



UNIVERSITÀ DI PISA

Master's Degree in Cybersecurity

Light Hash algorithm (AES S-box based)

Hardware And Embedded Security

Federico Bernacca, Kostantino Prifti

Contents

1	Introduction	3
2	Project specifications	3
3	Implementation	4
3.1	Module	4
3.1.1	Module ports	4
3.1.2	Approach	4
3.1.3	Registers	5
4	Waveform	6
5	Test	7
5.1	Validation test	7
5.2	Hash properties test	8
6	Static timing analysis	9
6.1	Constraint file	9
6.2	Virtual pins	9
6.3	Maximum frequency	9
7	Quartus report	10

1 Introduction

This report follows a top-down approach for the description of the implementation choices, starting from a high-level description of that choices, and then going down more specifically, detailing carefully. In particular, this document is organized as it follows:

- **Description**
- **Implementation**
- **Waveform**
- **Test**
- **Static timing analysis**
- **Quartus report**

2 Project specifications

In cryptography, the Advanced Encryption Standard (AES) is a symmetric key block cipher algorithm. The AES encryption algorithm is composed of 4 transformations which are SubBytes, ShiftRows, MixColumns and AddRoundKey.

The project consists in to design a custom hash algorithm to calculate the digest.

This algorithm uses a function that corresponds exactly to one of the functions of the AES algorithm: the S-box (which is a subpart of the SubBytes transformation).

The hash function produces a fixed output of 64 bits H obtained as a concatenation of 8 bytes $H[i]$, from $H[0]$ up to $H[7]$.

For each message computed, the hash H , which name is *digest*, must be initialized as follows:

	$H[0]$	$H[1]$	$H[2]$	$H[3]$	$H[4]$	$H[5]$	$H[6]$	$H[7]$
Init. value	8'h34	8'h55	8'h0F	8'h14	8'hDA	8'hC0	8'h2B	8'hEE

Figure 1: Initial hash value

Once the digest is initialized, for each byte of the message whose hash must be computed, the following operations are done:

```
for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 8; j++) {
        H[i] = S((H[(i + 2) mod 8] XOR M) << j);
    }
}
```

where:

- **mod n**: is the modulus operator of n.
- **XOR**: is the XOR operator.
- **<<**: is the circular shift operator of n bits.
- **S()**: is the S-box transformation of the AES algorithm.

Using the pseudocode presented above, it has been described at a high-level how the computation of the hash function works.

In the **Section 3** the focus is on the real implementation of the module.

3 Implementation

The schematic is the following:

3.1 Module

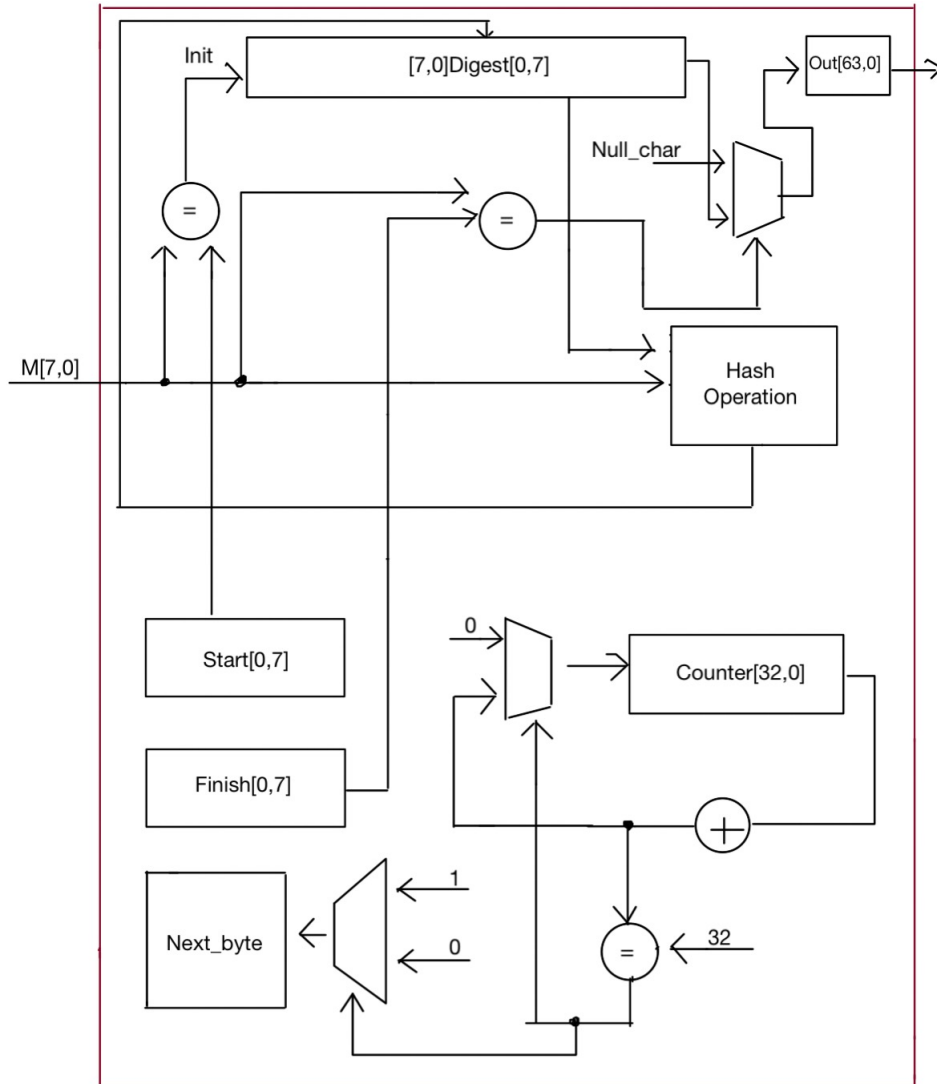


Figure 2: High-level diagram block

3.1.1 Module ports

```
module hash(
    input [7:0] m,
    input m_valid,
    input clk,
    input reset_l,
    output reg hash_ready,
    output reg [63:0] out
);
```

There are several input ports and output ports:

- **input [7:0] m:** through this port the module receives a single byte of an arbitrarily long message;
- **input m_valid:** port that signals when the incoming byte is valid and stable;
- **output reg hash_ready:** output port indicating that the message hash is computed;
- **output reg [63:0] out:** the hash of the message is returned through this port.

3.1.2 Approach

Before presenting the two types of approach considered, it is good to define the concept of round.

A round of the algorithm corresponds to the execution of a single loop of the outermost for of the algorithm seen above. More precisely, eight operations are performed on the digest in a round, one for each byte of the digest.

The eight operations in question correspond to the innermost for loop of the algorithm seen above. So 32 rounds are required to calculate the digest of a single byte.

- **Approach 1:** all the rounds necessary to calculate the digest of a single byte of the message to be encrypted are performed in a single clock cycle.
- **Approach 2:** each round of the byte of the message to be encrypted is performed in different clock cycles.

The second approach was implemented because the first one had a latency time of one clock cycle, instead of 32 cycles of the second approach.

This choice therefore allows to have a significantly shorter critical path and therefore to obtain a higher clock frequency.

In the following section it is described the structure of the module:

3.1.3 Registers

There are different registers in the module:

- **start:** 8-bit constant composed of the 11111111 bits.
Before starting a new message computation, a sequence of these bits is passed to the module; the module thus prepares to receive, as the next byte, the first byte of the message.
A comparison will then be made between **start** (local to the module) and the incoming byte on the input port **m**.
- **finish:** 8-bit constant composed of the 00000000 bits, dual of start.
At the end of an input message, these 8-bit sequence will be passed and compared with the local copy of the module, in case of match, the signal **hash_ready** is raised and the computed digest **out** is output.
- **counter_enable:** 1-bit register that indicates whether the module can correctly perform the 32 iterations necessary for the calculation of the digest.
This signal is raised when the incoming byte is valid and stable.
It returns to 0 when all 32 iterations of the hash calculation on the i-th byte have been executed.
- **count:** 32-bit register that keeps the number of computations performed on the i-th byte.
It increases by 1 in every clock cycle after the $H[i]$ computation is performed; starting from 1 arriving to 32.
- **next_byte:** 1-bit register indicating when the module is ready to receive a new byte of the input message, set to 1 at the end of the 32 iterations over the previous byte.
- **digest:** register that holds the temporary bytes of the computed digest up to the current byte **m** of the message.

