Points	Grade

Team: 15

01429540 Dinka MILOVANCEV #1
01225461 Benedikt TUTZER #2

Digital Integrated Circuits Lab (LDIS)

384.088, Summer Term 2018

Supervisors:
Christian Krieg, Martin Mosbeck, Axel Jantsch

Task 2: Blake2b cryptographic hash function

Abstract

The task of our group was to implement *BLAKE2b* hash function as specified in.[1] 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.[2] The *Argon2* function should be implemented targeting *Nexyx 4 DDR* board.

[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.

[2] A. Biryukov et al. *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.

1 Problem statement and motivation

BLAKE2 hash function comes in two variants: *BLAKE2b* for 64-bit platforms and *BLAKE2s* for smaller architectures. *BLAKE2b* hash function was implemented in this task based on;[3] the function is intended to be used in *Argon2* memory-hard function for password hashing[4] which is to be implemented in hardware targeting the *Nexys 4 DDR* board. The final output of the *Argon2* should be cryptographically secure key derived from the input password.

The input of the *BLAKE2b* hash function 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 algorithm the input message is divided into N 128 byte message blocks m_i; if the secret key is used then it is set as the first message block. In our design specification the secret key is not being used.

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

$h^0 = IV$: initialization of the hash state vector with initialization vector IV (obtained by taking the first 64 bits of the fractional parts of the square roots of the first eight prime numbers)

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 previous hash states, current message block m_i (divided into 16 words with length w =64 bit), 2·w 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

$h^N = (h_{N-1}, m_{N-1}, t_{N-1}, f = TRUE)$, **return** h^N : computation of 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 from 1 to 64 bytes.

Algorithm 1: *BLAKE2b* algorithm

The main challenge in implementing the algorithm is the compression function which is called for each message block. The mixing of the message block is done in 12 rounds, in each round message word schedule is defined by 10 possible permutations $\sigma 0 \ldots \sigma 9$ (hard coded into design as two dimensional SIGMA–array). The mixing of the messages requires additional mixing function which mixes two 64 bit words from the message m_i into the hash state h_i. The auxiliary local 4×4 working vector $v[0..15]$ is used for 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 the transmitted data. The 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 or that a small change in message data does not result in a change in h, and it should be computationally impossible to reconstruct the input data from the hash value. The quality of our implementation of *BLAKE2b* hash function is verified by using the reference C source code given in.[5]

2 Implementation (proposed solution)

The algorithm for *BLAKE2b* hash function was broken down into several main operations and coded in VHDL as a state machine. The states were the following:

- **STATE_IDLE**: initialization of the *BLAKE2b* hash function i.e. initial values for the hash state vector $h[0..7]$ (array of 8 64-bit values). If there is a new block message to be compressed the flag input bit $valid\_in$ is set to high, the next state is STATE_PREPARE. The counter of the compressed bytes ($compressed\_bytes$) is increased for 128, or if it is the last message block, the compressed bytes counter is set to total message length.
- **STATE_PREPARE**: initialization for **compress** function - the local state vector $v[0..15]$ is initialized with hash state vector $h[0..7]$ mixed with the number of received bytes, the counter ci_done for the mixing rounds is reset to zero (there should be 12 rounds for message mixing), the nest state is STATE_COMPRESS.
- **STATE_COMPRESS**: resets the counter for the mixing function mi_done to zero (maximum value 7), the next state is the first mixing state. At the end of the mixing states all the columns and diagonals of the working vector $v[0..15]$ will be mixed with words of the current message block. The first mixing state is STATE_MIX_A

[3] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC).*
[4] Biryukov et al., *The memory-hard Argon2 password hash and proof-of-work function.*
[5] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC).*

