Light Hash algorithm v2 (DES S-box based)

HARDWARE AND EMBEDDED SECURITY COURSE - PROJECT REPORT

CALOGERO COSTA, CHIARA VENERA LI DESTRI

A.A. 2022/2023

Sommario

Project Specifications

In this project, a module based on the S-Box of the DES algorithm is designed. The hash algorithm generates a 32-bit digest formed by the concatenation of 4-bit vectors $H[i]$, being $H[0]$ the most significant 4-bit vector. For each message, the $H[i]$ variables are initialized with the values shown in the table below:

	$H[0]$	$H[1]$	$H[2]$	$H[3]$	$H[4]$	$H[5]$	$H[6]$	$H[7]$
Init. value	4'hF	4'h3	4'hC	4'h2	4'h9	4'hD	4'h4	4'hB

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

for (r=0; r<4; r++)

for (i=0; i<8; i++)

$$H[i] = (H[(i+2)\ mod\ 8 \oplus S(M_6)]) \ll \left\lfloor \frac{i}{2} \right\rfloor$$

where:

- $mod\ n$ is the modulo operator by n.
- $\oplus$ is the XOR operator.
- $X \ll n$ is the left circular shift by n bits.
- $\lfloor x \rfloor$ is the floor function applied to argument x.
- M_6 is a 6-bit vector generated from the input message byte M by the following compression function:

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

- $S()$ is the S-box transformation of DES algorithm, that works over a bytes.

Other additional design specifications are the following:

- The module shall have an asynchronous active-low reset input port, **`reset`**, that can be used to reset the status of the module.
- The input message byte, M, can be any 8-bit ASCII character.
- The module is expected to process one byte of the input message, assumed as an ASCII character, in one clock cycle.
- The light hash v2 module shall have an input port, **`M_valid port`**, which has to be asserted when providing the input message byte M: 1'b1, when the input character is valid and stable, 1'b0, otherwise.
- The Light Hash v2 module shall have an output port, **`hash_ready port`**, which is asserted when the generated output digest or hash value is available at the corresponding output port. The **`hash_ready`** port shall be kept to logic 1 until a new message digest computation is performed.

Block Diagram and Design Choice

Based on what was said in the previous paragraph, the following choices were made for the module design and the corresponding architecture.

Block diagram

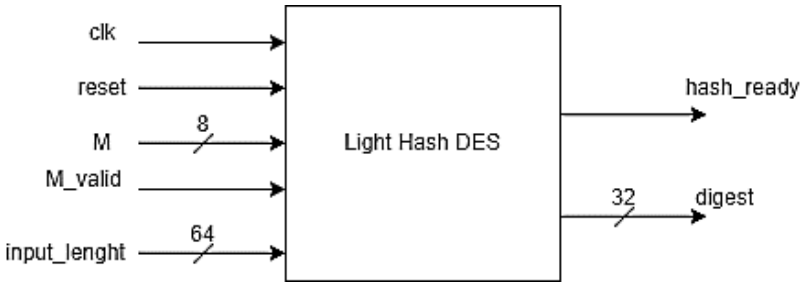


Figure 1 Module Interface Diagram

As can be seen from the figure, the module has five inputs and two outputs. The inputs are the following:

- *clk*: This is the clock signal; all the ports of the module are sampled on the rising edge of the clock.
- *reset*: is the low active asynchronous reset signal used to restore the default values of the registers inside the module.
- *M_valid*: This input signal is used as a flag to understand when the input character is valid and stable.
- *M*: represents the input byte to be processed by the module. It represents an 8-bit ASCII character and is encoded on 8 bits.
- *input_lenght*: specifies the input message's length M. This input is encoded on 64 bits.

The outputs of the module are:

- *digest*: This signal represents the result of the processing activity performed over the input message. It is encoded in 32 bits.
- *hash_ready*: This signal is used like a flag to understand when the digest of the input message is ready and stable.

Expected behaviour and usage

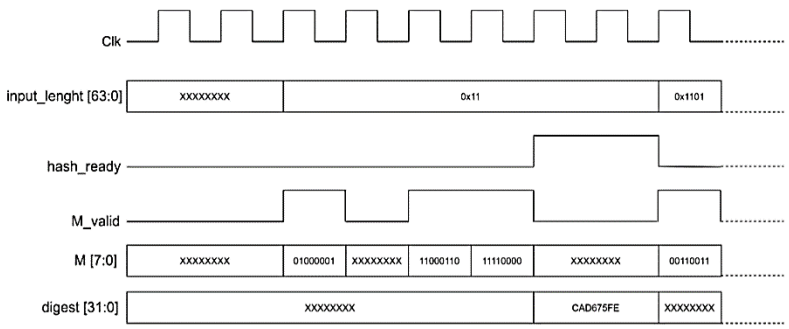


Figure 2 Expected waveform

The diagram represents the expected waveform. The following are some considerations on how the circuit must be used to be properly integrated into a safe hardware module:

- The input_lenght shall be set to the input message's length when the message's first character is provided on the M port. Remains set throughout the processing activity of the message.

