TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

Benedikt TUTZER
Dinka MILOVANCEV

Course on

Digital Integrated Circuits Design

384.088, Summer Term 2018

Supervisors:

Christian Krieg, Martin Mosbeck, Axel Jantsch

June 13, 2018

BLAKE2 cryptographic hash and message authentication code (MAC)

An implementation in VHDL

Abstract

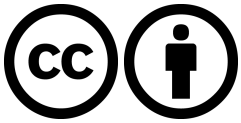
We implemented *BLAKE2b* and *BLAKE2s* hash function as specified in RFC 7693. We used state machines to implement *BLAKE2*. Functional correctness is verified by comparing the produced output to the reference implementation given in C. Our test bench compares the output of our entity with the reference hash value for the same message input and reports the result to stdout.

This work was originally developed in the course "Digital Integrated Circuits Design" held at the Institute of Computer Technology (ICT), Vienna University of Technology (TU Wien). If you find this work useful, please cite it using the following BibTeX entry:

```
@TechReport{Krieg2018,
  author      = {Christian Krieg and Benedikt Tutzer and Dinka Milovancev},
  title       = {BLAKE2 cryptographic hash and message authentication code (MAC) -- An
    implementation in VHDL},
  institution = {Institute of Computer Technology, Vienna University of Technology (TU Wien)},
  year        = {2018},
  type        = {techreport},
  address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
  month       = {June},
}
```

Contact us:

christian.krieg@alumni.tuwien.ac.at



1 Usage

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 [1]. Listing 1 shows how the BLAKE2s core is instantiated. Instantiating the BLAKE2b core is similar, just replace "blake2s" by "blake2b".

To create hashes from an arbitrarliy long message up to 2^{128} bytes, split the message into chunks of *BLOCK_SIZE* bytes. Apply them to the *message* port sequentially. Once the message is stable, raise *valid_in* for one clock cycle.

If a key shall be used, it has be be sent as first chunk, padded with zeroes. The length of the key needs to be available at the *key_len* port.

message_len needs to be set to the number of bytes that are to be hashed in total. This includes the additional chunk containing the key, so when a key is used, the size of one complete chunk needs to be added to *message_len* (128 for BLAKE2b or 64 for BLAKE2s respectively).

The length of the hash can be chosen by setting *hash_len*.

After sending one chunk, wait for *compress_ready* to be 'HIGH' before sending the next chunk. When the last chunk is sent, the input *last_chunk* needs to be set to 'HIGH'. After the last chunk is encoded, the output *valid_out* is raised and the hash is available at the *hash* output.

Listing 1: Usage example for the BLAKE2s core. Instantiating a BLAKE2b is similar.

```
architecture behav of example is
    component blake2s is
        port (
            reset          : in  std_logic;
            clk            : in  std_logic;
            message        : in  std_logic_vector(64 * 8 - 1 downto 0);
            hash_len       : in  integer range 1 to 32;
            valid_in       : in  std_logic;
            message_len    : in  integer range 0 to 2147483647;
            compress_ready : out std_logic;
            last_chunk     : in  std_logic;
            valid_out      : out std_logic;
            hash           : out std_logic_vector(32 * 8 - 1 downto 0)
        );
    end component;
begin
    hash : blake2s
    port map (
        reset          => reset,
        clk            => clk,
        message        => message,
        valid_in       => valid_in,
        message_len    => message_len,
        hash_len       => hash_len,
        compress_ready => compress_ready,
        last_chunk     => last_chunk,
        valid_out      => valid_out,
        hash           => hash
    );
...
end behav;
```

2 User Interface

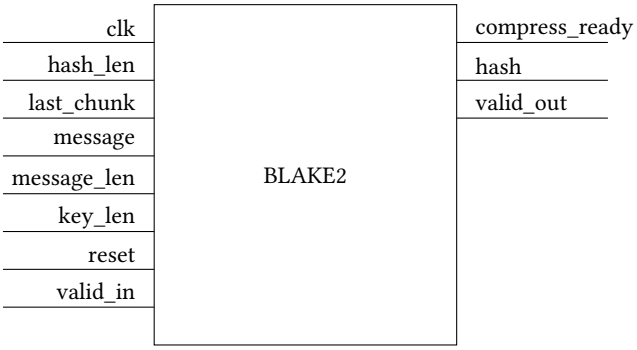


Figure 1: BLAKE2 entity

The functionality of BLAKE2s and BLAKE2b is encapsulated such that it is transparent to the user to instantiate these cores. The ports to the BLAKE2s and BLAKE2b cores are illustrated in Figure 1. Table 1 lists the ports with additional information.

Table 1: Ports for the BLAKE2 cores

Port	**Input/Output**	**Default value** (BLAKE2s/BLAKE2b)	**Description**
clk	Input	—	Clock signal
hash_len	Input	(32/64)	Length of hash in bytes
last_chunk	Input	—	Signal to indicate that last message chunk is hashed
message	Input	—	Input message to be hashed
message_len	Input	max. 2147483647	Total length of input message in bytes
key_len	Input	max. 128/64	Length of the key
reset	Input	—	Reset signal
valid_in	Input	—	When 'HIGH', data on the *message* input is processed
hash	Output	—	The generated hash value
compress_ready	Output	—	When 'HIGH', new data can be provided to the *message* input
valid_out	Output	—	When 'HIGH', the data at the *hash* output is valid

Internally, generics are used to switch between BLAKE2s and BLAKE2b functionality. Table 2 lists these generics along with value range and default valuesfor BLAKE2s and BLAKE2b.

