UNIVERSITÀ DI PISA

DEPARTMENT OF COMPUTER ENGINEERING

MASTER DEGREE COURSE IN CYBERSECURITY

PROJECT REPORT

FULL-HASH DES V2

STUDENTS

SARNO IVAN

LEO CARLO

MAZZARELLA ALESSANDRO

PROFESSOR

Prof. SAPONARA SERGIO

REFERENT

Ing. CROCETTI LUCA

AY 2020/2021

Summary

1. Specification analysis

In this chapter, we will discuss the overall system specification.

The *fullHashDES* module is a hardware implementation of a hash function, based on DES S-box, which returns a 32-bit digest. The digest is divided into 8 chunks of 4-bit and is initialized with default values before the process starts. The function performs two types of rounds:

– ***main round***: It reduces the byte to process to 6-bit and uses it as S-box entry, then the S-box output is combined through bitwise XOR operations with each chunk of the digest and those are rotated through circular shift operations. This combination-rotation process is repeated four times. The *main round* is formally described as follows:

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

where M_6 is the reduction of the byte on 6-bit, *S* is the S-box and *H* is the temporary digest. The $\oplus$ operator is the bitwise XOR and << is the circular shift operator.

– ***final round:*** It is executed after all bytes of the message have been processed; the total length of the message (64-bit) is divided into 8 chunks, then each one is reduced to 6-bit, used as an S-box entry and the outputs are combined with the chunks of the digest in the same position, in a single combination-rotation iteration. The *final round* is formally described as follows:

```
for(i = 0; i < 8; i ++)
    $H[i] = ( H[(i + 1) \bmod 8] \oplus S(C_6[i])) << \lfloor i/2 \rfloor$
```

where $C_6[i]$ is the i-th byte of the message length reduction on 6-bit, *S* is the S-box and *H* is the temporary digest.
The $\oplus$ operator is the bitwise XOR and << is the circular shift operator.

At the end of the process, the chunks are concatenated in a single 32-bit digest.

The interface of the module presents, as required by the specification, 3 input ports, and 2 output ports:

- *M*: current byte of the message to process
- *M_valid*: a 1-bit signal that expresses the presence of a new byte to process
- *C_in*: the length of the message represented on 64-bit
- *Hash_ready*: a 1-bit signal that expresses the completion of the process
- *Digest_out*: 32-bit digest

2. Block diagram and design choices

In this chapter, we will show as the code has been turned into hardware by Quartus, briefly showing the match between some snippet of code and RTL (Register Transfer Level) diagram. Furthermore, we are going to discuss the design choices applied.

2.1 Design choices

The M input is stored in the register *M_r* to avoid long combinatory paths that can increase the delay. Consequently, even *M_vaild* is stored to check if in the previous clock cycle a valid byte to process has been stored.

The input *C_in* is stored in 2 registers: the first, *C*, is used during the final round as the input of the S-box, the second, *counter*, is used to count the remaining bytes to process.

The output is not directly sent to the pins but is stored in the *digest_out* register along with a *hash_ready* register to avoid long combinatory output paths, then the critical path is between the input and the output ports.

Each Byte of the message is processed in a single clock cycle by some combinatory sub-modules:

