Università di Pisa

Master of Science in Computer Engineering

Project Report

Full Hash Algorithm (AES S-Box)

Hardware and Embedded Security

Tommaso Billi

Academic Year 2020/2021

Contents

Chapter 1

Specification analysis

The aim of the project was to develop a hardware implementation of a hash module based on the Rijndael S-box. In figure 1.1 are reported the full specifications of the module to implement.

Design the following hash module based on the S-box of AES algorithm. The hash algorithm generates a 64-bit digest formed by the concatenation of 8 bytes $H[i]$, from $H[0]$ up to $H[7]$, being $H[0]$ the MSB. For each message the $H[i]$ variables are initialized with the following values:

	$H[0]$	$H[1]$	$H[2]$	$H[3]$	$H[4]$	$H[5]$	$H[6]$	$H[7]$
Init. value	8'h20	8'hA3	8'h9F	8'h3E	8'hCC	8'h74	8'hCB	8'hF0

The, for each byte M of the input message (i.e. the 8-bit ASCII code of a message character) the hash module performs the following operation:

$$for\ (r = 0;\ r < 64;\ r++)$$
$$for(i = 0; i < 8; i++)$$
$$H[i] = S\big((H[(i+1)\ mod\ 8] \oplus M) \ll i\big)$$

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.

Once the last message byte has been processed, the hash module performs the following operation:

$$for(i = 0; i < 8; i++)$$
$$H[i] = S\big((H[(i+1)\ mod\ 8] \oplus C[i]) \ll i\big)$$

where $C[i]$ is the ith byte, for $i = 0,1,2,\ldots,7$, of the counter C which reports the byte length of message. Note that C reports the real number of byte length, i.e. if the message length is 1 byte, then $C = 1$ (not 0). The value of counter C has to be provided as input.

Figure 1.1: Project specifications.

Along with the specifications, a reference design that implements a similar hashing function based on the DES S-box was also provided.

The main differences of the AES S-box hashing module and the reference design are the following:

- The number of iterations of the outer-most round within the main round of the algorithm is 64, compared to 4 in the reference design;

- The main computation performed at each iteration, both in the main round and in the final round is substantially different;
- The produced digest is on 64-bits (formed by the concatenation of 8 bytes), instead of 32-bits (concatenation of 8 4-bit nibbles);
- The hash module obviously uses the Rijndael (AES) S-box, instead of the DES S-box.

The reference design is made out of 5 different submodules (5 individual SystemVerilog files). More specifically, the submodules are the following:

- `des_sbox.sv`: as the name states, it implements DES's S-box as a 4×16 LUT.
- `hash_main_round.sv`: this module computes the inner-most operation within the main iteration of the algorithm. All 8 4-bit vectors are computed concurrently within this module.
- `hash_main_iteration.sv`: it's a wrapper for all the operations that are performed in the first step of the algorithm (i.e. it encapsulates the iterations of the main steps of the computation of the digest). It does so by cascading 4 main round modules. More specifically, the input of the first round is used as an output for the second, and so on.
- `hash_final_round.sv`: this submodule implements the last step of the hashing operation.
- `full_hash_des_top.sv`: this module is a wrapper for all other submodules that make up the final circuit. It takes as input the clock signal, an asynchronous reset, the current byte to process, the *M_valid* flag, and the count of the total bytes of the message (on a 64-bit register). It outputs the digest, and a *digest_ready* flag, to indicate that the computation has ended for the current message.

The reference design is implemented as a finite state machine, to model the switching between the the different phases of the hashing algorithm. Three states have been defined: *init_state* (the first state that is entered as soon as a new message is provided; the circuit is initialized with the provided initialization vector), *compute_state* (the main state of the computation; after the initialization, when in this state, the module performs all the main round iterations), *final_state* (once the last byte of the message has been processed, the circuit enters in this final state, in which the output of the main round goes through the final round submodule; at the end of this step, the final digest is presented at the output of the circuit).

Additionally, there exists also a fourth "hidden" state: when the *M_valid* bit is set to 0, the module enters an idle state in which no computations are performed, and the circuit pauses every operation, waiting for a new valid input to be provided.

