University of Pisa

Computer Engineering Department

M.Sc. in Cybersecurity

Hardware And Embedded Security Course

Project Report:

Light Hash algorithm

(AES S-box based)

Alessandro Niccolini | Giacomo Vitangeli

Academic year 2021-2022

1 Specification analysis

1.1 Introduction

This project has the goal to design an hash module based on the S-box using AES algorithm. Given an input message of N bit, the hash module has to provide an output digest of 64 bits.
The algorithm starts the computation from an 8 bytes vector called H.
The input message is splitted into a stream of single bytes before passing them to the module, every byte of the input message is combined with the previous in a specific way, following AES, in order to produce a temporary digest, while the input bytes stream of the message is completed. Finally the module produce in output the digest associated to the input message.

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

(a) Initial vector H

$$for\ (r = 0;\ r < 32;\ r++)$$
$$for(i = 0; i < 8; i++)$$
$$H[i] = S\big((H[(i+2)\ mod\ 8] \oplus M) \ll i\big)$$

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.
$S(\ )$	is the S-box transformation of AES algorithm, that works over a byte.

(b) Algorithm for each byte M

2 Block diagram and design choices

2.1 Design architecture

In the figure 2.1, it's show the high level design of this project. In details, there are two input register, *message_ valid and state*, respectively for check if the *message_ byte* is valid and the second for represent the current state. The possible stare are:

1. head, indicate that the first byte of the message will arrive at the clock later;
2. tail, point out that the last byte of the message is arrived;
3. message, specify that a byte of the message is reading.

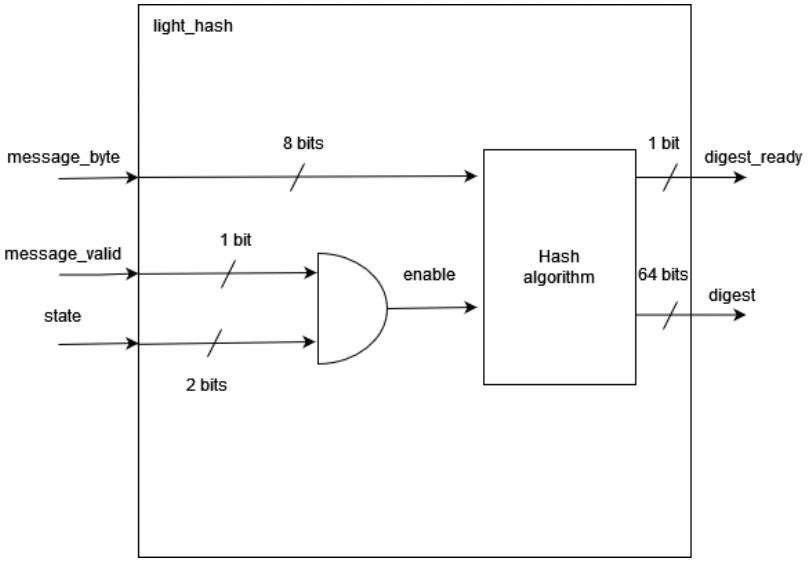


Figure 2.1: block diagram

The combinational always have a **case switch** and inside is checked the value of state, so every head state is reset the register for compute the correct digest, in the tail state a register that tell to the sequential always that the computation is finished. And finally, the message state indicate that the message byte is ready to be insert in the computation stage.
In this section are described the input and output registers that are necessary to build the hash function.
The in/out lines used are the following:

```
module light_hash (
    input               clk
    ,input              rst_n
    ,input      [7:0]   message_byte
    ,input              message_valid
    ,input      [1:0]   state
    ,output reg [63:0]  digest
    ,output reg         digest_ready
    );
```

Figure 2.2: Module of the Light Hash

- input *clk*: the clock that allow to synchronize all other ports;
- input rst_n: the asynchronous signal reset that allow to trigger other signals without waiting the clock;
- input [7:0] *message_byte*: a single byte of the message, allow to use messages of any size;
- input *message_valid*: single bit that tell to the module when the input message byte is ready and stable;
- input *state*: 2 bits register that stores the current state of the processing, specify if the register *digest_tmp* should be initialize to default value (figure 1.1a) or update the *digest_tmp* processing a new message byte (computation phase) or send out the final digest;
- output reg [63:0] *digest*: 64 bit register that collect the value of the final digest, that's the result of the computation phase (figure 1.1b) on each byte of the input message;
- output reg *digest_ready*: single bit that tells if the *digest* register is ready and stable in order to read its 64 bit value.