4 Waveform

Several explanatory waveforms of the behavior of the module are reported.

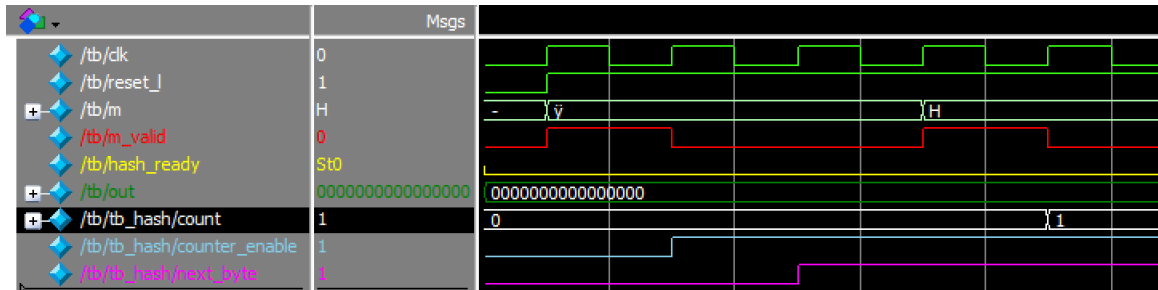


Figure 3: Beginning of digest computation

Figure 4 shows the waveform of the behavior of the module when the simulation begins: a reset is performed that restores all the registers. At the next front positive clock the initial byte 11111111 (\ddot{y} in ASCII) is passed: it signals the beginning of a message. The **m_valid** signal indicates that the byte **m** of the input message is valid and stable.

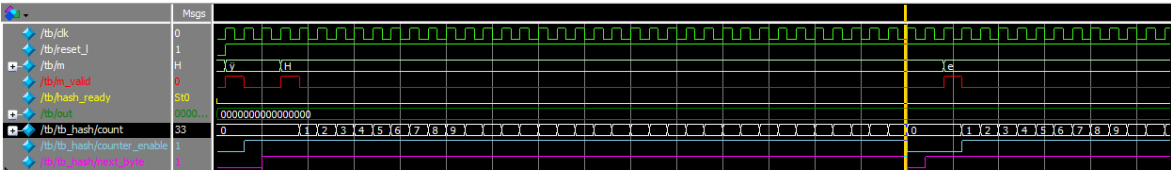


Figure 4: One byte digest computation

The signal **m_valid** is raised whenever the input byte **m** on the input port is valid and stable, at this point the **counter_enable** signal is raised, which indicates the possibility of starting to calculate the digest on byte **m**. The signal **m_valid** returns to 0 until the 32 iterations over the input byte **m** are done. One for each clock cycle. With each subsequent clock cycle, the value contained in the **counter** register will be incremented by one, until it reaches 33. For the duration of the computation on byte **m**, the signal **next_byte** remains high, necessary to make the testbench understand when to pass the next byte of the message. These operations are performed for all bytes of the input message.

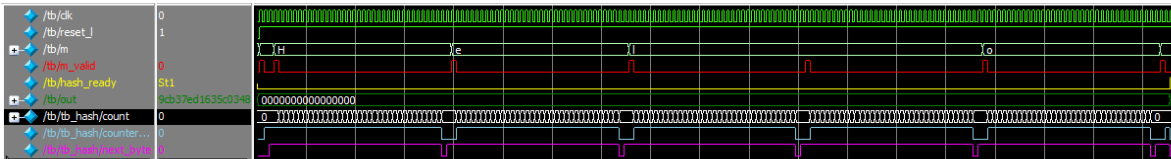


Figure 5: Digest calculated

The signal **hash_ready** remains low until all the digest has been calculated. The output value **out** remains null as long as **hash_ready** is low. At the end of the computation, **hash_ready** is set to 1 and the calculated digest is ready on the output port **out**.