To make sure I correctly understood the requirements, the first thing I did was write a Python scripts that emulates the design, and provides the hash computed according to the specifications for any given input. I tried to maintain the same modular design in this script as well. The results of such script will be useful in the debugging and testing phases of the development.

Chapter 2

Design choices and block diagrams

2.1 Design choices

The first step of the design phase was the analysis of the reference design, that is briefly explained in the previous chapter.

The idea was to understand the implementation of the reference design in order to reuse as many components from it as possible, to develop a variation of the design that implemented this project's requirements.

Apart from the S-box as a LUT implementation, that had to be re-written from scratch, many elements of the original design could be reused. The partitioning in submodules was also coherent with the requirements of this project, so the overall structure remained substantially unchanged.

As a design choice, I decided to maintain the original idea of the reference design, where each input byte is sampled at every clock cycle. As opposed to the reference design, where there are only 4 iterations of the main round, according to the specifications of this project, the main round performs 64 iterations. In order to perform all these operations within a single clock cycle, the final circuit will have a long critical path formed by a cascade of *hash_main_round* modules. The practical implications of such design choice will be discussed more in detail in chapter 6.

2.2 Block diagrams

In this section are reported the block diagrams of some key components of the circuit. The diagrams were exported from the RTL viewer tool of Quartus, after having successfully synthesized the design. As the final RTL design is extremely complex and hard to read, only some of the most significant sections of the circuit are shown.

In figure 2.1 we can see the *H_main* registers (8 in total, one for each byte of the digest) and the *M_r* register. They hold, respectively, the value of the digest computed at the previous iteration and the byte (ASCII character representation) that is currently being processed. Such registers form the inputs to the cascade of *hash_main_round* modules (in the picture, the first two are shown).

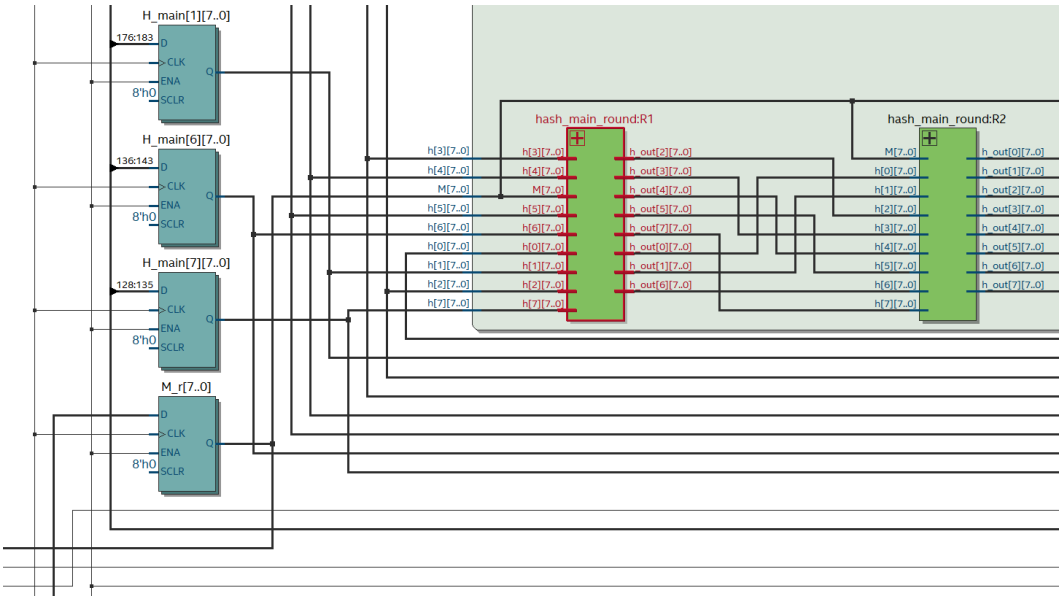

Figure 2.1: Circuit main iteration.