Focusing on the figure 2.3 we can see the **Finite State Machine** with three states that represents the stages of the light hash module with 2 bit register *state*.
The machine start with input $state = HEAD$ entering in the **initialization stage** all the registers are initialized to default values (e.g. $digest\_tmp = restore\_digest$); Than if $state = MESSAGE$ we enter in **computation stage**, it means that the module has received a message byte and it's processed in accord with the algorithm in figure 1.1b, so the *digest_tmp* value

is updated. If the computation stage continue to receive other message bytes it continue to perform the digest computation.
Finally when $state = TAIL$ the module enter in the **output stage**, means that digest processing is finished and the 64 bit final output *digest* is sent out. If there are other messages in input to perform, *state* has been set again on *head* value, initializing the module registers and preparing it to receive a new message.

At begin the input byte is give to a *for*, within which are performed different operation:

- mod n, the module operation by n;
- $\oplus$, the XOR operation;
- $\ll$, the left circular shift by n bits;
- $S - box()$, the S-box transformation of the AES algorithm.

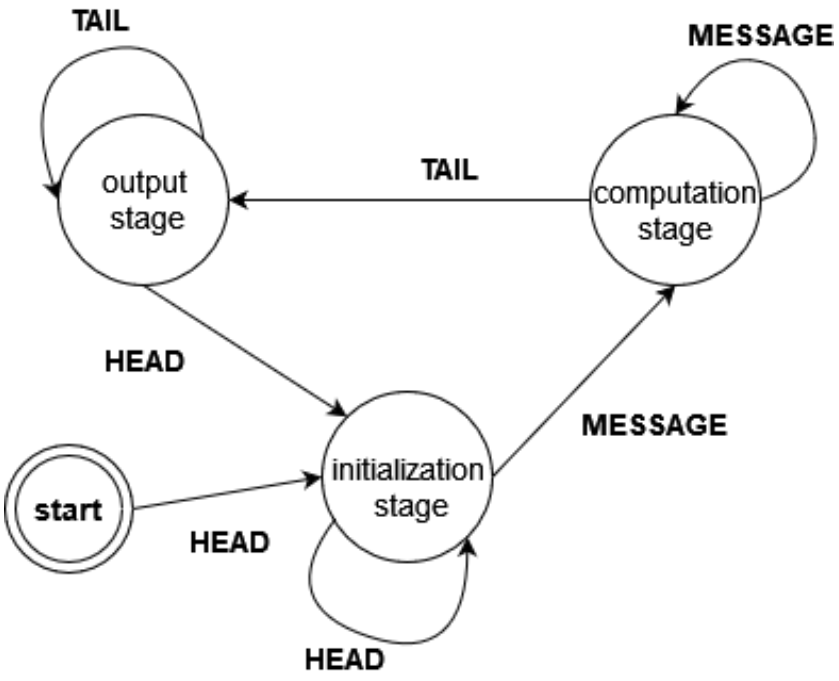
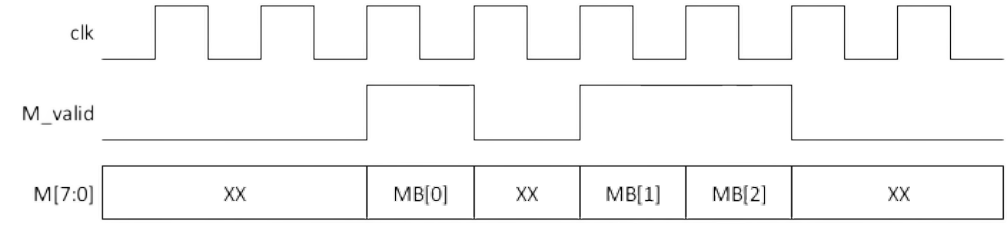
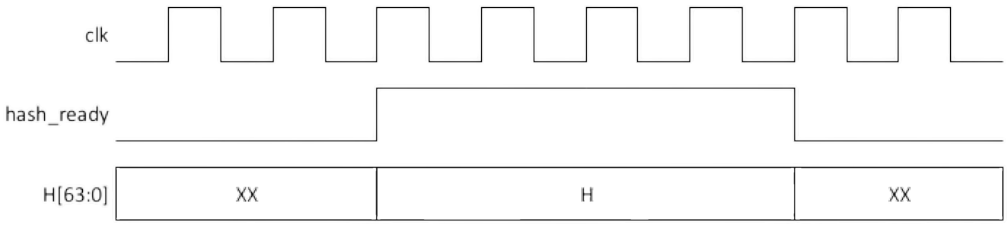


Figure 2.3: Finite state machine of the stages of each input string



(a) Valid byte in input



(b) Valid digest in output

Figure 2.4: Valid signal

Before reading the input *message_byte* the signal must be valid and stable, in order to guarantee this constraint we use the signal *message_valid*, when it's enabled the module can read the input *message_byte* to process it (figure 2.4a).

