Full Hash Algorithm v2 (DES S-Box based)

Project Report

2021/2022

UNIVERSITÀ DI PISA

Project Report for the Project of the Course

'Hardware and Embedded Security' of the M.Sc. in Cybersecurity

Professors:

Prof. Sergio Saponara

Ing. Luca Crocetti

Students:

Francesco Venturini

Pierfrancesco Bigliazzi

Summary

Chapter 1 - Specification Analysis

The given specification presents the requirements for the hardware design of an hash module based on the S-box of the DES algorithm. The input of the module is a message that can have an arbitrary length and that is divided in bytes to perform the computation, the output is a 32-bit digest formed by the concatenation of 8 nibbles.

The module is characterized by the following parameters:

```
module full_hash_des_box(
      input rst_n,
      input clk,
      input M_valid,
      input [7:0] message,
      input [63:0] counter,
      output reg [31:0] digest_out,
      output reg hash_ready
);
```

where:

- **rst_n**: asynchronous active-low reset port
- **clk**: system clock
- **M_valid**: input port that must be asserted when providing the input message byte `message` (`1'b1` when input character is valid and stable, `1'b0` otherwise)
- **message**: input message byte, can be any 8-bit ASCII character
- **counter**: real byte length of `message` (if the length is 1 byte, then `counter = 1`)
- **digest_out**: 32-bit output register for the computed hash value
- **hash_ready**: output port that must be asserted when the generated output digest is available (`1'b1` when output digest is valid and stable, `1'b0` otherwise)

For each byte of the input message the hash module performs the following operations:

$$\text{for}(r=0;\ r<4;\ r{+}{+})$$
$$\quad \text{for}(i=0;\ i<8;\ i{+}{+})$$
$$\quad\quad H[i] = (H[(i+1) \bmod 8]) \oplus S(M_6)) \ll \lfloor i/2 \rfloor$$

We can see that the computation is composed by 4 rounds. In each round every byte of the message is compressed according to the formula:

$$M_6 = \{\mathrm{M}[3] \oplus \mathrm{M}[2], \mathrm{M}[1], \mathrm{M}[0], \mathrm{M}[7], \mathrm{M}[6], \mathrm{M}[5] \oplus \mathrm{M}[4]\}$$

and then it is given as input to the S-box of the DES algorithm (the first and last bits of the sequence select the row of the table and the remaining 4 central bits select the column). The table outputs a specific 6-bit value that is then XORed with a value of the 32-bit array H according to the formula above (this variable is initialized with the values: 0x4B71DF03). The partial result of this operation is then left-shifted circularly by n bits, where n=$\lfloor i/2 \rfloor$ and $\lfloor\ \rfloor$ is the floor function.

Once the last message byte has been processed, the hash module performs a final elaboration on the partial result given by the overlap of the previous computations:

$$\text{for(i=0; i<8; i++)}$$
$$\mathrm{H[i]} = (\ \mathrm{H[(i+1)\ mod\ 8]}) \oplus \mathrm{S(C_6[i])}\ ) \ll \lfloor i/2 \rfloor$$

In this case, the S-box is called with a parameter C_6[i], a 6-bit vector that is obtained from the i-th byte of the input counter according to the following operation:

$$C6\ [i] = \{C[i][7] \oplus C[i][1], C[i][3], C[i][2], C[i][5] \oplus C[i][0], C[i][4], C[i][6]\}$$

The S-box is structured as the following:

S_5		4 central bits															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
External bits	00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
	01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1100	0011	1001	1000	0110
	10	0100	0010	0001	1011	1100	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
	11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1100	0100	0101	0011

Figure 1: DES S-box

The expected waveforms are the following. For the input:

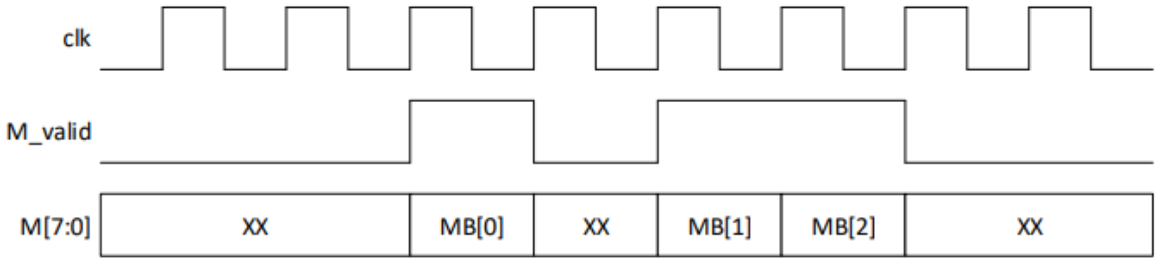

Figure 2: Expected waveform for the input

For the output:

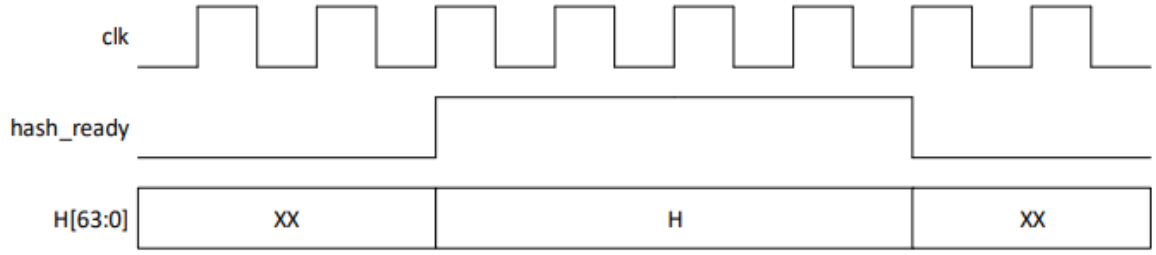

Figure 3: Expected waveform for the output

Chapter 2 - Block Diagram and Design Choices

2.1 - Design Choices

For the implementation of the hash module, we have decided to split the computation in two main submodules: `H_main_computation` and `H_last_computation`.

The first module `H_main_computation` is declared as the following:

```
module H_main_computation(
        input [7:0] m,
        input [7:0] [3:0] h_main,
        output [7:0] [3:0] h_main_out
);
```

The input parameters are the message byte to be elaborated and the registers in which are stored the initialization value for the array variable H or the partial result of the previously

elaborated bytes. On the other hand, the output parameter is a variable which contains the result produced by the four rounds of the hash algorithm.

Indeed, this H_main_computation module exploits another important submodule that is the Hash_Round module.

Furthermore, it uses another module, called S_Box, that calculates the output of the given DES-based S-box according to the value of the variable m6. The variable m6 is the result of the compression function detailed above applied on the input variable m. The module is declared as the following:

```
module S_Box(input [5:0] in, output reg [3:0] out);
```

This module implements a LUT version of the DES S-box in which the first and the last bits of the input in select the row of the S-Box table and the four central bits select the column of the latter. The output is then presented on the register out.

For what concerning the Hash_Round module, it is declared as follows:

```
module Hash_Round(
        input [3:0] S_Box_value,
        input [7:0] [3:0] h_main,
        output reg [7:0] [3:0] h_out
);
```

The first parameter is the value produced as output of the S_Box module called with the m6 specific value. The second parameter is the actual value of the array variable H. And, finally, the third one is used to store the result of the module computation according to the specifications.

This submodule is used four separate times, one for each round of the algorithm, in the H_main_computation module, where the output of each round is given as input for the next round. The overall result is then provided as output of the main module.

The other primary module is the H_last_computation module. It is declared as shown below:

```
module H_last_computation(
       input [7:0] [3:0] H_main,
       input [63:0] counter,
       output reg [7:0] [3:0] H_last
);
```

The input parameters are the array variable H with the values computed in the main computation and the counter which contains the real byte length of the message. The output parameter is the digest which is given as output of the main module.

Within this module are used two submodules: the first one is S_Box and the other is Counter_to_C_6. The S_Box module is the same described previously. Counter_to_C_6 performs a compression based on the byte length of the message (as detailed in the specification section), its declaration is:

```
module Counter_to_C_6(input [7:0] in_c, output reg [5:0] out_c);
```

The first parameter in_c is the real byte length of the message. The second parameter out_c is the result of the compression function. At the end of the module H_last_computation a combinational network is used to provide the 32-bit digest output.