- **STATE_MIX_A**: the first mixing state is one part of the implementation of mixing function G from[6] which requires additions of words. In each cycle one of the working vector $v[0..15]$ values is computed depending on the SIGMA permutation constant, current mixing round and mio_left 2-bit tag which serves for internal codification of the mixing equations. The next mixing state is STATE_MIX_B.
- **STATE_MIX_B**: the second mixing state is the second part of the implementation of mixing function G which requires xor operation and bit shifting. Again, in each cycle one of the working vector $v[0..15]$ values is computed depending on the SIGMA permutation constant, current mixing round and internal mio_left 2-bit tag which serves for internal codification of the mixing equations. The mio_left is updated after each mixing B operation. If there are still mixing operations to be done in a given round ($mi\_done \neq 7$) then the next state is STATE_MIX_A. There are in total 8 mixing operations (mi_done) divided in states A and B, and four operation codes (mio_left) for AB pairs. When all the codes are computed in a given order, the mixing code counter (mio_left) is incremented. If the last mixing operation is done but there are still mixing rounds to be computed the next state is STATE_COMPRESS and rounds counter 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 message is still not sent but all the mixing rounds are computed the next state is STATE_MIX_H.
- **STATE_MIX_H**: mixing of the upper $v[0..7]$ and lower half $v[8..15]$ of the working vector $v[0..15]$ into the current state vector $h[0..7]$. The next state is the STATE_WAIT for the next message block to be compressed.
- **STATE_WAIT**: expecting the following message block, if received $valid\_in$ goes high, the system variables are updated and the next state is STATE_PREPARE.
- **STATE_DONE**: when the last message block has been compressed, the hash output is computed. Next state is STATE_IDLE in which we wait the next message input.

The operations described above can be seen in Listing 1.

Listing 1: VHDL implementation of the *BLAKE2b* hash function

```
1
2  process(clk, reset)
3          --help variables for the mixing operations. These correspond
4          --to the variable names in the paper
5          variable a : std_logic_vector(63 downto 0);
6          variable b : std_logic_vector(63 downto 0);
7          variable c : std_logic_vector(63 downto 0);
8          variable d : std_logic_vector(63 downto 0);
9          variable x : std_logic_vector(63 downto 0);
10         variable y : std_logic_vector(63 downto 0);
11         variable help_sigma_x : integer range 0 to 15;
12         variable help_sigma_y : integer range 0 to 15;
13         begin
14                 if reset = '1' then
15                         state <= STATE_IDLE;
16                         current_chunk <= (others => '0');
17                         seen_last <= '0';
18                         compress_ready <= '1';
19                         h <= (others => (others => '0'));
20                         v <= (others => (others => '0'));
21                         compressed_bytes <= (others => '0');
22                         mio_left <= "00";
23                         valid_out <= '0';
24                         hash <= (others => '0');
25                 elsif rising_edge(clk) then
26                         --assign the right local vector and message to the variables
27                         --according to the index table
28                         a := v(ind(mi_done, 0));
29                         b := v(ind(mi_done, 1));
30                         c := v(ind(mi_done, 2));
31                         d := v(ind(mi_done, 3));
32                         help_sigma_x := SIGMA(ci_done, ind(mi_done,4));
33                         x := current_chunk(help_sigma_x*64+63 downto help_sigma_x*64);
34                         help_sigma_y := SIGMA(ci_done, ind(mi_done,5));
35                         y := current_chunk(help_sigma_y*64+63 downto help_sigma_y*64);
36
37                         case(state) is
38                                 when STATE_IDLE =>
39                                         --initialize the persistent state vector
40                                         h(1 to 7) <= VI(1 to 7);
41                                         h(0) <= VI(0) xor (X"00000000010100" &
42                                                 std_logic_vector(
43                                                 to_unsigned(hash_len, 8)));
44                                         --no bytes yet received
45                                         if valid_in = '1' then
46                                                 --if a message chunk is received, it is saved together
47                                                 --with all inputs and the state machine moves to the
```