5 Test

To test the functionality of the module, it has been carried two different tests, located in the test file. In particular, two different types of tests were performed, described in **Section 5.1** and **Section 5.2**.

5.1 Validation test

The first test was about in reading from different test vector files provided by third parties several messages with their related digest.
The test consisted in reading these messages, calculating the digest, comparing it with the expected one.
All the tests carried out resulted in a correct match between the expected digest and the calculated digest.

Below are shown a schema of how the test works and a waveform with the expected and calculated output of some of the test vectors provided.

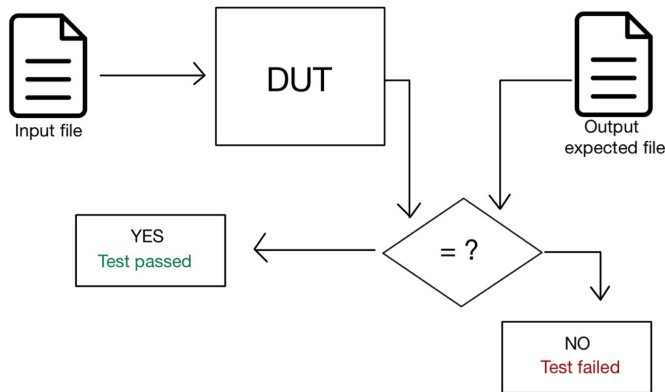


Figure 6: Validation test schema

This is a schema representing the functionality of this test, where DUT means Device Under Testing, that is the module implemented.

First test vector was:

```
Messaggio in chiaro di prova // message
a4f31ba062ddf155 // expected digest
```

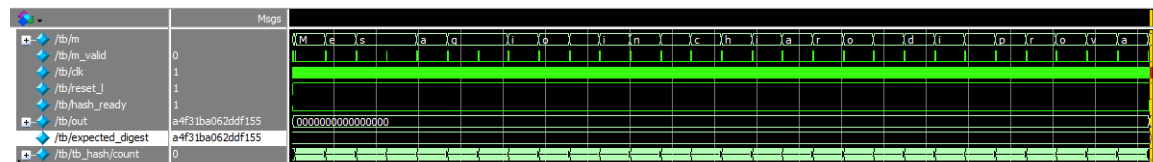


Figure 7: Test vector 1

It is possible to observe how the calculated digest **out** and the expected digests **expected_digest** are equals.

Second test vector was:

```
abcdefghijklmnopqrstuvwxyz // message
dcdfac61d4981831 // expected digest
```

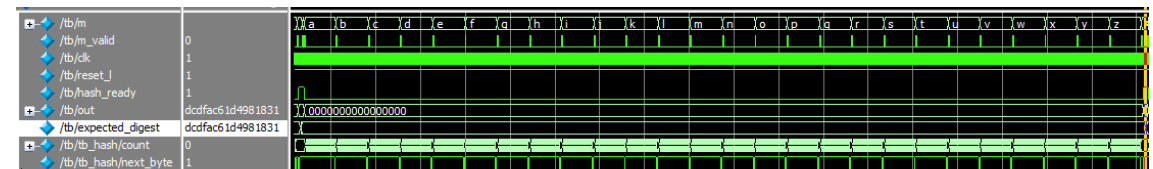


Figure 8: Test vector 2

It is possible to observe how the calculated digest **out** and the expected digests **expected_digest** are equals.

All the given test vector are located in

```
light_hash\modelsim\reference_tv
```

5.2 Hash properties test

The other type of test was designed to check two “properties” of hash functions:

1. Same messages generate equal digests.
2. Identical messages, but different even for a single character, generate completely different digests.