For the implementation of all these functions the following registers and wires have been used:

- **H_MAIN**: this register is used to store the initial value of the array variable H and the partial result of the main computation
- **H_LAST**: this register is used to store the result of the last computation
- **MSG**: register that contains the input message byte
- **C_COUNT**: this is a utility register that maintains the remaining number of message bytes to be computed
- **COUNTER**: this register samples the real byte length of the message input

- **M_VALID_R**: a utility register that samples the correct value of the full_hash_des_box M_valid input
- **half_hash**: a wire that maintains the partial result of the elaboration on the main computation only (the one based on the rounds)
- **STAR**: status register, to implement the Finite State Machine (see below)

2.2 - Block Diagram

The following block diagrams can be found in Quartus Prime Lite Edition from Tools > Netlist Viewer > RTL viewer.

This portion of the entire hash module shows the `H_main_computation` module:

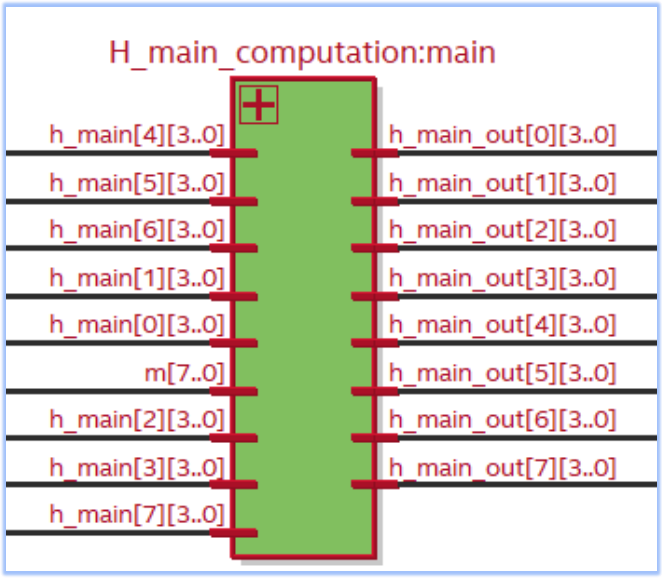

Figure 4: H_main_computation module

The block can be expended to see that it is composed by the cascade of the four hash rounds (Figure 5).

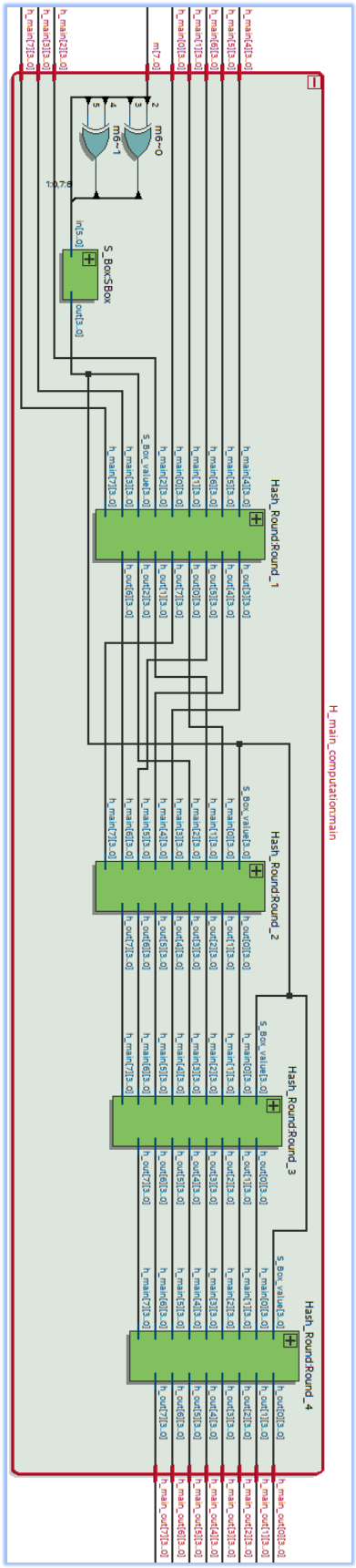

Figure 5: Rounds within H_main_computation

Figure 6 below shows the block diagram of the `Hash_Round` module:

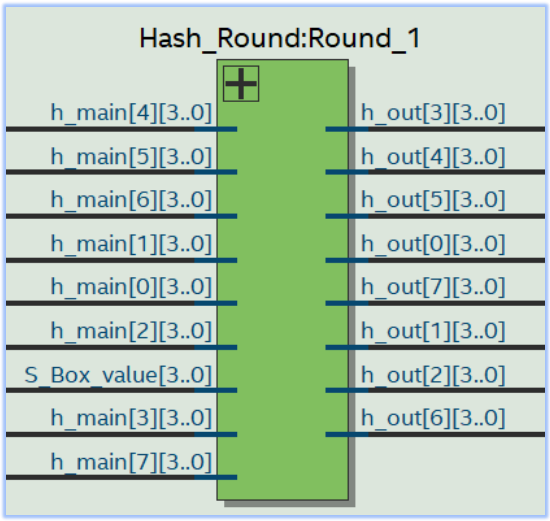

Figure 6: General Hash_Round module

Inside the module there are several XOR gates which, together with some concatenation, perform the operation described in the specification analysis section.

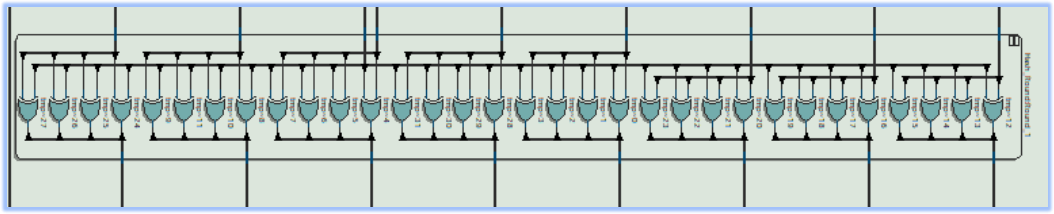

Figure 7: XORs and shifts in the Hash_Round module

The following is the `H_last_computation` module. As detailed before, it takes as input the value of the main computation (partial result given by the rounds computed on all the characters of the input message) and the real byte length of the latter. The output is the hash value of the algorithm.

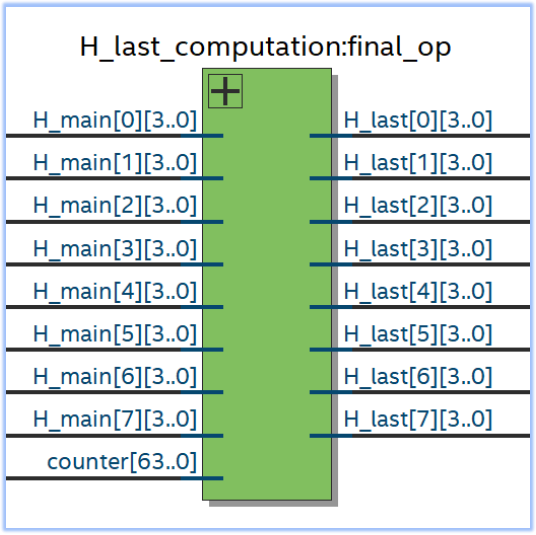

Figure 8: H_last_computation

The module is composed by several `S_Box` and `Counter_to_C_6` submodules and it ends with a few XOR operations and left circular shifts:

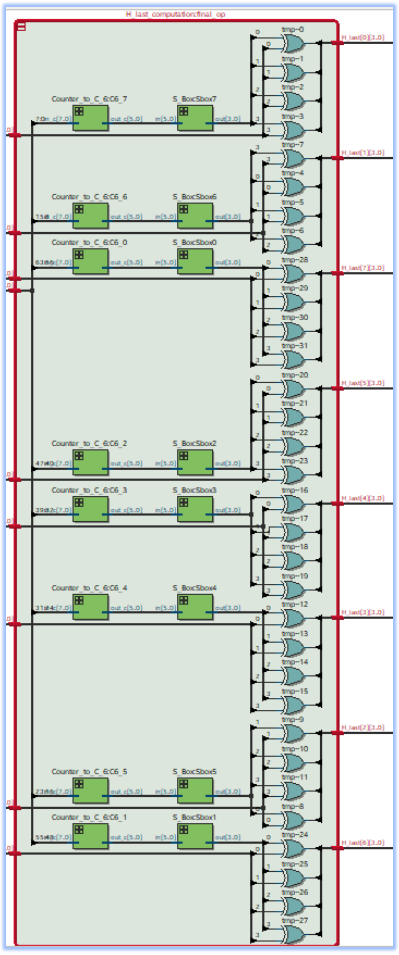

Figure 9: submodules of H_last_computation

Finally, the S_Box module is:

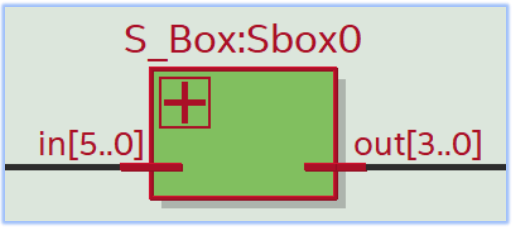

Figure 10: S_Box module

The schematic below shows that the implementation of this module is basically based on a decoder, some OR gate and four multiplexers in parallel:

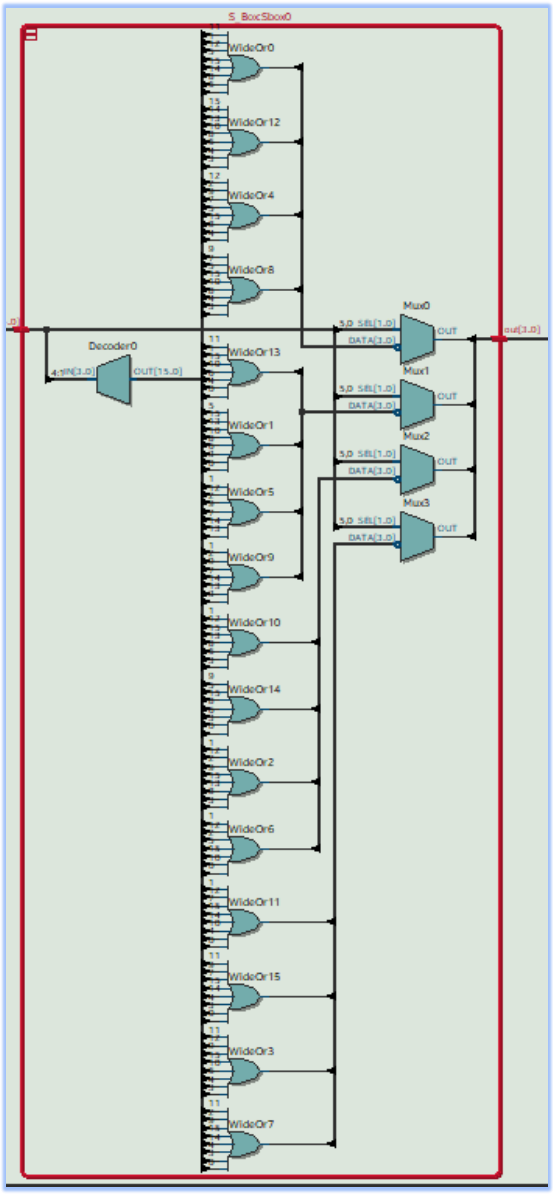

Figure 11: S_Box implementation

2.3 - Finite State Machine

The FSM is built on the usage of a register called STAR (STAtus Register). As the name suggests this register maintains the actual state of the finite machine, determining the operations to be performed and allowing to control the flow of execution of the modules.

The FSM is defined as an always block that depends on two conditions:

1. The rising edge of the system clock
2. The falling edge of the asynchronous reset of the system

This second condition allows to discriminate the initialization phase of the system in which all the relevant register are set according to the specific needs. This is represented by the following snippet of code (taken from *fullHashDesBox.sv* file):

```
// Finite State Machine, see documentation
always @(posedge clk or negedge rst_n) begin

    if(!rst_n) begin

        // initialization
        STAR <= S0;
        hash_ready <= 0;

    end else begin
```

Figure 12: reset condition

In the else branch we have the actual implementation of the FSM. The latter is divided into three states:

- **S0**: the first state is the one in which the module waits for an input to be asserted in order to sample it. There we have also the initialization of the H_MAIN register with the value provided by the specification and the reset of the hash_ready output port each time a new computation (signalled by the M_valid input) is started. Lastly, the state transition is done only if a new computation must start and the next state for the FSM is decided depending on the value of the input counter (if it is equal to 0 then the next state will be S2, in which only the last computation is performed, otherwise the next state will be S1).