[6] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*.

```
                        --prepare state
                        state <= STATE_PREPARE;
                        current_chunk <= message;
                        seen_last <= last_chunk;
                        ci_done <= 0;
                        compress_ready <= '0';
                        total_bytes <= message_len;
                        --if this was the last chunk, the number of received
                        --bytes is equal to the length of the received message.
                        --Otherwise it is increased by 128.
                        if last_chunk = '1' then
                                compressed_bytes <= std_logic_vector(
                                        to_unsigned(message_len, 128));
                        else
                                compressed_bytes <= std_logic_vector(
                                        unsigned(compressed_bytes) + 128);
                        end if;
                        --the entity is not ready to receive new input
                        valid_out <= '0';
                end if;
        when STATE_PREPARE =>
                --the persistent state vector is copied onto the local
                --state vector
                for i in 0 to 7 loop
                        V(i) <= h(i);
                end loop;
                V(8) <= VI(0);
                V(9) <= VI(1);
                V(10) <= VI(2);
                V(11) <= VI(3);
                --the number of received bytes is mixed into the vector
                V(12) <= VI(4) xor compressed_bytes(63 downto 0);
                V(13) <= VI(5) xor compressed_bytes(127 downto 64);
                --inverted if the last chunk is sent
                if seen_last = '1' then
                        V(14) <= not VI(6);
                else
                        V(14) <= VI(6);
                end if;
                V(15) <= VI(7);
                --reset the counter for the compression stage
                ci_done <= 0;
                --move on to the compress state
                state <= STATE_COMPRESS;

        when STATE_WAIT =>
                --a subsequent message chunk was received (not the first)
                if valid_in = '1' then
                        state <= STATE_PREPARE;
                        current_chunk <= message;
                        seen_last <= last_chunk;
                        compress_ready <= '0';
                        if last_chunk = '1' then
                                compressed_bytes <= std_logic_vector(
                                        to_unsigned(total_bytes,128));
                        else
                                compressed_bytes <= std_logic_vector(
                                        unsigned(compressed_bytes) + 128);
                        end if;
                end if;

        when STATE_COMPRESS =>
                --reset the counter for the mixing stage
                mi_done <= 0;
                --start mixing
                state <= STATE_MIX_A;
        when STATE_MIX_A =>
                --additions as defined by blake2b
                case mio_left is
                        when "11"|"01" =>
                v(ind(mi_done, 2)) <= std_logic_vector(
                        unsigned(c)+unsigned(d));
                        when "00" =>
                v(ind(mi_done, 0)) <= std_logic_vector(
                        unsigned(a)+unsigned(b)+unsigned(x));
                        when "10" =>
                v(ind(mi_done, 0)) <= std_logic_vector(
                        unsigned(a)+unsigned(b)+unsigned(y));
                        when others =>
                end case;

                state <= STATE_MIX_B;
        when STATE_MIX_B =>
                --xor's and shifts as defined by blake2b
                case mio_left is
                        when "00" =>
                v(ind(mi_done,3)) <= std_logic_vector(
                        unsigned(d xor a) ror 32);
                        when "11" =>
                v(ind(mi_done,1)) <= std_logic_vector(
                        unsigned(b xor c) ror 24);
```

```
                                        when "10" =>
                                    v(ind(mi_done,3)) <= std_logic_vector(
                                        unsigned(d xor a) ror 16);
                                        when "01" =>
                                    v(ind(mi_done,1)) <= std_logic_vector(
                                        unsigned(b xor c) ror 63);
                                        when others =>
                                    end case;

                                    --last mix
                                    if mi_done = 7 and mio_left = "01" then
                                        --also last compression
                                        if ci_done = 11 then
                                            --also last chunk
                                            if seen_last = '1' then
                                                state <=
                                                STATE_DONE;
                                            else
                                                state <=
                                                STATE_MIX_H;
                                            end if;
                                            --ready to receive a new chunk
                                            compress_ready <= '1';
                                        else
                                            --next compression
                                            state <= STATE_COMPRESS;
                                            ci_done <=
                                                ci_done + 1;
                                        end if;
                                    else
                                        if mio_left = "01" then
                                            mi_done <=
                                                mi_done +1;
                                        end if;
                                        state <= STATE_MIX_A;
                                    end if;
                                    mio_left <= std_logic_vector(unsigned(mio_left) + 3);
                                when STATE_DONE =>
                                    --write output
                                    for i in 0 to 7 loop
                                        hash(i*64+63 downto i*64) <= h(i) xor v(i) xor v(i+8);
                                    end loop;
                                    valid_out <= '1';
                                    compressed_bytes <= (others => '0');
                                    state <= STATE_IDLE;
                                when STATE_MIX_H =>
                                    state <= STATE_WAIT;
                                    --mix into h
                                    for i in 0 to 7 loop
                                        h(i) <= h(i) xor v(i) xor v(i+8);
                                    end loop;
                                when others =>
                                    state <= STATE_IDLE;
                            end case;
                        end if;
                end process;
```

3 Results (verification plan)

In order to verify the proper operation of the implemented *BLAKE2b* hash function, a test bench was made that uses as an input `messages.txt` file. The content of this file is the input data to be hashed. The same input data is being used by the reference C code[7] which computes its output hash values into the document `hashes.txt`. In the test bench each line from the `messages.txt` is being read and the final hash values are computed by using our VHDL implementation. These values are then compared with the results of implementation in the reference C code stored in the document `hashes.txt`. In order to read the hash values from `hashes.txt`, a conversion from `hexadecimal` to `std_logic_vector` was needed; the ASCII_2_STD function was used for this purpose. This can be seen in the Listing 2.