As we can see in figure 2.4b, to read correctly the final output *digest* is necessary to have the signal valid and stable, to guarantee this constraint we use the signal *digest_ready*, otherwise the read of the digest could be incorrect.

3 Expected waveforms and Testbench

3.1 Expected waveforms

In this Section are analyzed some waveforms extracted from Modelsim, are displayed the input and output signals about different and relevant registers, in order to help the run-time behavior analysis of this Light Hash hardware module.
In the figure 3.1 we can see the full wide waveform of all the test performed, to appreciate the behavior and how the input signals are related with the output ones, we will focus on some relevant tests.

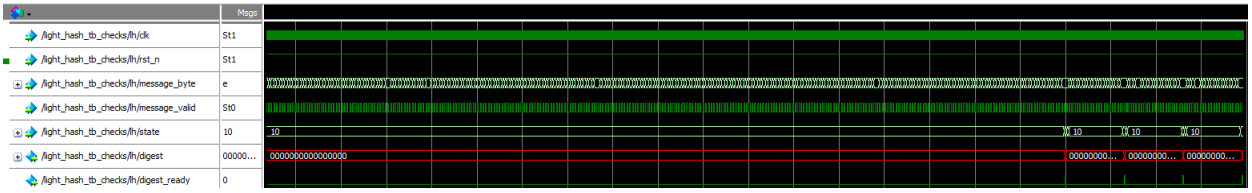
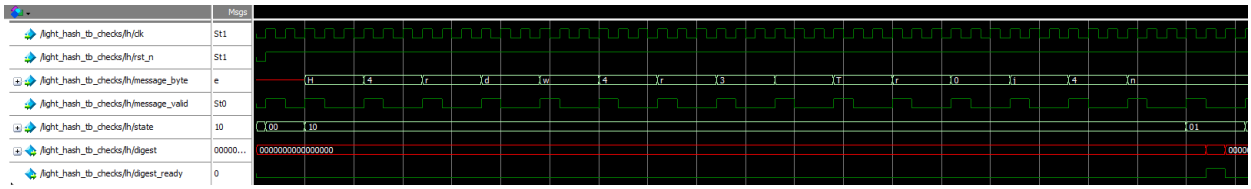


Figure 3.1: Full Waveform of the Battery Test

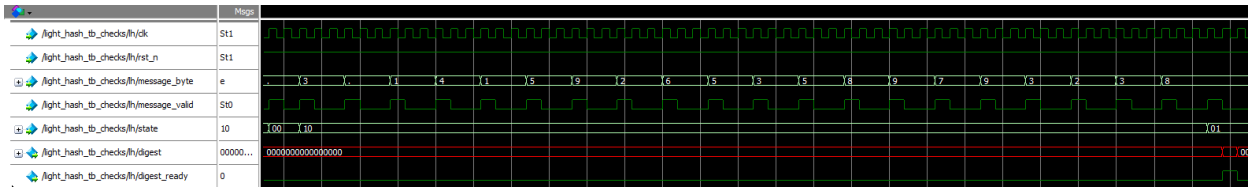
The Battery Test is composed by three messages in string type:

- *H4rdw4r3_Tr0j4n*
- *Nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura, ch la diritta via era smarrita. Ahi quanto a dir qual era cosa dura esta selva selvaggia e aspra e forte che nel pensier rinova la paura! Tant' è amara che poco è più morte; ma per trattar del ben ch'i' vi trovai, dir de l'altre cose ch'i' v'ho scorte.*
- *3.141592653589793238*

First of all we want to prove that at different messages with different length as input, corresponds different digest with same length (64 bit). The two first input messages are significantly different in terms of length but the related digests produced are different with the same length, so the property of the same size of output digest is respected.
The input message can be any dimension, it will impact only on the time necessary to perform the analysis. As a matter of fact, in the first waveform that show all the tests, is clear that the second string is the longest in terms of time and size.