In figure 2.2 we can see the logic behind the counter of the circuit. The counter holds the value of the number of bytes that still need to be processed in the current message. If the *M_valid* bit is set, the counter is decremented by 1 at the next clock cycle. Once the counter reaches a value of 0, the *final_state* is triggered.

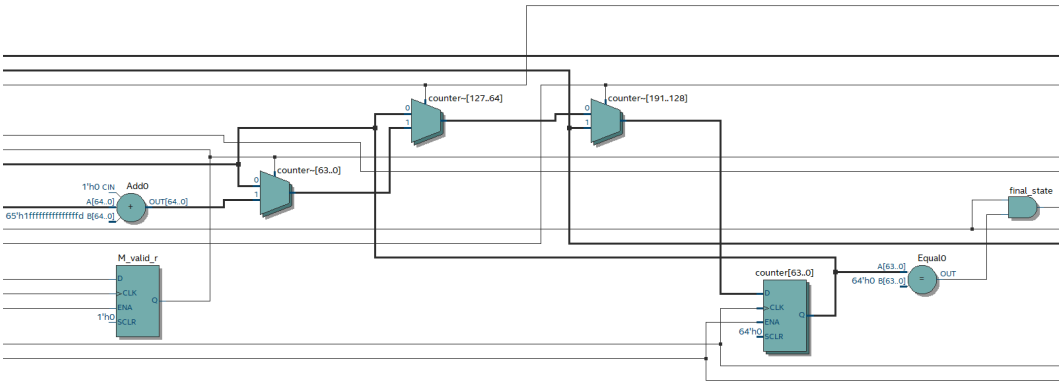

Figure 2.2: Counter logic.

Finally, in figure 2.3, we can observe that when the *final_state* is reached, the output from the final round module (whose inputs are the outputs of the main iteration, and a register which stores the message length) is propagated to the `digest_out` register, while at the same time the `hash_ready` register is set to 1.

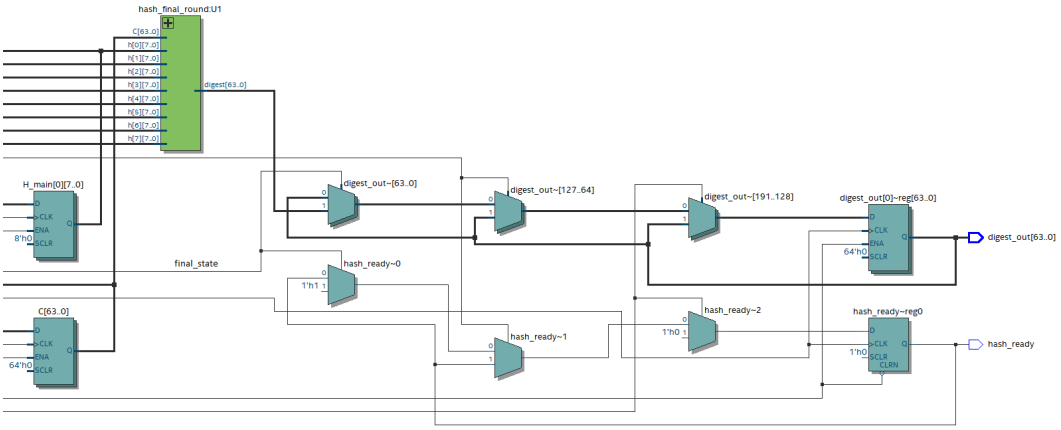

Figure 2.3: Circuit's final round.

Chapter 3

Expected waveforms

In this chapter the different expected waveforms associated with different scenarios are going to be addressed and discussed.

For the first example, shown in figure 3.1, one of the two testbenches (described in chapter 4) will be taken into account.

In this case, we provide as input to the module the string "Hello", with each byte separated from one another by a number of clock cycles between 1 and 5. As soon as the first valid input is provided, the circuit performs first the initialization round, performs the main iteration for the first character, decrements the counter, and then waits for the next valid input. Once even the last character has been processed, the output of the `final_round` submodule is presented at the output of the circuit at the next clock cycle, and the `hash_ready` flag is enabled.