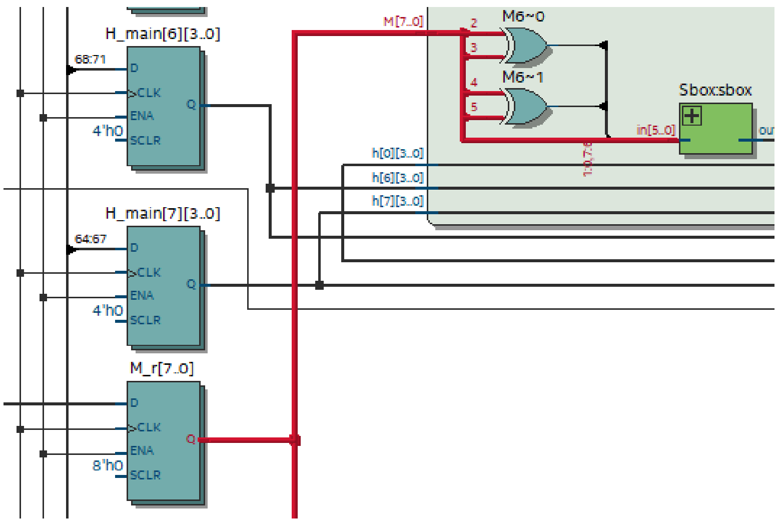
- The ***S-box*** is implemented as a combinatory network with 6-bit input and 4-bit output.
- ***hashRound***: is a combinatory network that computes a single iteration of the combination-rotation phase of the main round. Its interface includes
  - *s_value*: The S-box output as input
  - *h*: an array of 8 signals, 4-bit each, represents the temporary digest in input
  - *h_out*: an array of 8 signals, 4-bit each, represents the temporary digest in output
- ***mainHashIteration***: is a combinatory submodule that computes the reduction of the input byte, the S-box lookup, and the four combination-rotation phases, integrating four concatenated hashRound sub-modules. Its interface includes：
  - *M*: an 8-bit signal representing the byte of the message to process.
  - *h*: an array of 8 signals, 4-bit each, represents the temporary digest in input
  - *h_out*: an array of 8 signals, 4-bit each, represents the temporary digest in output

  This structure allows to process a byte in a single clock cycle, at the cost of a longer combinatory path.
- ***hashRound_final***: is a combinatory submodule that computes the final round, it is composed of 8 reduction networks, 8 S-box and 8 combination-rotation networks, one for each 8-bit chunk; this allows to process all chunks parallelly. Its interface includes:
  - *C*: the total length of the message represented using 64-bit
  - *h*: an array of 8 signals, 4-bit each, represents the temporary digest in input
  - *digest*: the computed digest.

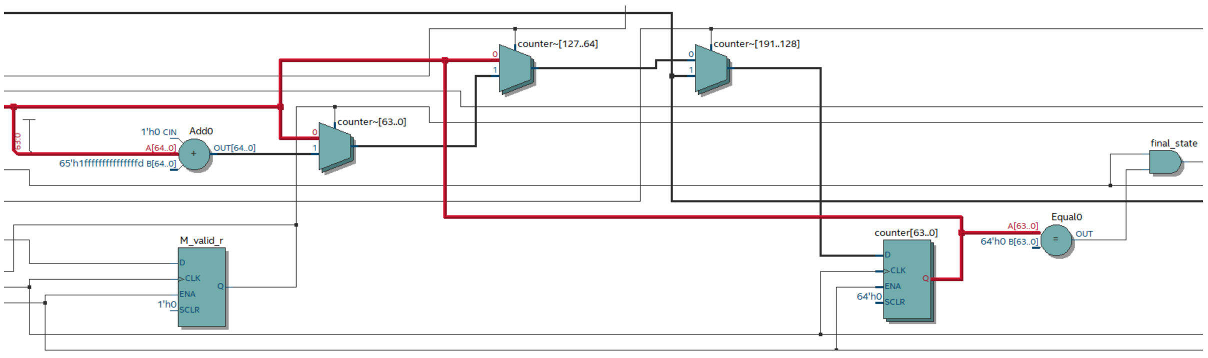
Those sub-modules are integrated into the *fullHashDES* modules that includes the control network and the state elements of the modules.

2.2 Block diagram

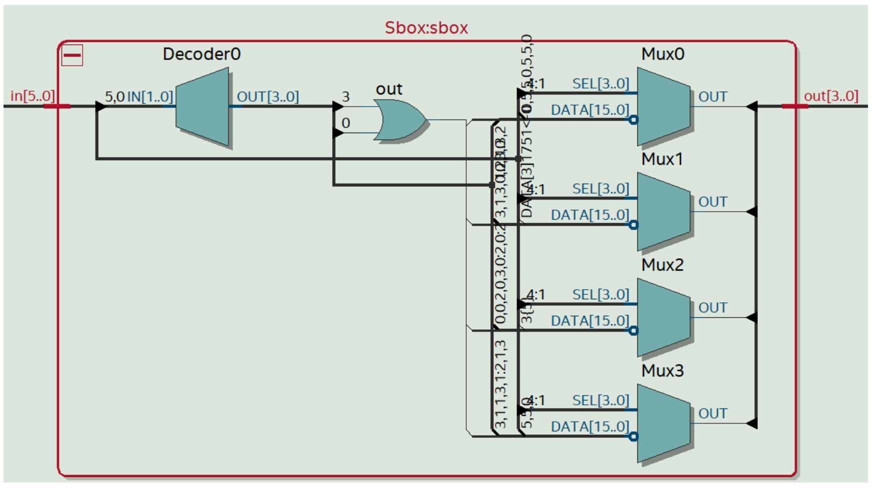
This section reports the most relevant portions of the netlist, represented as RTL scheme using Quartus.