(a) Waveform of the string *H4rdw4r3_Tr0j4n*



(b) Waveform of the string *3.141592653589793238*

Figure 3.2: Waveforms

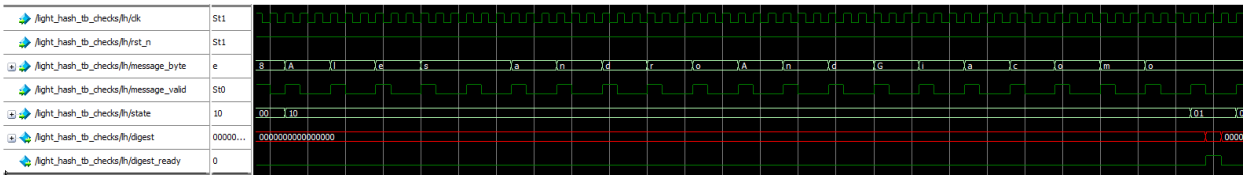
Another property that has been analyze, is the digest comparison between two string identically, except for one character. In particular, the results are:

- the string *AlessandroAndGiacomo* produce the digest *e19e79abcdf021f1*;
- in the other hand, the string *AlessandroandGiacomo* produce the digest *48f63b14b5c40a5a*.

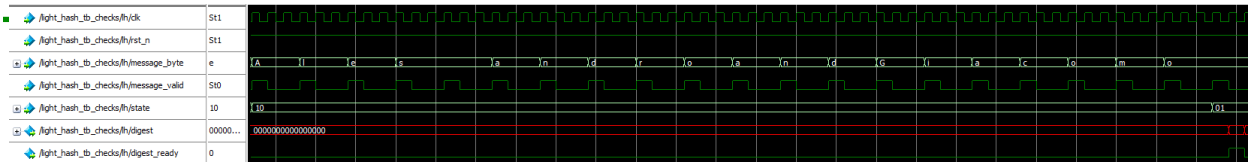
The result shows that also in case of two very similar strings the generated digests are completely different each others.

3.2 Testbench

Testbench is characterized by an approach not complete, because testing all the possible input was computationally hard, but the tested input are focused on particular cases that the light hash module could process in order to ensure the expected behavior in most cases.
In the implementation of the testbench are declared the linking to the module in order to call it later. The messages are defined in string format to test them as input for the module.
For each message the testbench run a process that for each byte (ASCII character) in the string pass the value to the module that will process the temporary digest until the module will receive the last byte of the string, identified by the register *state*, that has been set to **tail** value.



(a) Waveform of the string *AlessandroAndGiacomo*



(b) Waveform of the string *AlessandroandGiacomo*

Figure 3.3: Waveforms

The module is capable to produce the final digest of the message, so the testbench compare the output digest with a pre-computed digest, in order to check the correctness and the expected behavior of the module.

```
1  //input message to test
2  string m0 = "H4rdw4r3_Tr0j4n";
3  //expected output digest
4  string d0_expected = "5aecbf4f5fe467bc";
5
6  //set the state of the machine to HEAD value
7  fork
8      @(posedge clk) message_valid = 1'b1;
9      @(posedge clk) state = HEAD;
10 join
11 @(posedge clk) message_valid = 1'b0;
12
13
14 foreach(m0[j]) begin
15     //set the state of the machine to MESSAGE value
16     fork
17         @(posedge clk) message_valid = 1'b1;
18         @(posedge clk) state = MESSAGE;
19         @(posedge clk) message_byte = m0[j];
20     join
21     @(posedge clk) message_valid = 1'b0;
22     // wait until the signal next_byte is set to 1
23     wait(!lh.next_byte) @(posedge clk);
```

```
24 end
25
26 //set the state of the machine to TAIL value
27 fork
28     @(posedge clk) message_valid = 1'b1;
29     @(posedge clk) state = TAIL;
30 join
31 @(posedge clk) message_valid = 1'b0;
32 @(posedge clk);
33
34 //convert bin final digest into a string to display its value
35 digest_str = $sformatf('%0h', digest);
36 $display('Digest string 0: %s', digest_str);
37 if(digest_str == d0_expected)
38     $display('Test ok, the digest is equal
39             to pre-calculated one.');
40 else
41     $display('Test failed, the digest is different
42             from pre-calculated one: %s', d0_expected);
```

```
VSIM 3> run -all
# Digest string 0: 5aecbf4f5fe467bc
# Test ok, the digest is equal to the pre-calculate.
```

Figure 3.4: Modelsim Transcript output

In the testbench code above is shown the test of the first string of the Battery Test (*H4rdw4r3_ Tr0j4n*), followed by it's output (figure 3.4) produced in *Modelsim Transcript*.
As we can see in the output of the first test, the output digest is equivalent with pre-calculated one, so that test is passed successfully.