In particular it have been passed 4 messages in sequence:

```
string s0 = "Hello";  
string s1 = "Hello";  
string s2 = "World123456789";  
string s3 = "World123456780";
```

Where the first two, being identical, should give out the same digest:

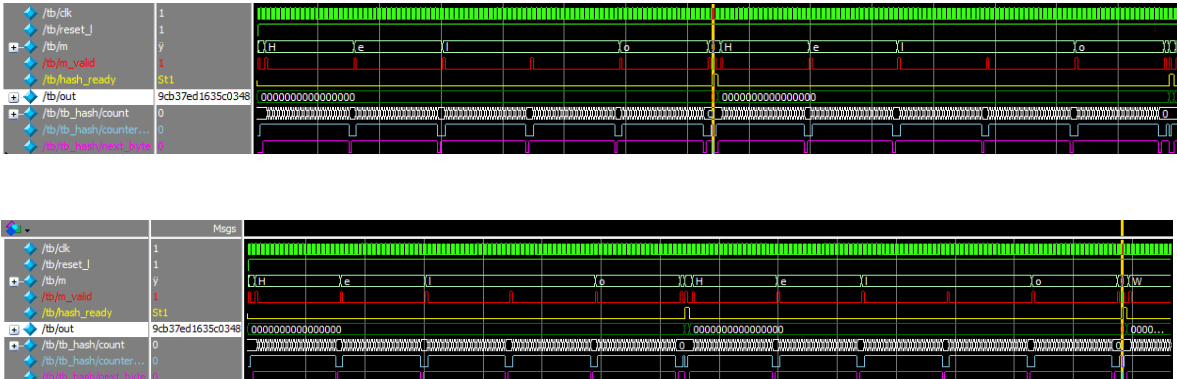


Figure 9: Digest of two identical messages

It is possible to observe from the two cursors how the calculated digests are effective the same. The other two strings are identical except for one character; the expected digests should be completely different:

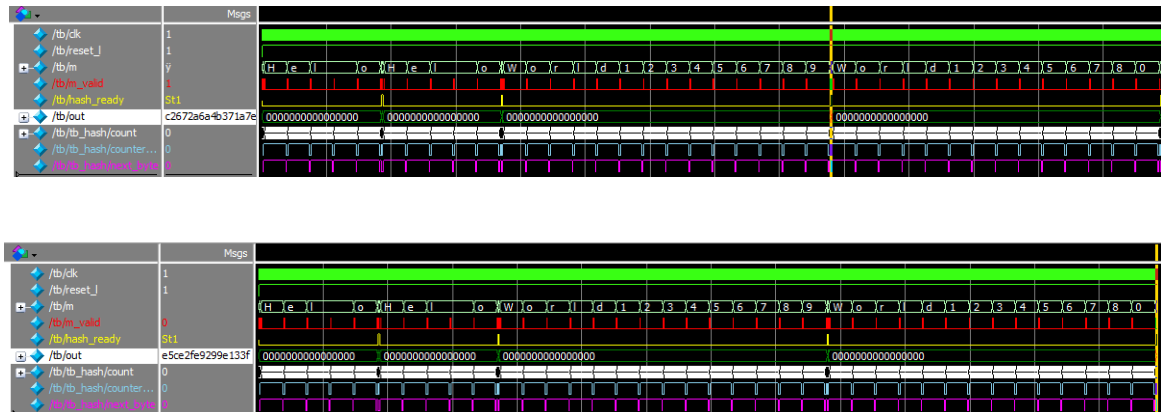


Figure 10: Digest of two identical messages except for one character

It is possible to observe from the two cursors how the calculated digests are completely different.

6 Static timing analysis

Static time analysis is a significant part of the design flow, because according to the report provided by this process, it is possible to verify whether the designed circuit verifies the timing constraints, such as the clock frequency.

Providing the `.sdc` file, the synthesis engine, in addition to creating the netlist (logical synthesis) and mapping it to the technology (fitter), also verifies the time constraints.

6.1 Constraint file

```
// hash.sdc
create_clock -name clk -period 10 [get_ports clk]
set_false_path -from [get_ports reset_l] -to [get_clocks clk]
set_input_delay -min 1 -clock [get_clocks clk] [get_ports {m[*] m_valid reset_l}]
set_input_delay -max 2 -clock [get_clocks clk] [get_ports {m[*] m_valid reset_l}]
set_output_delay -min 1 -clock [get_clocks clk] [get_ports {hash_ready out[*]}]
set_output_delay -max 2 -clock [get_clocks clk] [get_ports {hash_ready out[*]}]
```

The constraint file, as it is possible to see above, is composed of a few lines of code. In the first line we create the clock object specifying the period of 10 nanoseconds.