Table 2: Generics for the BLAKE2 cores

Name	**Range**	**Default value** (BLAKE2s/BLAKE2b)	**Description**
BASE_WIDTH	32–64	32/64	Width of the internal state vectors
COMP_ROUNDS	1–64	10/12	Number of compression rounds to be used
BLOCK_SIZE	1–512	64/128	Size of each message chunk
MAX_HASH_LENGTH	1–2147483647	32/64	Maximum length of the hash
MAX_MESSAGE_LENGTH	2147483647	2147483647	Maximum length of input messages

3 Functional description

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 [1]. Figure 1 shows a block diagram of the BLAKE2 entity. 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 [1, pp. 16-26], 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. The hash value is iteratively computed as shown in Algorithm 1.

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 message authentication code (MAC), use a secret key instead.

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

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

$h^N = (h_{N-1}, m_{N-1}, t_{N-1}, f =$ '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.

return h^N

The main challenge in implementing the algorithm is the compression function which is applied to each message block. The message block is mixed in 10–12 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 use 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 [1, pp. 16-26].

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 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 , 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 [1, p. 7]. G requires word additions . 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 , 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*.

4 Functional verification

In order to verify that the *BLAKE2b* specification was correctly implemented, 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 provided by the reference implementation in RFC 7693 [1], 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*, wherefore we define the function *ASCII_2_STD* in the testbenches.

Figure 2 shows the situation where a message block has been 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.

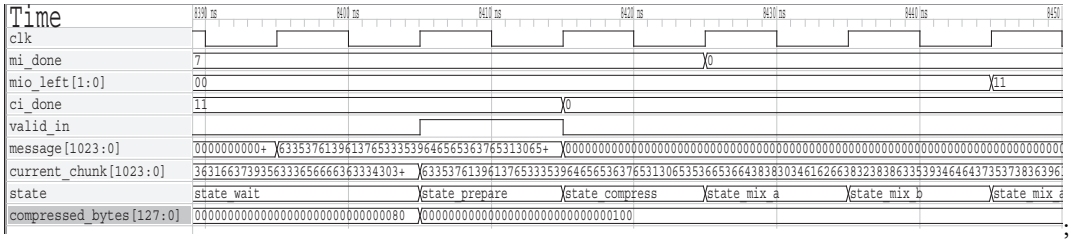


Figure 2: Simulation results for BLAKE2b hash function

Table 3: Example hashes and corresponding input messages for BLAKE2b

Input message	**Hash value**	**Hash length**
`Message`	`24`	1 byte
`Message`	`2120`	2 bytes
`Message`	`0d93575d`	4 bytes
`Message`	`184c2242 418d7bc6`	8 bytes
`Message`	`ea30fef5 179fa367 0595e817 933e0ce8`	16 bytes
`Message`	`8a5efe6b 0c5a8d56 304804e8 aecfae16` `2e3799dc 92ad1942 b4bd72c5 2a97f3bd`	32 bytes

`Message`	`547ee071 8a19d53f df9f6123 8f31c139`
	`1979e95a 191505ba aa821805 1c4702d6`
	`f95d4312 d5c268f0 c6b4aa98 787dd91e`
	`3296ea3c d6d94833 fdfd5bae 39656b3c`
		64 bytes

`Massage`	`566cb134 f53e14f2 7b49fcf5 45f4c71c`
	`4486c824 c0fe5359 c61df882 d610c38e`
	`88b25a1a 3e5790c4 b0099c9f 55f0ebc6`
	`71cf2aec 4a9b9134 d3f05b9e 8eee1612`
		64 bytes

`abc`	`0981401f 1f3003cb 92666dc6 a71e9fc0`
	`4c212f14 685ac4b7 4b12bb6f dbffa2d1`
	`7d87c539 2aab792d c252d5de 4533cc95`
	`18d38aa8 dbf1925a b92386ed d4009923`
		64 bytes

Table 3 shows two example hashes for BLAKE2b. In order to show that a small change in the message results in a large change in the hash, we replaced one character of the original message ("`Message`" vs. "`Massage`"). Table 4 shows the hash values for the same input messages hashed with BLAKE2s.

Listing 2 shows the register-transfer level (RTL) optimization report.

Table 4: Example hashes and corresponding input messages for BLAKE2s

Input message	Hash value	Hash length
Message	fb	1 byte
Message	46ce	2 bytes
Message	57c3d825	4 bytes
Message	4a690e84 7edb95d8	8 bytes
Message	6e5cca08 1ec78d57 2b50cd1e efd754d1	16 bytes
Message	94bb1d33 b1ae1a65 aa1dad9b ade6c30b 14f6ba03 0b6ab5e5 c9756aba 26b77219	32 bytes
Massage	f153acb5 47c9d8a3 199c4820 3d488df4 82cc5a21 e760251a b7f090b7 7bdf8b5f	32 bytes
abc	508c5e8c 327c14e2 e1a72ba3 4eeb452f 37458b20 9ed63a29 4d999b4c 86675982	32 bytes

5 Resource allocation

Listing 2: 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
---------------------------------------------
```

6 Additional notes

For the reference implementation given in RFC 7693 [1], the input parameter is the whole message whose length can have between 1 and 2^{128} bytes. This long message is intended to be divided into 128-byte message blocks during hash computation. Since VHDL does not support such large input vectors, we decided to send the message block by block. This way, our data input is 128 bytes long, and we introduced additional information regarding the message length (*message_len* input). 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). For the output we provide handshaking signals *compress_ready* and $valid\_out$, the user of our package must make sure that *compress_ready* is high before sending a new message block, and the output 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 whitespace.

7 Concluding remarks

We implemented *BLAKE2b* and *BLAKE2s* hash algorithms in VHDL. The design is fully synthesizable. We functionally verified our design by simulation. We used the reference implementation given in C in the original specification of BLAKE2 [1] to verify that we correctly implemented the algorithm.

References

[1] 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.