4 RTL Design and results

4.1 Implementation of RTL design on FPGA and results

In figure 4.1 is represented the Register Transfer Level (RTL) design of the light hash module, it's an abstraction used to describe digital components design. It's the fundamental abstraction currently used to describe electrical systems and frequently acts as the choice model during the design and verification process.
Hardware Description Language (HDL), such as Verilog or VHDL, are typically used to define the RTL design.

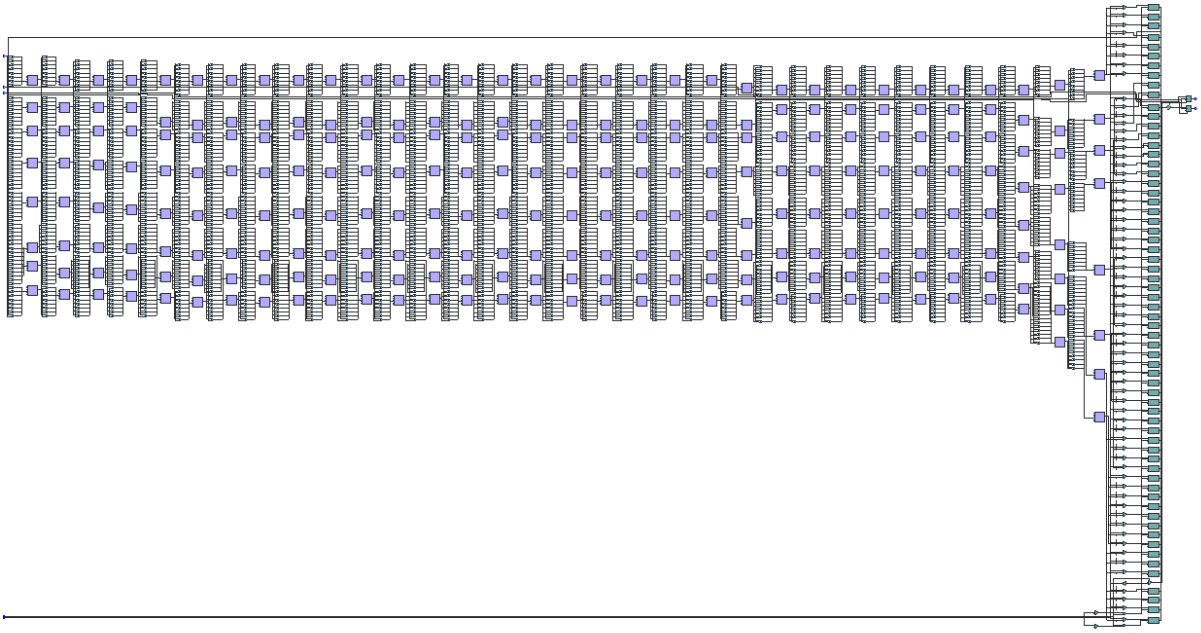


Figure 4.1: RTL view of the project

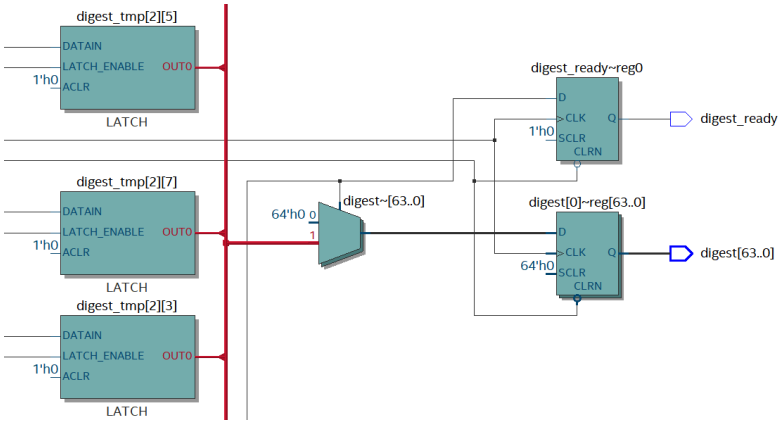


Figure 4.2: Digest register for 64-bit output

Analysis & Synthesis Summary

<<Filter>>

Analysis & Synthesis Status	Successful - Fri Jul 08 18:11:21 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	LH_AES
Top-level Entity Name	light_hash
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	65
Total pins	0
Total virtual pins	78
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