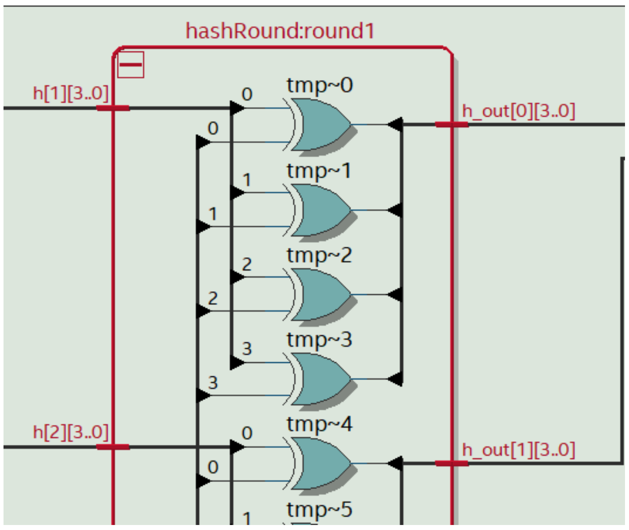
This portion shows how the byte of the message stored in the *M_r* register is used as input of the *mainHashIteration* network, which executes the reduction to 6-bit and accesses the S-box.



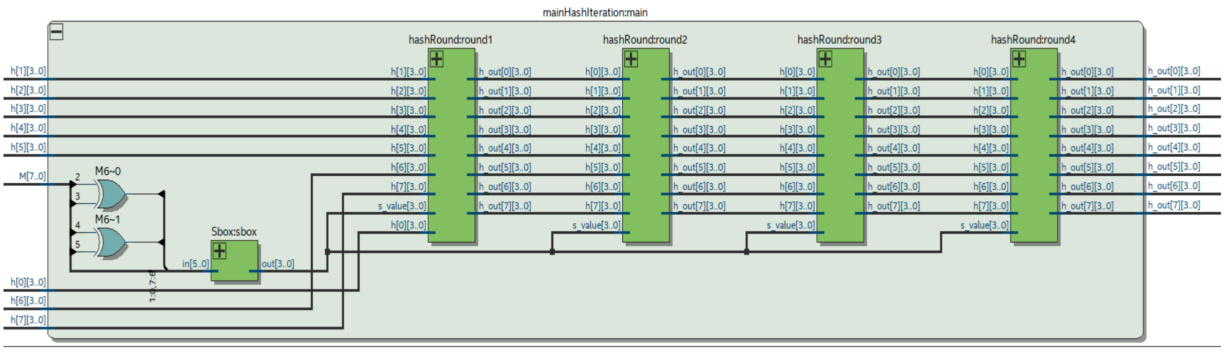
This portion shows how the *C_in* input is stored in *C* and *counter* registers, and *counter* is decremented for each byte processed and it is used to compute control signals.