- **S1**: in this state the main computation of the algorithm is performed, indeed the `H_MAIN` register is updated with the partial result produced by the `H_main_computation` module (variable `half_hash`), but only if we are in a valid situation, otherwise `H_MAIN` is going to keep the old value. In this state we have also implemented the counting of the remaining bytes to elaborate decrementing the `C_COUNT` register value, as well as the continuous sampling of the inputs to not skip possible consecutive bytes. In this state we can have a state transition to S2 only when all the bytes of the message have been elaborated (when `C_COUNT == 1`), otherwise the FSM cycles in the state S1.

- **S2**: in this state the output is asserted through the setting of the `hash_ready` port to the value `1'b1` and the digest value is presented at the `digest_out` output register. In there we continue also to sample the M_valid value and we perform an unconditional state transfer to S0 in order to be able to start a new computation from scratch.

The state diagram is the following:

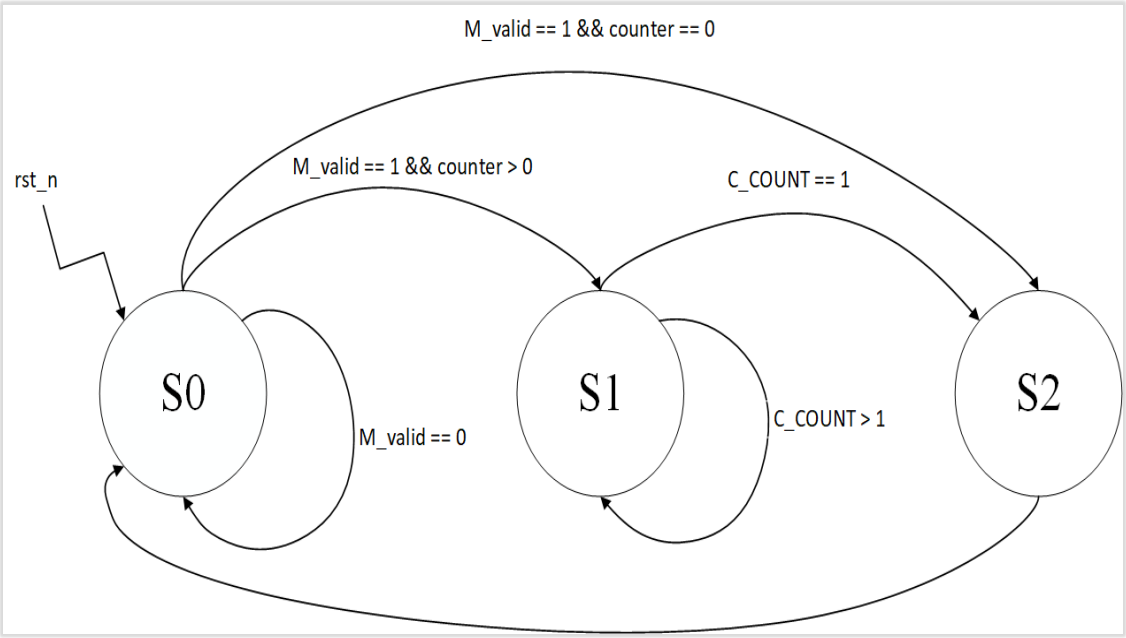

Figure 13: Finite State Machine Diagram

Chapter 3 - Expected Waveforms

In this chapter we show three examples of expected waveforms taken from Modelsim. See Chapter 4 for details about the testbenches.

- Empty message: this waveform shows the values assumed by the signals during the testbench `TEST_ZERO_LENGTH`. As we can see the input is sampled into the relative registers and after only one clock cycle `digest_out` and `hash_ready` are set.