(a) Analysis & Synthesis Summary

Fitter Summary

<<Filter>>

Fitter Status	Successful - Fri Jul 08 18:14:37 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	LH_AES
Top-level Entity Name	light_hash
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	10,587 / 113,560 (9 %)
Total registers	65
Total pins	0 / 378 (0 %)
Total virtual pins	78
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 %)

(b) Fitter Summary

Figure 4.3: Summary

In figure 4.2 we can see the 64bit register used to push out the processed results, this register is used to collect the final digest produced, in Q line there is the output digest. Whereas the input D is link with a mux that follow the logic of the SystemVerilog file. Until the entire message is processed the final digest has been set to 64bit zero value ($64'h0$), otherwise the final value of the digest take the value of *digest_ tmp* (temporary register).

Fitter summary in figure 4.3b makes it clear that the implemented module consumed less than 9% of the total logic resources available. This finding shows that this kind of implementation is not the ideal for the Light Hash module, because there would be some logical resource waste.

5 Static Timing Analysis (STA)

5.1 Static Timing Analysis (STA)

By examining all potential avenues for timing violations, static timing analysis (STA) is a technique for evaluating the timing performance of a design. In STA, a design is divided into timing paths, the signal propagation delay along each path is calculated, and timing restrictions inside the design and at the input/output interface are checked for violations.
However STA can only examine a circuit design's timing, it cannot evaluate its functionality.

5.1.1 STA function

STA divides the design into time pathways before doing timing analysis. The following components are found along each timing path:

- *Starpoint* → the beginning of a timing path where data is launched by a clock edge or where data availability requirements are imposed. A register clock pin or an input port must be present at each startpoint.
- *Combinational logic network* → elements without internal states or memories. Flip-flops, latches, registers, and RAM are not permitted in combinational logic, however it can have AND, OR, XOR, and inverter elements.
- *Endpoint* → the point in a timing path where data must be available at a certain time or where a clock edge captures the data. Every endpoint must be either an output port or a pin for registering data.

```
create_clock -name clk -period 9.1 [get_ports clk]
set_false_path -from [get_ports rst_n] -to [get_clocks clk]
set_input_delay -min 1 -clock [get_clocks clk] [get_ports {message_byte[*] message_valid state}]
set_input_delay -max 2 -clock [get_clocks clk] [get_ports {message_byte[*] message_valid state}]
set_output_delay -min 1 -clock [get_clocks clk] [get_ports {digest_ready digest[*]}]
set_output_delay -max 2 -clock [get_clocks clk] [get_ports {digest_ready digest[*]}]
```

Figure 5.1: Constraint file for Static Time Analysis

The figure 5.1 shows the constraint file used for static time analysis. The first constraint indicate the clock period to consider during the timing analysis, so decreasing that value of course the clock frequency will increase. In this section we have manipulated the clock period parameter in order to find

the limit, that was 9.1ns (corresponding to 109.89MHz in frequency) for our machine, to be compliant with the timing constraints.

	Status	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	Ok		digest_ready	Virtual Pin	On	Yes	light_hash		
2	Ok		message_valid	Virtual Pin	On	Yes	light_hash		
3	Ok		digest	Virtual Pin	On	Yes	light_hash		
4	Ok		message_byte	Virtual Pin	On	Yes	light_hash		
5	Ok		state	Virtual Pin	On	Yes	light_hash		
6	Ok		clk	Virtual Pin	On	Yes	light_hash		
7	Ok		rst_n	Virtual Pin	On	Yes	light_hash		

Figure 5.2: Virtual pins

The figure 5.2 represent the fact that the module is specified as virtual. In this way the module can be used in a more complex system and integrated perfectly.

5.2 Frequency

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	110.33 MHz	110.33 MHz	clk	

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	111.38 MHz	111.38 MHz	clk	

Regardless of the user-specified clock periods, this panel reports F_{MAX} for each clock in the design. Only pathways where the source and destination registers or ports are driven by the same clock are considered for computing F_{MAX}.
Different clock paths, including those of created clocks, are disregarded. F_{MAX} is calculated for pathways between a clock and its inversion as if the rising and falling edges were scaled in tandem with F_{MAX}, preserving the duty cycle (expressed as a percentage). For sign-off analysis, Altera advises constantly using clock restrictions and other slack reports.