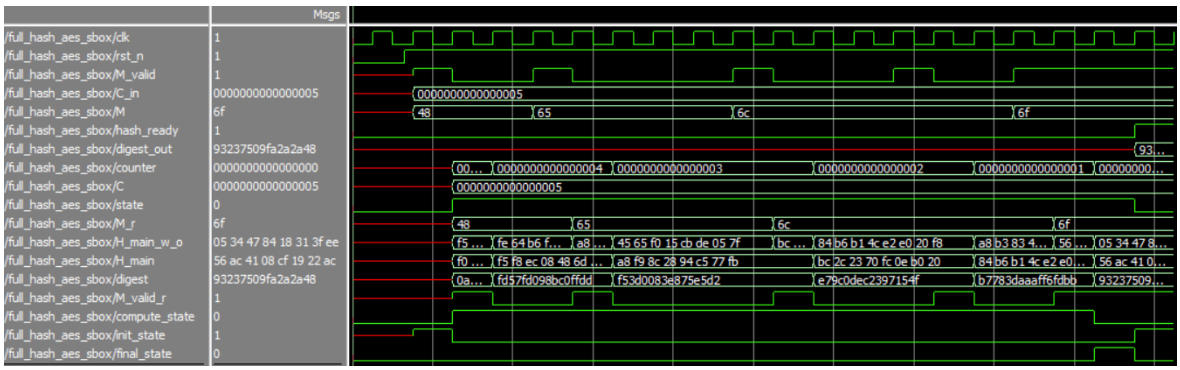

Figure 3.1: Waveform with non-sequential input characters.

In figure 3.2 we can observe what happens when we provide a continuous stream (one character at every clock cycle). The situation is basically the same as before: at every round the same partial results are obtained, and in the end, since we provided the same string as before, we obtain the same digest. The only difference is that once asserted, the *M_valid* bit always remains set to 1, and thus the circuit never enters into the idle state.

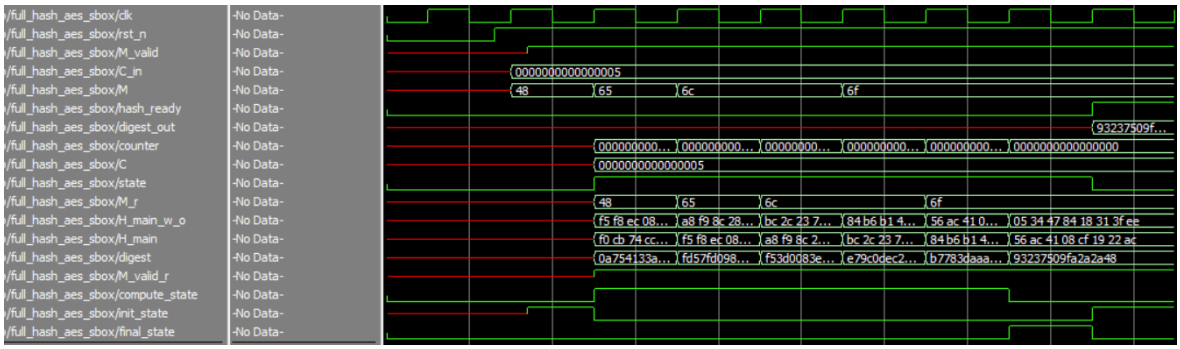

Figure 3.2: Waveform with a continuous stream of characters.

Another interesting case to analyze is the one in which we provide an empty string as input. In this case, the counter is already set to zero, so the circuit never actually enters into the *compute_state* phase. The only operations that it needs to perform are the initialization and the final round; the output is ready after two clock cycles (fig. 3.3).