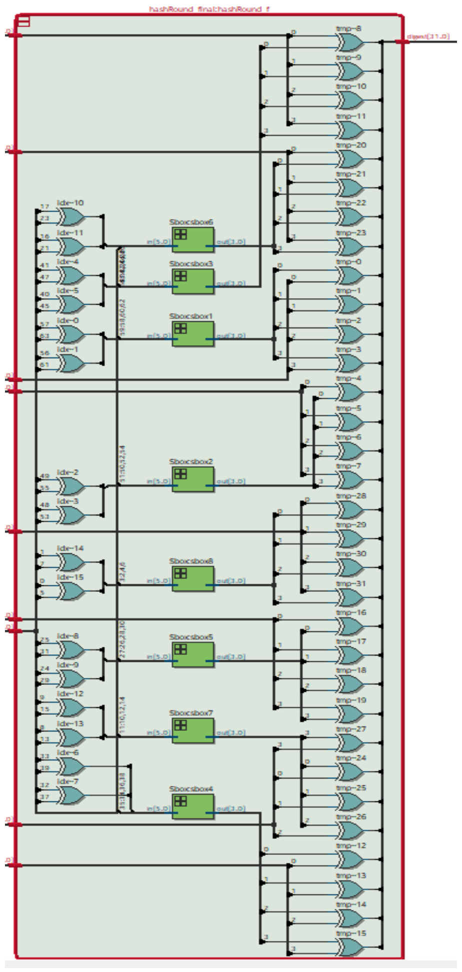
This image shows the implementation of the S-box (described in section 2.1).



This image shows part of the implementation of *hashRound*, that executes the XOR and the rotation operations.



This image shows the implementation of the *mainHashRound* sub-module, which is composed of the gates for the reduction, one S-box, and the concatenation of four *hashRound* sub-modules. As we can see the S-box output is sent to each *hashRound* sub-module.



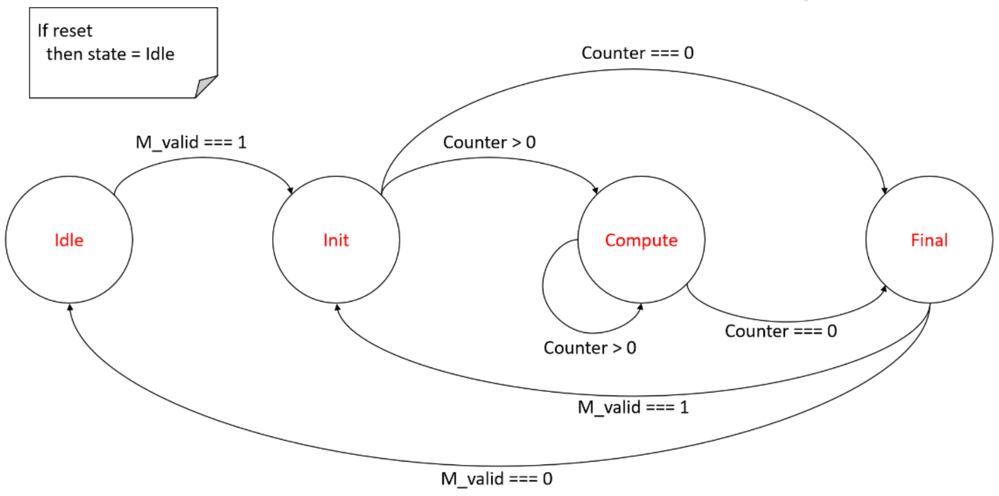
This image shows the structure of the *hashRound_final* module (described in section 2,.1), which includes 8 S-box, one for each chunk of the digest, that are combined with the output of the S-boxes, rotated, concatenated, and returned as output to the *digest_out* register.

2.3 States & transitions

The module has four states:

I. **Idle** → The module waits for a new message to process. The *M_valid* signal triggers the transition to the *init_state*.
II. **init_state** → The module initializes its working registers, stores the length of the message and, if present, the first byte to process.
III. **compute_state** → The module processes, if present, the byte stored in the previous cycle, updates the temporary digest, and decrements the counter. If *M_valid* is set, the module stores the next byte to process in the *M_r* register.
IV. **final_state** → If the *counter* is 0, there is no other byte to process and the module executes the final round, processing the length of the message, store the result in the *digest_out* output register and set the *hash_ready* register to signal to the outside that the process has been completed.

The *rst_n* signal puts the module in *Idle* state and unset *hash_ready*.



This image shows the state diagram of the module as FSM.