- The hash_ready signal is only set after the last character of the input message has been calculated, so when the output digest is ready and stable; the signal is reset when a new character is provided in M_valid.
- The M_valid signal shall be set when the input character M has to be considered valid and stable.
- The input M has to be set with the ASCII value of the character that has to be processed.
- The digest register stores the digest value only after the entire message in input is processed. The content of the register must be considered valid only if hash_ready is set.

To use the model, the following steps shall be performed:

- To start a processing activity, the user must set M_valid and send the first byte M, along with the input length of the entire message.
- The user must set M_valid and send one byte per clock cycle to port M.
- The module by setting hash_ready notifies when the processing task ends and the digest is ready; the value of the digest is stored in the digest register.

Design choices

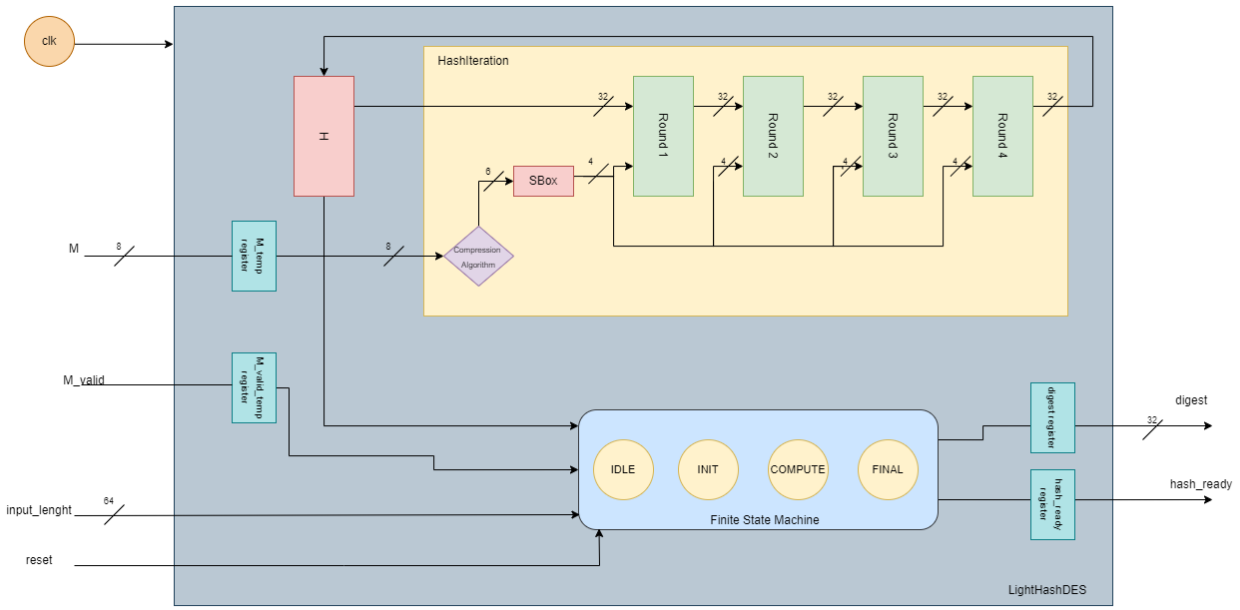


Figure 3 Block diagram of the design

All the design choices that were made to increase the performance and readability of the implemented design are listed below:

- To avoid long combinatorial paths, the input byte M and the M_valid flag are stored in two dedicated registers, respectively temp_M and M_valid_temp.
- The input_lenght port has been added to let the device know when the last byte has been processed, whether the digest is ready or whether there are more bytes to process.
- To avoid long combinatorial paths, the digest and the hash_ready flag are stored in two dedicated registers: digest e hash_ready.

Each byte of the message is processed in a single clock cycle by a series of combinational sub-modules; the decision to split the project into different modules (stored in different files) was made to improve the clarity and reliability of the code.