Listing 2: Test bench process for verifying the hardware implementation using the reference implementation in C

```
stimuli : process
                type char_file_t is file of character;
                file message_file : TEXT open read_mode is "messages.txt";
                file hash_file : TEXT open read_mode is "hashes.txt";
                variable line_buffer : line;
                variable value_in : std_logic_vector(64*8-1 downto 0);
                variable char_value_1 : std_logic_vector(7 downto 0);
                variable char_value_2 : std_logic_vector(7 downto 0);
                variable read_ok : boolean;
                variable current_char : character;
                variable counter : integer;
```

[7] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*.

```
12      begin
13
14              --always generate 64-byte hashes
15              hash_len <= 64;
16              last_chunk <= '0';
17
18              --start with reset
19              reset <= '1';
20              wait for 10 ns;
21              reset <= '0';
22              wait for 5 ns;
23
24              counter := 0;
25              message <= (others => '0');
26              while not endfile(message_file) loop
27                      counter := 0;
28                      message <= (others => '0');
29                      wait for period;
30
31                      --read single line
32                      readline(message_file, line_buffer);
33                      --message length equals line length
34                      message_len <= line_buffer'length;
35
36                      for i in 0 to line_buffer'length-1 loop
37                              --read one byte of data and write it to message
38                              --if message is filled up, send it to the entity
39                              --and start over
40                              if counter = 128 then
41                                      wait for period;
42                                      last_chunk <= '0';
43                                      valid_in <= '1';
44                                      wait for period;
45                                      valid_in <= '0';
46
47                                      counter := 0;
48                                      message <= (others => '0');
49                                      wait for period*835;
50                              end if;
51
52                              read(line_buffer, current_char);
53                              char_value_1 := std_logic_vector(to_unsigned(
54                                      character'pos(current_char),8));
55                              message(counter*8+7 downto counter*8) <=
56                                      char_value_1;
57                              counter := counter + 1;
58                      end loop;
59
60                      --send the remaining bytes as last chunk
61                      wait for period;
62                      last_chunk <= '1';
63                      valid_in <= '1';
64                      wait for period;
65                      valid_in <= '0';
66                      wait for period*835;
67
68                      readline(hash_file, line_buffer);
69                      --report "line " & line_buffer.all;
70
71                      --read hash file in hex and compare with the output
72                      --generated by the entity
73                      counter := 0;
74                      value_in := (others => '0');
75                      for i in 0 to 63 loop
76                              read(line_buffer, current_char);
77                              char_value_1 := std_logic_vector(to_unsigned(
78                                      character'pos(current_char),8));
79                              read(line_buffer, current_char);
80                              char_value_2 := std_logic_vector(to_unsigned(
81                                      character'pos(current_char),8));
82                              value_in(counter*8+7 downto counter*8) :=
83                                      ASCII_2_VEC(char_value_1) &
84                                      ASCII_2_VEC(char_value_2);
85                              counter := counter + 1;
86                      end loop;
87
88                      --report "valu " & to_hstring(value_in);
89                      --report "hash " & to_hstring(hash);
90
91                      if value_in = hash then
92                              report "[_OK]_HASH_correct";
93                      else
94                              report "[NOK]_HASH_incorrect";
95                      end if;
96              end loop;
97
98              ended <= '1';
99
100             wait;
101     end process;
```

The state transitions for the case when the next block message is to be received, together with the counters for mixing rounds and mixing operations, are shown in Figure 1. After the previous message block was compressed (round counter ci_done is 11, and the state is STATE_WAIT), as the new message block is available the $valid\_in$ goes high for one clock cycle, the $current\_chunk$ becomes the new message block, the state is STATE_PREPARE, and the message block register $message$ is freed. The following are the mixing states, after each mixing A and mixing B pair the mixing code mio_left is updated. The compression of one message block takes 835 clock cycles.