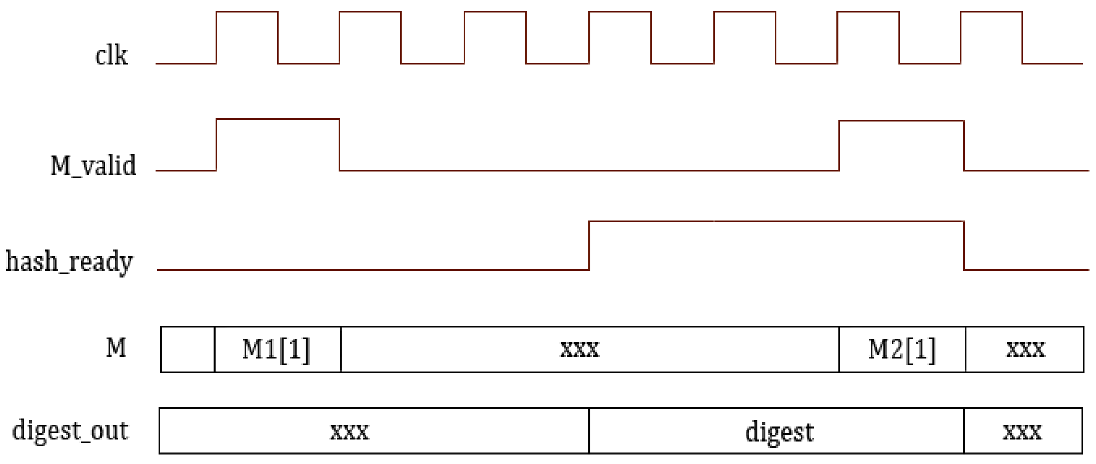
3. Usage

In this chapter, we will explain how the circuit must be integrated inside a hardware secure module.

The interface of the module has been described in chapter 1. To start using the module, the user must set *M_valid*, provide to the module the *length* of the message through the *C_in* port, and the first byte to process through the *M* port. The first byte of the message must be provided along with the message length.

To process the remaining bytes, the user must set *M_valid* and send each byte to the *M* port, one per cycle.

The module signals when the process is completed by setting the signal *hash_ready*, then the user can read the digest on the *digest_out port*.

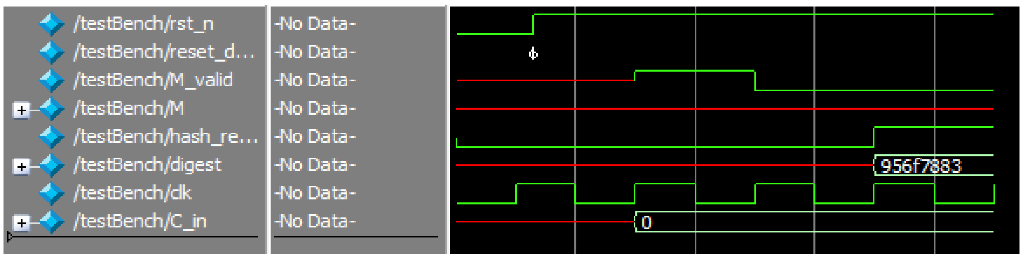


In this figure we can observe the expected behavior of the circuit with a 1 char message input and the computation of its digest, which remains valid until a new message (byte) is provided.

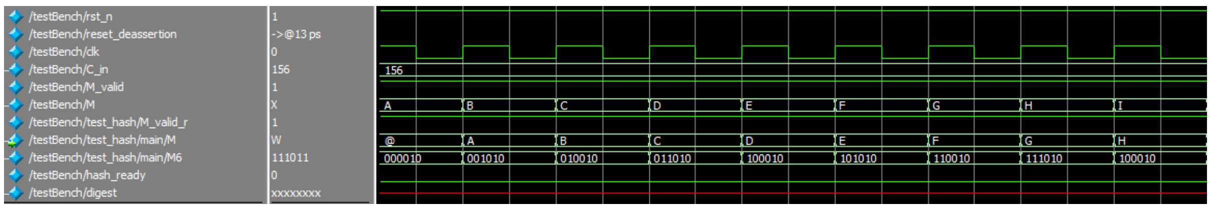
4. Waveform discussion

In this chapter we will briefly discuss the most relevant pieces of wave form:

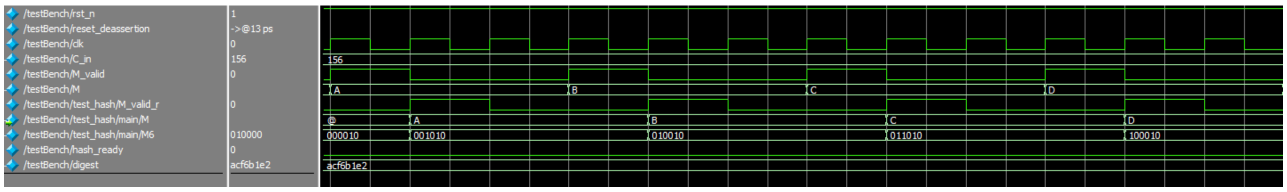
- **Empty message processing →** it shows the correct evolution of the output signals during the process of an empty message, this process lasts 2 clock cycles, because only the *init* phase and the final round are executed. We can see that *M_valid* starts the computation and at the end, *hash_ready* and *digest_out* are set at the same time.



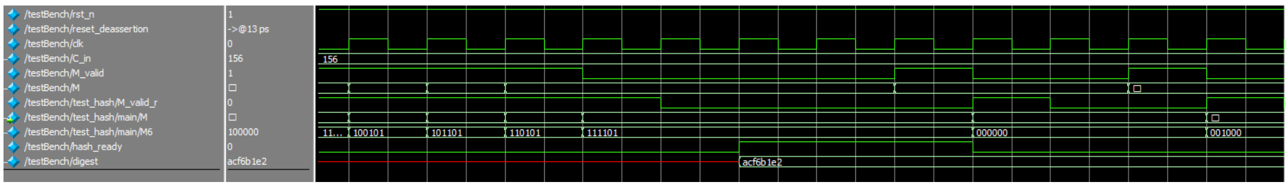
- **Continuous message computation →** the message is sent in a continuous manner, one byte per cycle. Each byte needs 2 cycles to be processed, one for sampling it from the input port and one to execute the *mainHashRound*. We can see the correct timing of the store-compute process. Also, we can see that the *M_valid* is never unset during all the computation.

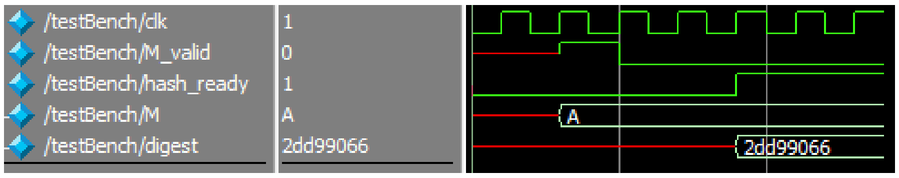


- **Not continuous message computation →** in this case the bytes are provided separated by a time interval of 2 cycles. We can see as the module handles this case correctly.



- **Output signals →** it shows the behavior of the output signals. We can see that *hash_ready* is set when the digest is available and is kept to one until a new message is sent to the module.





By observing the waveform obtained from the *TEST_ONE_CHAR,* we can conclude that if the bytes of the message are provided one per cycle, the whole process takes *length + 3 clock cycles*, except in the case of an empty message.

5. Testbenches

In this chapter, we will explain how we have designed the test benches and what they do.

- **TEST_EMPTY**: it aims to test the corner case of an empty message, which has length 0 and requires only the final round. The expected output has been computed by hand.