- *Sbox* is a combinatory sub-module that implements a LUT S-Box with 6-bit and 4-bit output. The interface of the module is composed of:
    - The 6-bit signals used to access the S-Box
    - The 4-bit output of the S-box
- *Round*, like the Sbox module, is a combinatory sub-module that performs the operations: XOR, modulo, floor, and circular shift. The interface is composed of:
    - sbox_out: represents the output of the Sbox module.
    - h_in: It represents the temporary 32-bit digest in input.
    - h_out: This is the temporary 32-bit digest in output.
- *HashIteration* is a combinatory sub-module that performs: the compression algorithm over the input byte, the S-box lookup operation, and instances four Round sub-module, to perform the round operation four times. The interface is composed of:
    - M: representing the message byte to process.
    - h: the temporary digest in input.
    - h_out: the temporary 32-bit digest in output

All these sub-modules are incorporated in the lightHashDES top-level module, which contains the control network and the FSM that regulates the data flow of the hash function.

Finite State Machine, FSM

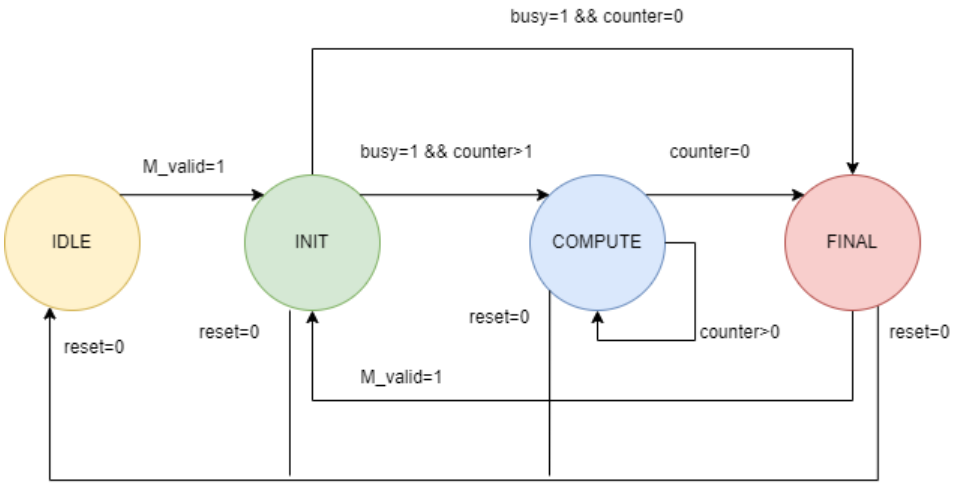


Figure 4 Finite State Machine

The lightHashDES module integrates a Finite State Machine (FSM) implemented to better manage the project's data flow. The FSM has four states and transitions are executed by checking the conditions on the values of the module's support variables and input/output ports of the model:

The FSM uses the following variables:

- Busy: is a binary variable set when a digest computation is performed. Otherwise, is unset when the module is not computing a digest.
- Counter: This is a 64-bit variable used to track the remaining number of bytes that need to be processed. Once the first byte of the incoming message is entered, the length of the message is stored in the counter and after each byte calculation, the counter variable is decremented by one.

The states of the FSA are:

- *IDLE*: The module is waiting for a new message to process. In this state hash_ready and busy are set to 0. If:
    - M_valid is set, triggering a transition from IDLE to INIT state.
- *INIT*: in this state, the module sets the initial value of the digest, stores the length of the input message in the counter register, sets busy to 1 and hash_ready to 0, stores M in the register M_temp and M_valid in M_valid_temp. If:
    - counter > 0 and busy = 1 a transition to COMPUTE state is triggered.
    - counter = 0 and busy = 1 a transition to the FINAL state is triggered.
    - reset = 0 a transition to the IDLE state is triggered.
- *COMPUTE*: The module processes the byte that is stored in the register M_temp, updates the temporary digest value, and decrements the counter by one. If M_valid is set, the module stores the next byte to process in the M_temp register. If:
    - counter > 0 a self-referencing transition to COMPUTE state is triggered.
    - counter = 0 a transition to the FINAL state is triggered.
    - reset = 0 a transition to the IDLE state is triggered.
- *FINAL*: The module sets hash_ready to 1, sets busy to 0, and stores the temporary digest value in the output port digest. If:
    - M_valid = 1 a transition to INIT state is triggered.
    - reset= 0 a transition to the IDLE state is triggered.

Python Model

A high-level implementation of the lightHashDES module is provided. The language chosen is Python. The Python model reproduces at a high level the behaviour, described in SystemVerilog, of the hash module based on the SBox of the DES algorithm. The program takes a string as input and returns the string digest on 32 bits.