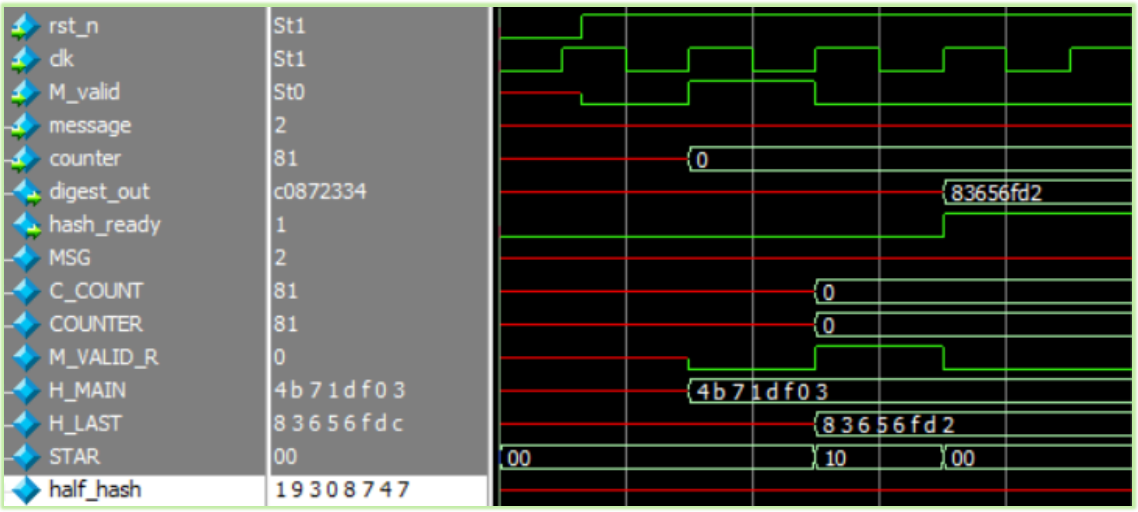

Figure 14: waveform for an empty message input

- One character: this is the waveform that represents the computation of the testbench `TEST_UPPERCASE_A`. In this example the digest is produced two clock cycles after the sampling of the inputs.

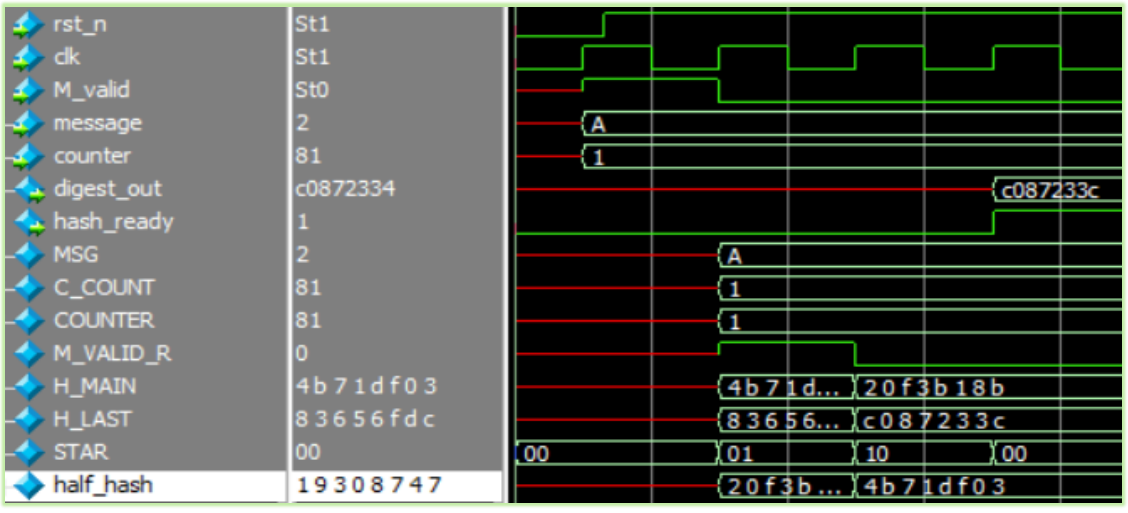

Figure 15: waveform for character 'A' input

- Long sequence: this figure shows the elaboration of the testbench TEST_LONG_SEQUENCE.