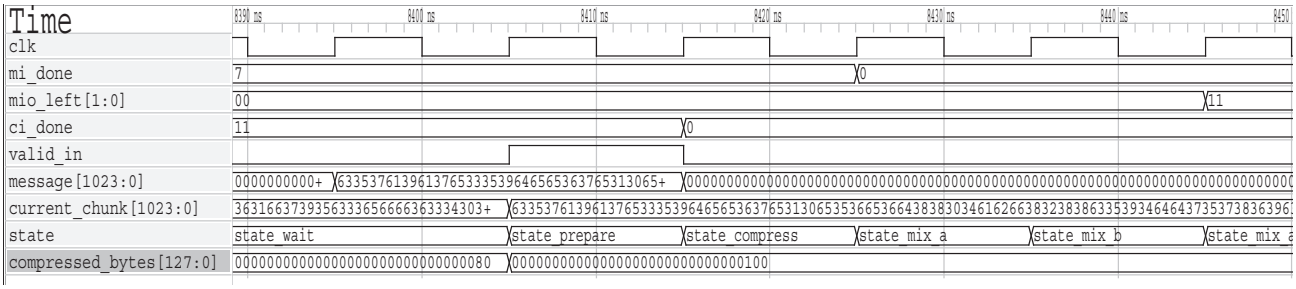

Figure 1: Simulation results for blake2b hash function: state transitions

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.

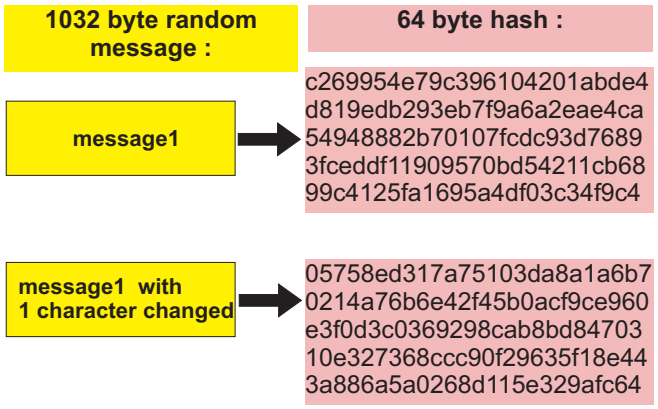

Figure 2: Similar input data results in a very different hash messages

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 3.

Listing 3: 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 :
```

```
25|            8 Input    128 Bit        Muxes := 1
26|            2 Input    128 Bit        Muxes := 2
27|            3 Input     64 Bit        Muxes := 3
28|            8 Input     64 Bit        Muxes := 23
29|            4 Input     64 Bit        Muxes := 1
30|            9 Input     64 Bit        Muxes := 1
31|            8 Input      4 Bit        Muxes := 11
32|            8 Input      3 Bit        Muxes := 11
33|            2 Input      3 Bit        Muxes := 1
34|            3 Input      3 Bit        Muxes := 1
35|            8 Input      1 Bit        Muxes := 17
36|           11 Input      1 Bit        Muxes := 12
37|            4 Input      1 Bit        Muxes := 16
38|            2 Input      1 Bit        Muxes := 27
39|---------------------------------------------
40|Finished RTL Hierarchical Component Statistics
41|---------------------------------------------
```

4 Discussion

In the reference paper[8] 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.

[8] Saarinen and Aumasson, *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*.

6 Assessment

This is the place for the teaching staff to add notes for team assessment.

#	Issue	Yes	No
1 Implementation			
1.1	Does the implementation conform to the specification?		
1.2	Is the implementation resource-efficient?		
1.3	Is the implementation's hardware description language (HDL) complexity low?		
1.4	Is the implementation well-documented?		
1.5	Is the file structure's complexity low?		
2 Coding style			
2.1	Is the line width of code code limited to 80 characters?		
2.2	Is white space appropriately used?		
2.3	Are tabs used for indentation?		
2.4	Are separators used to logically divide the file contents?		
2.5	Are meaningful comments given?		
3 Code reuse			
3.1	Is publicly available code re-used?		
3.2	Is non-publicly available code re-used?		
3.3	Are the sources of re-used code cited?		
4 Interaction			
4.1	Was the specification unclear to the team?		
4.2	If yes, did the team contact the teaching staff to make the specification clear?		
5 Report			
5.1	Are there typos?		
5.2	Is the report grammatically correct?		
5.3	Is there redundant information?		
5.4	Is the report's format consistent?		
5.5	Are captions properly used and numbered? Page numbers?		
5.6	Are figures and tables properly referenced in the body text?		
5.7	Are resources properly referenced?		

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.