Model description

To be able to correctly implement the behaviour described at a low level, auxiliary functions have been defined. These functions are shown below.

```
#definition of the xor operation
def xor(bit1, bit2):
    if(bit1 == bit2):
        return '0'
    else:
        return '1'

#utility function used to convert the content of a list into a string
def toString(l):
    s = l[0]+l[1]+l[2]+l[3]+l[4]+l[5]+l[6]+l[7]

    return s
```

After initialising M6 and the vector H according to the project specification, the functions, which perform at a high level the tasks of the modules written in SystemVerilog, are defined.
The function compute_sBox calculates the Sbox that will be used for the hash calculation and emulates at a high level the behaviour described in the SBox.sv module. The code is as follows:

```
#function used to compute the SBox
def compute_sBox(M_six, sBox):

    row = int(M_six[5]+M_six[0], 2)
    column = int(M_six[4]+M_six[3]+M_six[2]+M_six[1], 2)

    return sBox[row][column]
```

The sbox is declared and initialised like this:

```
#declaration of the S-Box
sBox = 4*[16*[0]]

#initialization of the S-Box by rows
sBox[0] = ( "0010", "1100", "0100", "0001", "0111", "1010", "1011", "0110",
            "1000", "0101", "0011", "1111", "1101", "0000", "1110", "1001" )

sBox[1] = ( "1110", "1011", "0010", "1100", "0100", "0111", "1101", "0001",
            "0101", "0000", "1111", "1100", "0011", "1001", "1000", "0110" )

sBox[2] = ( "0100", "0010", "0001", "1011", "1100", "1101", "0111", "1000",
            "1111", "1001", "1100", "0101", "0110", "0011", "0000", "1110" )

sBox[3] = ( "1011", "1000", "1100", "0111", "0001", "1110", "0010", "1101",
            "0110", "1111", "0000", "1001", "1100", "0100", "0101", "0011" )
```

The Round.sv module that performs the following operation: h_out[i]=(h_in[i+2]^sbox_out)<<floor(i/2) is realised by the following function:

```
def compute_round(H, sBox_result, left):

    temp = [None] * 4

    temp[3] = xor(H[0], sBox_result[0])
    temp[2] = xor(H[1], sBox_result[1])
    temp[1] = xor(H[2], sBox_result[2])
    temp[0] = xor(H[3], sBox_result[3])

    round_res = ""

    #implementation of the left rotation
    for i in range (0, 4):
        round_res += temp[3-(i+left)]

    return round_res
```

The function calculates the xor between the i-th position of H and the output of the Sbox, meorises the result into a support vector and then performs the left rotation.

The following function implements the lightHashDES algorithm:

```
#implementation of the light hash DES algorithm as a function
def lightHashDES(test, sBox, H):
    #declaration and initialization of the message from std input
    M = test

    temp_res = [None] * 8
    H_out = [None] * 8

    if test == '':
        return H

    #implementation of the Light Hash DES Algorithm
    for c in M:
        M_six = init_M_six(c)
        sBox_result = compute_sBox(M_six, sBox)

        for r in range (0, 4):
            if r == 0 :
                H_out[7] = compute_round(H[5], sBox_result, math.floor(0/2))
                H_out[6] = compute_round(H[4], sBox_result, math.floor(1/2))
                H_out[5] = compute_round(H[3], sBox_result, math.floor(2/2))
                H_out[4] = compute_round(H[2], sBox_result, math.floor(3/2))
                H_out[3] = compute_round(H[1], sBox_result, math.floor(4/2))
                H_out[2] = compute_round(H[0], sBox_result, math.floor(5/2))
                H_out[1] = compute_round(H[7], sBox_result, math.floor(6/2))
                H_out[0] = compute_round(H[6], sBox_result, math.floor(7/2))
            else:
                H_out[7] = compute_round(temp_res[5], sBox_result, math.floor(0/2))
                H_out[6] = compute_round(temp_res[4], sBox_result, math.floor(1/2))
                H_out[5] = compute_round(temp_res[3], sBox_result, math.floor(2/2))
                H_out[4] = compute_round(temp_res[2], sBox_result, math.floor(3/2))
                H_out[3] = compute_round(temp_res[1], sBox_result, math.floor(4/2))
```

```
                H_out[2] = compute_round(temp_res[0], sBox_result, math.floor(5/2))
                H_out[1] = compute_round(temp_res[7], sBox_result, math.floor(6/2))
                H_out[0] = compute_round(temp_res[6], sBox_result, math.floor(7/2))

            for k in range (0, 8):
                temp_res[k] = H_out[k]

        for k in range (0, 8):
            H[k] = H_out[k]

    return H_out
```

The function takes as input the string whose digest is to be calculated, the sbox and the H vector and returns the digest of the string. The function describes at a high level the behaviour of the lightHashDES.sv module.

