Mix the upper half $v[0..7]$ and the lower half $v[8..15]$ of the working vector $v[0..15]$ into the current state vector $h[0..7]$. The next state is STATE_WAIT to compress the next message block.

7. STATE_WAIT

   Expect the next message block. If *valid_in* goes 'HIGH', the system variables are updated and the next state is STATE_PREPARE.

8. STATE_DONE

   When the last message block has been compressed, the hash output is computed. The next state is STATE_IDLE in which we wait for the next message input.

The state machine described above is implemented in *blake2.vhd*.

3 Verification

In order to verify that the *BLAKE2b* specification was correctly implementedd, we designed a test bench that reads input from a file *messages.txt* line by line. Each line is hashed by the implemented algorithm. The same input data is being provided the reference implementation from RFC 7693 [2], which calculates the corresponding hash values. The reference values are stored in the file *hashes.txt*. The test bench compares the output created by the design under test to the reference values created by the reference implementation.

NOTE: In order to read the hash values from *hashes.txt*, we need to convert the values from hexadecimal to *std_logic_vector*. We used the *ASCII_2_STD* function.

Figure 1 shows the situation where a message block is fully compressed and a new message block is ready to be compressed (i.e., the round counter *ci_done* is '11', and the state is STATE_WAIT). The BLAKE2 core signals this situation by setting *valid_in* to 'HIGH' for one clock cycle. Therefore, the next message block is written to *current_chunk*, and the next state is STATE_PREPARE. The message block register *message* is reset to zero when *valid_in* is set to 'LOW', and the state machine enters STATE_COMPRESS. Then *mi_done* is reset to zero, and the BALKE2 core performs the mixing function, visible in the *state* variable (STATE_MIX_A, STATE_MIX_B). After the core performs the mixing, it updates the mixing code signal *mio_left*. Compressing of one 1024-bit message block consumes 835 clock cycles.

For the message data input we used the hexadecimal random data from Task 1 as 1032 byte input message. Additional message was created by replacing only the first character with '1' in order to verify that even such a small change in input can lead to entirely different hash values. This can be seen in Figure 2. These hash values matched the hash values from the C code implementation.

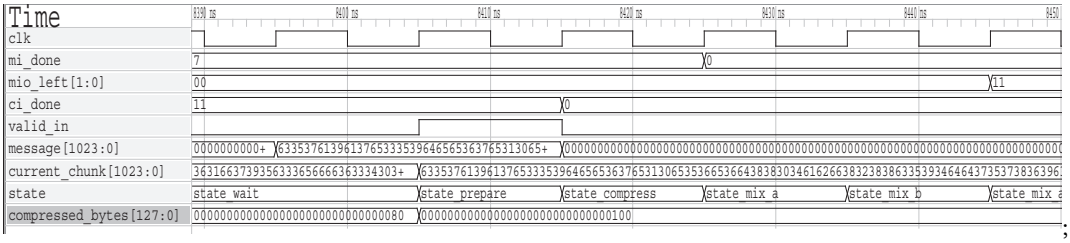

Figure 1: Simulation results for blake2b hash function: state transitions

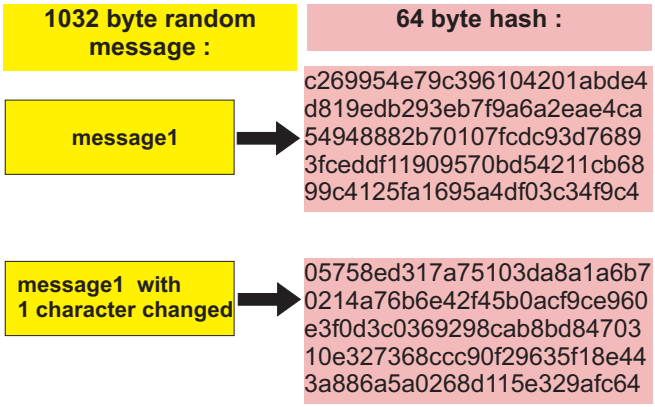

Figure 2: Similar input data results in a very different hash messages

Since this design is to be used as a module of the larger *Argon2* design, there were no ports to be mapped to the hardware and therefore placing and routing of the design was not possible, but the RTL synthesis and optimization was successfully completed. The RTL optimization report is shown in Listing 1.

Listing 1: RTL Hierarchical Component Statistics

```
-
Hierarchical RTL Component report
Module blake2b
Detailed RTL Component Info :
+---Adders :
            2 Input    128 Bit       Adders := 1
            3 Input     64 Bit       Adders := 1
            2 Input      4 Bit       Adders := 1
            2 Input      3 Bit       Adders := 1
            2 Input      2 Bit       Adders := 1
+---XORs :
            2 Input     64 Bit         XORs := 5
            3 Input     64 Bit         XORs := 8
+---Registers :
                       1024 Bit    Registers := 1
                        512 Bit    Registers := 1
                        128 Bit    Registers := 1
                         64 Bit    Registers := 24
                         11 Bit    Registers := 1
                          4 Bit    Registers := 1
                          3 Bit    Registers := 2
                          2 Bit    Registers := 1
                          1 Bit    Registers := 3
+---Muxes :
            8 Input    128 Bit        Muxes := 1
            2 Input    128 Bit        Muxes := 2
            3 Input     64 Bit        Muxes := 3
            8 Input     64 Bit        Muxes := 23
            4 Input     64 Bit        Muxes := 1
            9 Input     64 Bit        Muxes := 1
            8 Input      4 Bit        Muxes := 11
            8 Input      3 Bit        Muxes := 11
            2 Input      3 Bit        Muxes := 1
            3 Input      3 Bit        Muxes := 1
            8 Input      1 Bit        Muxes := 17
           11 Input      1 Bit        Muxes := 12
            4 Input      1 Bit        Muxes := 16
            2 Input      1 Bit        Muxes := 27
---------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------
```

4 Discussion

In the reference paper [2] for *BLAKE2b* hash function coded in C, the input parameter was the whole message that can have between 1 and 2^{128} bytes. This long message was intended to be divided into 128-byte message blocks during computing. Since the VHDL cannot support such large input vectors we decided to send the message block by block. In this way our data input is 128 bytes long and we have additional information about the message length ($message\_len$). The maximum message length was specified to be 1032 bytes since this is the maximum length needed by *Argon2*. For every message block we need the information of the main module whether there is a new message block available (signal $valid\_in$ goes high) and the flag register for the last message block $last\_chunk$ (high if the last message block is sent). As the output we provide handshaking signals $compress\_ready$ and $valid\_out$, the user of our entity must make sure that $compress\_ready$ is high before sending a new message block, and the output i.e. the hash can be stored when $valid\_out$ is high.

The input messages in `messages.txt` file can be empty message. However messages are not allowed to contain whitespaces.

5 Conclusions

In this task we implemented *BLAKE2b* hash algorithm in the hardware. The design was synthesizable, however the final utilization report can be done once the entity module is used inside the *Argon2* implementation, when it will be mapped to the target hardware (*Nexys 4 DDR*).

The functional verification of our design was performed through simulation. The random data was used as the message input and the reference implementation in C was used to validate that the correct hash output has been computed.

References

[1] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson. *The memory-hard Argon2 password hash and proof-of-work function*. Internet-Draft draft-irtf-cfrg-argon2-03. Work in Progress. Internet Engineering Task Force, Aug. 2017. 44 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2-03.

[2] M.-J. O. Saarinen and J.-P. Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Nov. 2015. DOI: 10.17487/RFC7693. URL: https://rfc-editor.org/rfc/rfc7693.txt.