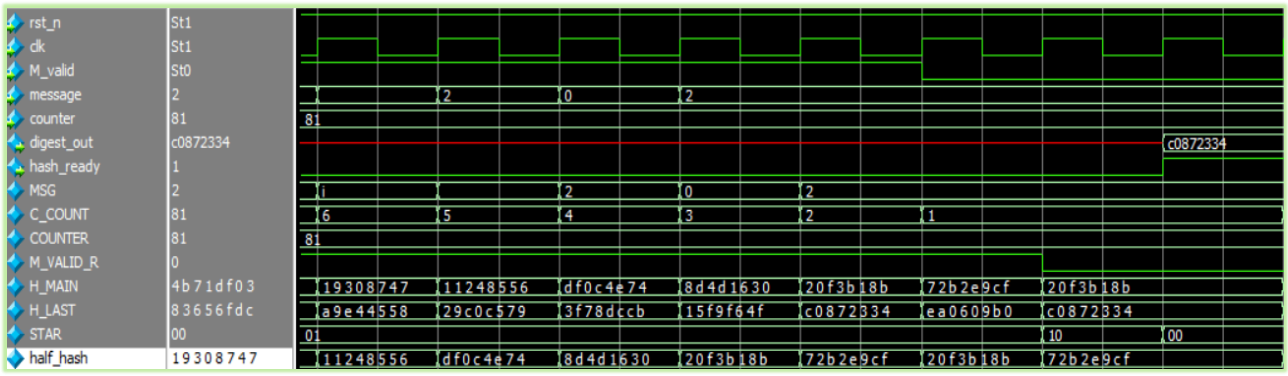

Figure 16: waveform for a long sequence input

Chapter 4 - Testbench

The testbenches developed to test the functionalities of the module are reported below:

- **TEST_ZERO_LENGTH**: It produces the digest of an empty message which has length equal to 0 and computes only the final computation.
- **TEST_UPPERCASE_A**: This test processes a single character message. In this case the character selected is the letter A. The test is composed by the main computation composed by the 4 rounds and the final computation.
- **TEST_SEQUENCE_AB**: It reads and processes a sequence composed by two consecutives characters. The characters are the letters A and B.
- **TEST_SEQUENCE_A_CLK_B**: This test computes the same sequence as above, but the two characters are separated by a clock cycle.
- **TEST_SAME_MESSAGE_SAME_HASH**: It processes the same sequence twice. Once processed the first one the second sequence is computed after some clock cycles. This test aims to ensure that the same message produces the same digest.
- **TEST_LONG_SEQUENCE**: This one produces the digest of a long sequence of characters. The message used for this test is:

“HARDWARE_AND_EMBEDDED_SECURITY_FULL_HASH_DES_BOX_PROJECT_bigliazzi_venturini_2022”

All the outputs of the testbenches have been compared with the results produced by a script written in python which simulates the behaviour of the hash module, available on GitHub.

Chapter 5 - Implementation of RTL design on FPGA and results

In the image below it is reported the number of resources used by the module after the synthesis and the fitting on a FPGA device.

Fitter Status	Successful - Mon Jun 27 11:35:36 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	fullHashDesBox
Top-level Entity Name	full_hash_des_box
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	195 / 113,560 (< 1 %)
Total registers	207
Total pins	1 / 378 (< 1 %)
Total virtual pins	107
Total block memory bits	0 / 12,492,800 (0 %)
Total RAM Blocks	0 / 1,220 (0 %)
Total DSP Blocks	0 / 342 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 17 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 17: results of the design

The total logic resources are less than 1% in relation to those available in the device.

The number of registers implemented is 207.

In the device there are 108 pins, 107 virtual pins and 1 real pin which is the clock:

- `rst_n`: 1-bit
- `M_valid`: 1-bit
- `clk`: 1-bit
- `message`: 8-bit
- `counter`: 64-bit
- `digest_out`: 32-bit
- `hash_ready`: 1-bit

Chapter 6 - Static Timing Analysis

For the static timing analysis, we created a file containing the timing constraints for the hash module and the relative input/output delays for the unconstrained paths (from a minimum of 10% to a maximum of 20% of the clock). The file can be found in *quartus/constr/fullHashDesBox.sdc*.

After the assignment of virtual pins, the results for the clock frequency increased, reaching the following values:

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	130.92 MHz	130.92 MHz	clk	

Figure 18: clock frequency at 85°C

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	130.51 MHz	130.51 MHz	clk	

Figure 19: clock frequency at 0°C

Our design is then capable to guarantee a 130.51 MHz frequency in the worst case rising up to 130.92 MHz.

The details of the project can be found at:

https://github.com/Portgas97/fpga_full_hash_algorithm_des_sbox

https://github.com/PierfrancescoBigliazzi/Hardware-and-Embedded-Security-Project