Test case

To verify that the model defined in Python correctly emulates the behaviour of the hardware, a series of tests were performed. To be sure that the behaviour is indeed the same as that defined at a low level, the model reads the test vectors from the testbench.txt file, which is the same as that used by the testbench and runs the same tests as the testbench.

The calculated results are saved to the file test_results.txt, for convenience the file is written in binary. The results obtained with the Python model will be compared with the results obtained from the testbench to demonstrate that the model correctly emulates the behaviour of the hardware.

After opening the file containing the test vectors, creating the results file and storing the contents of the testbench.txt file in a list, we start testing the behaviour of the model.

```
f = open("modelsim/tv/testbench.txt", 'r', encoding='utf-8')
out = open("modelsim/tv/test_results.txt", 'wb')                    #create the test file if
it does not exists

for str in f.readlines():
    testbench.append(str.replace("\n", ""))

f.close()
```

The model is initially tested with an empty string and with a string containing only one character. The case of the empty string is shown below.

```
#Test 1: empty string
res_test1 = lightHashDES(testbench[0], sBox, H)

res = toString(res_test1)

#writing in output file the first result
out.write(res.encode())
out.close()

#reopening the file to append the rest
out = open("modelsim/tv/test_results.txt", 'ab')
```

Next, the model takes two equal strings as input and calculates the digest. Since they are equal strings, they will have the same digest.

```
#Test 3: hash of two equal strings (Testing_equal_strings)
H = []
H = init_H()
res_test3 = lightHashDES(testbench[2], sBox, H)

out.write(toString(res_test3).encode("utf-8"))
out.write(b'\n')
```

Finally, tests are carried out with different strings. As they are different strings, the calculated digests are expected to be different. One of these is shown below:

```
#Test 4: hash of two different strings (Testing_equal_strings, Testing_different)
H = []
H = init_H()
res_test5 = lightHashDES(testbench[4], sBox, H)

out.write(toString(res_test5).encode("utf-8"))
out.write(b'\n')
```

The results obtained are then saved in the file test_results.txt.

```
out.write(toString(res_test6).encode("utf-8"))
out.write(b'\n')
out.close()
```

As described in the next section, the results obtained at a high level are cross-referenced with the results obtained from the tests carried out in Modelsim to check the correctness of the behaviour of the model realised in Python. The figure below shows how the results obtained in python and the results obtained with the testbench are crossed. Please refer to the next section for a more detailed description.

```
***TEST SAME MESSAGE SAME HASH START***

Digest of the test_string: 00011110011101100111110011001111
Digest of the test_string python: 00011110011101100111110011001111

test result: Successful: the digest is the same as that obtained with python
Digest of the test_string: 00011110011101100111110011001111
Digest of the test_string python: 00011110011101100111110011001111

test result: Successful: the digest is the same as that obtained with python
test result: Successful: same strings have the same digest
***TEST SAME MESSAGE SAME HASH END***

***DIFFERENT MESSAGE DIFFERENT HASH BEGIN***

Digest of the test_string_1: 00011110011101100111110011001111
Digest of the test_string: 01001011011101101101011000110000
Digest of the test_string python: 01001011011101101101011000110000

test result: Successful: the digest is the same as that obtained with python
test result: Successful: different strings different digest
***DIFFERENT MESSAGE DIFFERENT HASH END***
```

Figure 5 Example testbench execution

Testbench

This section shows the tests performed on the implemented design and briefly discusses the resulting waveforms. The tests were performed using Modelsim.

Test case: empty string

In this case test an empty string is given in input to the module, and the result is expected to be the initial value of H.

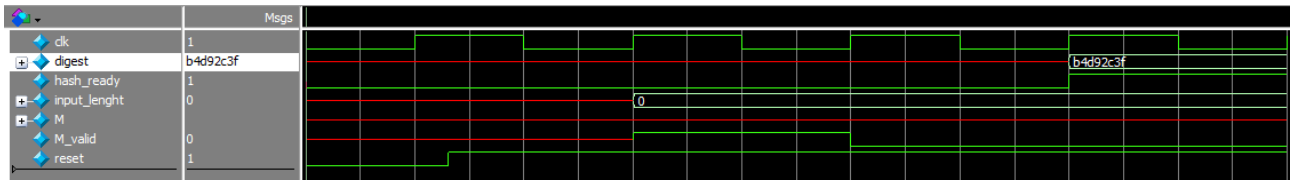


Figure 6 Empty string test case waveform

The above figure shows the waveforms relating to the processing of an empty message. The test takes two clock cycles. When M_valid is set, the computation of the empty string digest begins. In the end, both digest and hash_ready are validated. The resulting digest is the initial vector H, set in the INIT state.