Another very significant directive is the one concerning the reset: in the system the reset is asynchronous, this means that when the reset is raised, there is no need to respect the time constraints; this translates into a false path that specifies to the synthesis engine not to go to verify that the paths related to the reset signal respect the constraints of the clock.

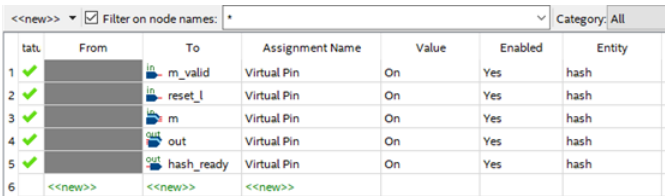
The last four lines of code concern the time constraints on the input and output ports of the module.

As it is possible to see in the code, for the input and output ports a minimum delay and a maximum delay are specified: as minimum delay values we used 10% of the clock period, while as a maximum delay 20% of the clock period.

6.2 Virtual pins

To make static time analysis correct, it is specified, to the system engine, the input and output ports as virtual pins.

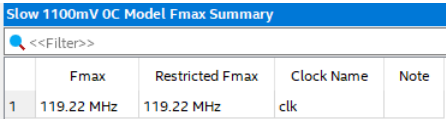
There are used virtual pins since this project deals with a subsection of a larger project so in this phase it is difficult to place precisely the input and output ports, so through the virtual pins it is telling the synthesis engine to map the input and output ports as a pin on the internal logic of the FPGA.



	tatu	From	To	Assignment Name	Value	Enabled	Entity
1	✓		in m_valid	Virtual Pin	On	Yes	hash
2	✓		in reset_l	Virtual Pin	On	Yes	hash
3	✓		in m	Virtual Pin	On	Yes	hash
4	✓		out out	Virtual Pin	On	Yes	hash
5	✓		out hash_ready	Virtual Pin	On	Yes	hash
6	<<new>>	<<new>>	<<new>>				

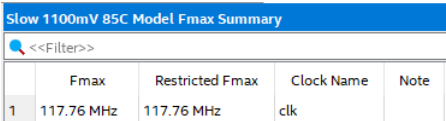
Figure 11: Virtual pins

6.3 Maximum frequency



Slow 1100mV 0C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	119.22 MHz	119.22 MHz	clk	

Figure 12: Slow 1100mV 85C Model



Slow 1100mV 85C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	117.76 MHz	117.76 MHz	clk	

Figure 13: Slow 1100mV 0C Model

As we can see from the images, having declared the virtual pins, the module is able to support a maximum frequency of 117.76 MHz.

Between the two maximum frequencies shown in figures 12 and 13 we must consider the lower one, it is the frequency of the worst case. This ensures that the module works in all circumstances.

7 Quartus report

In conclusion it is illustrated the complete report provided by Quartus. In the following report, there isn't any error message so the system is constrained from a time point of view in an adequate way.

Table of Contents		Flow Summary	
Flow Summary		<<Filter>>	
Flow Settings		Flow Status	Successful - Mon Jul 19 20:07:23 202
Flow Non-Default Global Set		Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite E
Flow Elapsed Time		Revision Name	hash_light
Flow OS Summary		Top-level Entity Name	hash
Flow Log		Family	Cyclone V
> Analysis & Synthesis		Device	5CGXFC9D6F27C7
> Fitter		Timing Models	Final
Flow Messages		Logic utilization (in ALMs)	971 / 113,560 (< 1 %)
Flow Suppressed Messages		Total registers	190
> Timing Analyzer		Total pins	1 / 378 (< 1 %)
		Total virtual pins	75
		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 14: Quartus report

From the report it is possible to see that the FPGA chosen is from the Cyclone V family. The device used for synthesis is 5CGXFC9D6F27C7 as show by report in Figure 14. The logic resources used are less than 1%. This indicates that this type of FPGA is not the best hardware solution on which to implement this module, because there would have a waste of logical resources. The total pin number is 1, and it is the clock pin (**clk**). The other input and output pins have been considered as virtual, they do not appear in the total pin count.