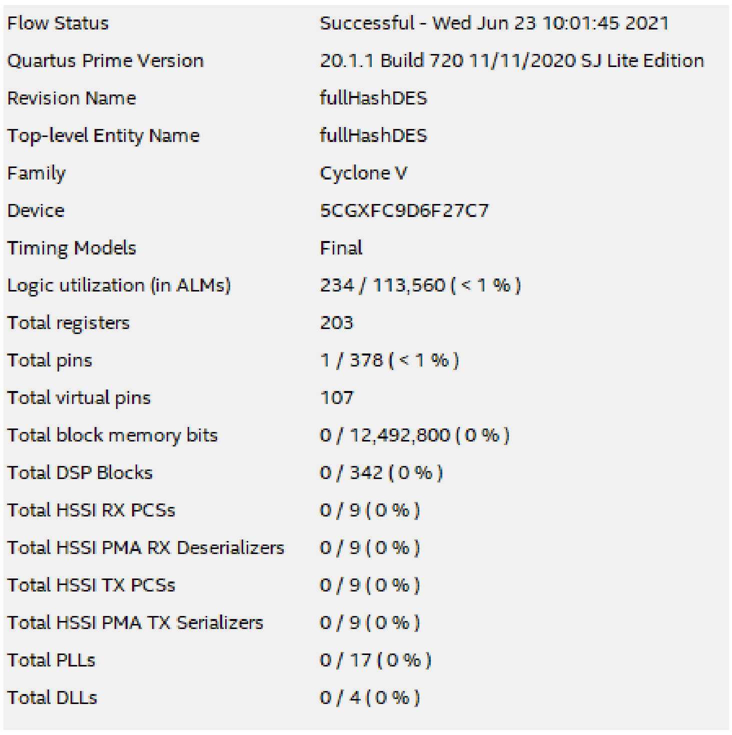
- **TEST_ONE_CHAR**: it aims to test the corner case of a single character message, it requires message reading and storing, main round, and final round execution. The expected output has been computed by hand.

- **TEST_HASH**: this test computes 3 digest for the same message, one sending the bytes in a continuous manner, one per cycle, the second sending the bytes separated by 2 clock cycles, and the third removing the last char of the message. Then the digests are compared to ensure that the first two are identical and the third is different. This test aims to verify that identical messages produce identical digests, that different messages produce different digest, even for a small difference between the messages, and that the module correctly handles the input acquisition and the state transition.

- **TEST_LONG_MESSAGE**: this test aims to stress the module with a long message. Due to non-standard algorithm, for non-trivial messages it is not possible to easily obtain an expected output (unlike TEST_EMPTY and TEST_ONE_CHAR). Hence, this test verifies the correct behavior of the circuit in terms of the correct amount of clock cycles to produce the digest.

6. Synthesis result and Timing analysis

We discuss the synthesis result in terms of resources employed over the target device and the summary of the timing analysis.



Flow Status	Successful - Wed Jun 23 10:01:45 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	fullHashDES
Top-level Entity Name	fullHashDES
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	234 / 113,560 (< 1 %)
Total registers	203
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 %)

This image shows the summary of the resources used by the module after the fitting.

We can see the low usage of resources which is less than 1%, in relation to those available in the target.

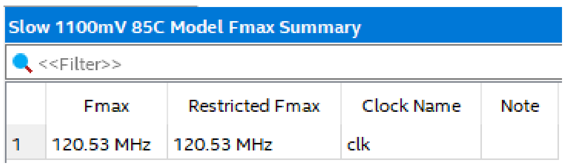
The number of used pins is coherent with our expectations:

- *M*: 8-bit
- *C_in*: 64-bit
- *M_valid*: 1-bit
- *clk*: 1-bit
- *rst_n*: 1-bit
- *digest*: 32-bit
- *hash_ready*: 1-bit

  108 in total, 107 virtual pins and 1 real pin (clk).

The number of used registers as we estimated from the code is 203 registers, as well as we observe in the netlist.

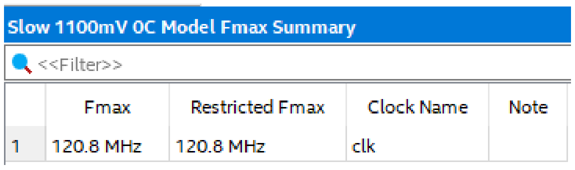
For the timing analysis we have defined a timing constraints file (*timing_constraints.sdc*) specifying a target frequency of 100MHz, defining a 10 nanoseconds period clock. Furthermore, for each unconstrained path (input/output signals), we have defined maximum and minimum delay as a percentage of the system clock, 20% and 10% respectively.
The frequencies obtained during the slow tests (85C and 0C) are reported below:



Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	120.53 MHz	120.53 MHz	clk	



Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	120.8 MHz	120.8 MHz	clk	

As we can see, the obtained frequencies meet the requirements even in the worst case (on the left), with a frequency equal to 120.53 MHz (which is the guaranteed one) and up to 120.8 MHz with a much lower temperature (on the right).