Test case: string containing only one character

The module is tested by giving as input a string containing only one character. We expect that the result will be a known digest that has been previously calculated. The figure below shows the waveform of this test case.

Figure 7 string containing only one character test case waveform

The test takes three clock cycles, and the calculation of the digest begins when M_valid is set to 1. When the module finishes the computation of the digest, hash_ready is set and the digest is validated with the expected result.

Generic test case

In this test case, we work with four strings that have been declared in the test setup.

Two of the strings are the same: test_string_1 = test_string_2 ="Testing_equal_strings"; the third differs from the previous ones by only one byte: test_string_3= " Testing_equal_strongs"; the fourth string used in the tests is: test_string_4=" Testing_equal_string". This test case, therefore, has three sub-cases: one in which equal strings are given as input to the module and the calculated digests are expected to be equal, and two in which different strings are given as input to the module and are expected to produce different digests. The strings test_string_3 and test_string_4 are used to demonstrate that different strings produce different digests.

The figure below shows the waveform of the sub-case of the generic test in which two equal strings are fed into the module.

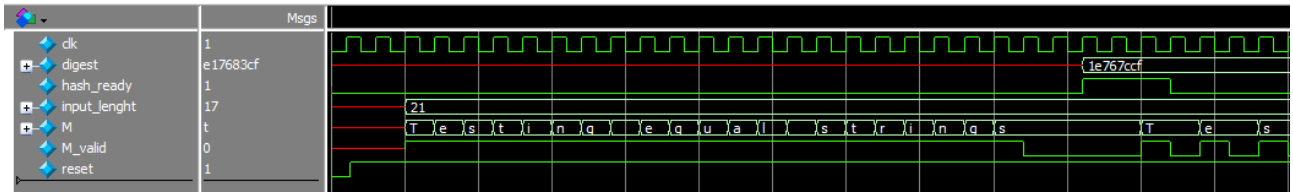


Figure 8 equal strings sub-case waveform of the generic test case

The digests of the strings test_string_1 and test_string_2 have been calculated in two different ways: for the first one byte was given as input for each clock cycle, while for the second one byte was given as input every two clock cycles. This choice was made to demonstrate that the behaviour of the model is the same in the case of both continuous and alternating calculations. As expected, the digest of the two equal strings is the same.

In both cases, the calculation of the digest begins when M_valid is set to 1. Note that in the case of continuous computation, M_valid remains set for the entire execution while in the case of non-continuous computation M_valid is, every two clock cycles, set and reset.

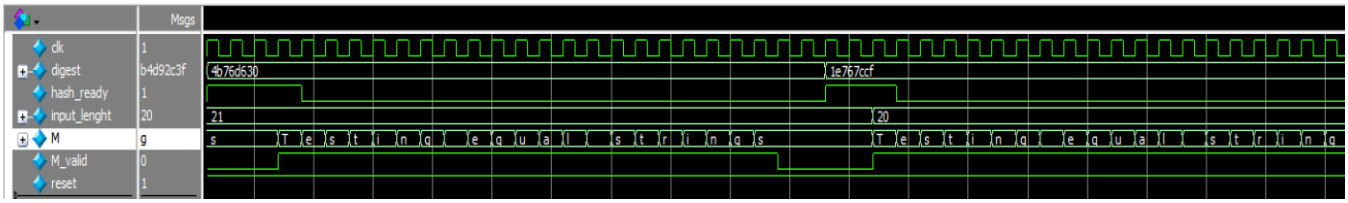


Figure 9 different strings sub-case waveform of the generic test case

The above figure shows the waveform of the test sub-case in which two different strings are fed into the module, as expected the digests produced are different.

Integration between the Python model and testbench

Testbench reads the test vectors from testbech.txt, the same file used by the Python model to read test vectors, calculates the digest of the read string and compares it with the one obtained by running the Python code. This is done to verify that both the high-level model and the hardware model produce the same results from the same test vectors.

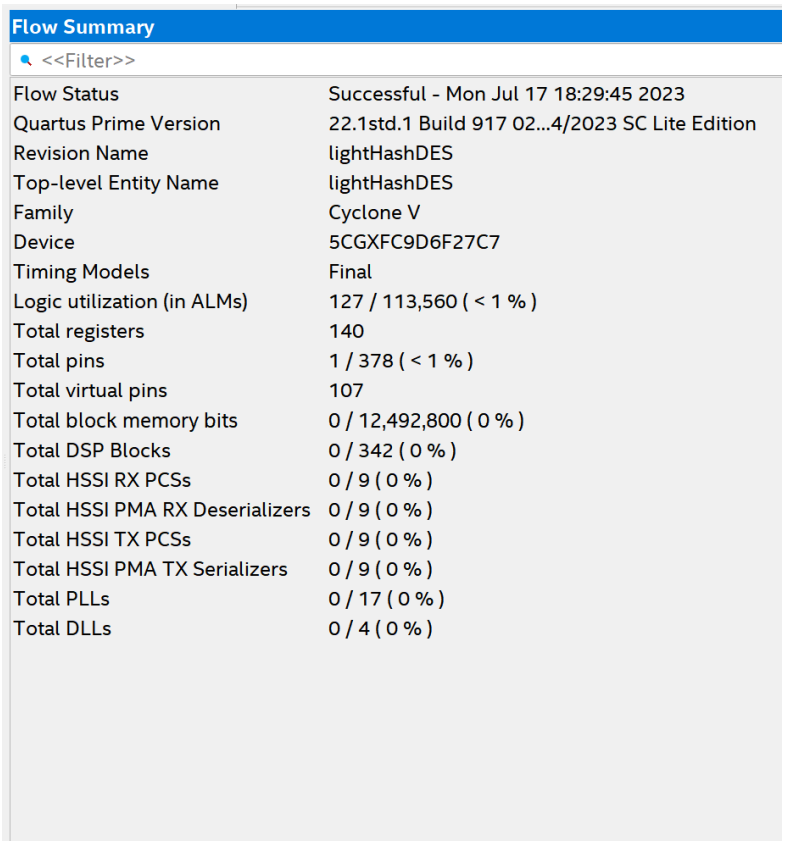
```
***TEST SAME MESSAGE SAME HASH START***

Digest of the test_string: 00011110011101100111110011001111
Digest of the test_string python: 00011110011101100111110011001111

test result: Successful: the digest is the same as that obtained with python
Digest of the test_string: 00011110011101100111110011001111
Digest of the test_string python: 00011110011101100111110011001111

test result: Successful: the digest is the same as that obtained with python
test result: Successful: same strings have the same digest
***TEST SAME MESSAGE SAME HASH END***

***DIFFERENT MESSAGE DIFFERENT HASH BEGIN***

Digest of the test_string_1: 00011110011101100111110011001111
Digest of the test_string: 01001011011101101101011000110000
Digest of the test_string python: 01001011011101101101011000110000

test result: Successful: the digest is the same as that obtained with python
test result: Successful: different strings different digest
***DIFFERENT MESSAGE DIFFERENT HASH END***
```

Figure 10 Example testbench execution

The figure above shows the test case where the strings are the same and the test case where the strings are different. In both cases, the input data is read from the testbench.txt file, processed and then first the digest obtained in hardware is compared to the one calculated in Python and then the digests of the two strings are compared.

Implementation of RTL design on FPGA

The circuit design, the physical design, and the static time analysis were performed using the tool Quartus Prime by Intel. The technology on which the model was chosen to be synthesized is the FPGA 5CGXFC9D6F27C7 of the Cyclone V family by Intel.



Flow Summary

<<Filter>>

Flow Status	Successful - Mon Jul 17 18:29:45 2023
Quartus Prime Version	22.1std.1 Build 917 02...4/2023 SC Lite Edition
Revision Name	lightHashDES
Top-level Entity Name	lightHashDES
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	127 / 113,560 (< 1 %)
Total registers	140
Total pins	1 / 378 (< 1 %)
Total virtual pins	107
Total block memory bits	0 / 12,492,800 (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 11 Flow summary

In the figure is reported the flow summary of Quartus synthesis, It can be seen that the resource usage of target technology is low (< 1%) so the design can be easily integrated on the FPGA board along with other components. Only one physical pin is used, and it is the clock pin, input and output pins are assigned as virtual pins, to increase the overall frequency of the circuit.

The number of virtual pins used is coherent with the number of inputs and outputs of the RTL design generated with Modelsim (107):

- input_lenght: 64-bit
- digest: 32-bit
- M: 8-bit
- M_valid: 1-bit
- reset: 1-bit
- hash_ready: 1 bit

To show how the tool synthesized the developed design, in the next few lines are reported several significant screenshots of the Quartus RTL viewer.

HashIteration module

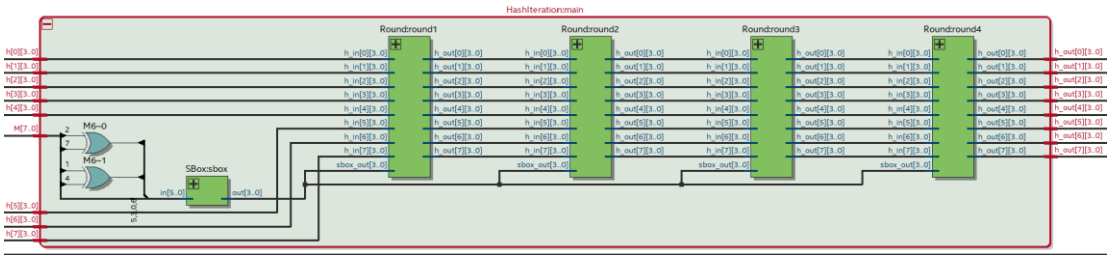


Figure 12 HashIteration module

The figure above shows the HashIteration module, which integrates 4 Round modules, 1 Sbox5 module, and the compression algorithm which computes the signal M_6.

Sbox module

The figure below shows the structure of the Sbox. In particular, the module uses DES's SBox 5. The input signal of the Sbox module is M_6, which drives the four MUX that constitute the Sbox design.

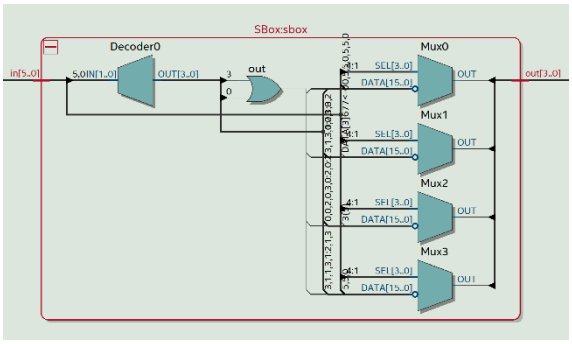


Figure 13 SBox module

Round module

The Round module executes the XOR and circular shift operations over the temporary digest and the Sbox5 output signal. Part of its internal structure is shown in the figure below.

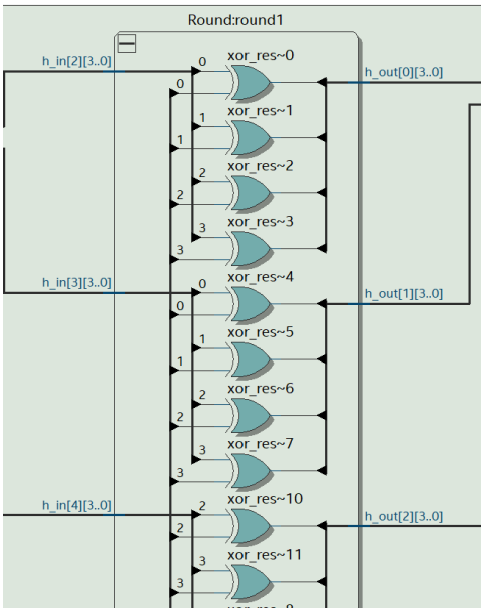


Figure 14 Round module

Static Timing Analysis

A Static Timing Analysis has been performed over the developed model. In the file “SDC1.sdc” are defined the following timing constraints:

- A target clock frequency of 100 MHz was specified, defining a 10 nanoseconds clock period.
- The reset signal was unpaired from the clock signal because the reset signal has to be asynchronous from the clock signal.
- Minimum and maximum delay has been defined for each input and output port, the minimum accepted delay is 10% of the clock period, and the maximum accepted delay is 20% of it.

Below are reported frequency measurements, one obtained during the Slow Test at 0°C and the other during the Slow Test at 85°C.

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	129.48 MHz	129.48 MHz	clk	

Figure 15 Slow 1100mV 0C Fmax Summary

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	129.62 MHz	129.62 MHz	clk	

Figure 16 Slow 1100mV 85C Fmax Summary

As can be seen, the timing requirements are met in both cases. The worst case is at 0°C with a frequency equal to 129.48 MHz, and this is the one that has to be taken into account to deal with the worst possible environmental conditions.