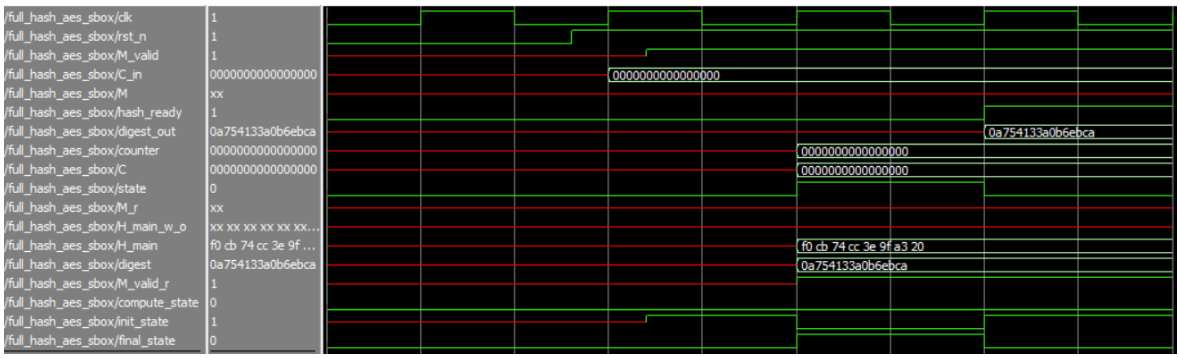

Figure 3.3: Waveform with an empty string as input.

With this in mind, we can say that the overall latency L of the module (i.e. the delay between an input is provided to the circuit, and the corresponding output is computed), when providing a continuous stream of data as input, is

$$L = T_{clk} \times (N_b + 2),$$

where N_b is the number of bytes in the input message. The final addend of two is accounting for the two additional clock cycles needed for the first input character to be available on the M_r register, and for the final digest to be ready on the *digest_out* register, respectively.

Chapter 4

Testbenches

In order to test the RTL design, a SystemVerilog testbench and a modified version of the Python script that was initially used for prototyping were developed.

The Python script, which can be found inside the `scripts/` directory, asks the user the input message, computes the hash, and outputs the digest on screen. By providing the `"-w"` option when launching the script, the script also produces two files, that by default will be created in the `tb/test_files/` directory.

The first file (`message.txt`) will contain the entire input for which the user wishes to compute the hash, as well as the length of the message itself, that is a mandatory input parameter to the circuit.

The second file (`target.txt`) will contain instead the hexadecimal (lowercase) string of the computed hash.

These files are then used for the SystemVerilog testbench itself. Please note that both these files are required for the testbench to run properly; you should run at least once the Python script with the `"-w"` option specified.

From the first file, the length of the message is initially read, and C_in (the 64-bit register on which the message length is stored) is set accordingly. Then, at each clock cycle, a single character of the message is read, until the end of the file is reached. After an additional clock cycle after the last character is read, the final digest is computed.

At this point, the second file is opened, and the hash computed by the circuit is compared to the one that is found in the `target.txt` file. If the two digests match, the test passes, and the console will output "Success!"; otherwise, an error message is printed.

Many different inputs were tried during the test phase, to ensure the correct functionality even in some edge cases. In particular, it was verified:

- that an identical input message would always produce the same hash;
- that the circuit correctly computed digests for long messages (500+ characters);
- that the digest was computed even for an empty message (0 characters long input).

In a second testbench, the correct behavior of the circuit when non-contiguous inputs were provided was tested (i.e. not a character to process per clock cycle). The input string "Hello" was tested, providing one byte at a time, and then waiting for a random number of clock cycles (between 1 and 5) before providing the next character. In order

to generate a different random sequence at each simulation, one must change the seed value (line 45) before launching the simulation.

In this case, it's important to set the `M_valid` bit to 0 after providing each input byte, to let the module know that subsequent inputs have to be ignored, and not considered as repeated characters. If everything is done correctly, the circuit enters into an idle state, where neither the counter and the output are updated, and the waveform is the one seen in figure 3.1.

The test is considered passed if the final hash is always the same, even in different clock wait cycles.

Chapter 5

Implementation of RTL design on FPGA and results

Once the implementation was completed and adequately tested, the last step of the project was to use the Quartus tool by Intel to try to synthesize the RTL design on an actual FPGA. In this case, the 5CGXFC9D6F27C7 device of the Cyclone V family was chosen.

At the end of the processing, the final synthesis report that was produced is the one shown in figure 5.1.

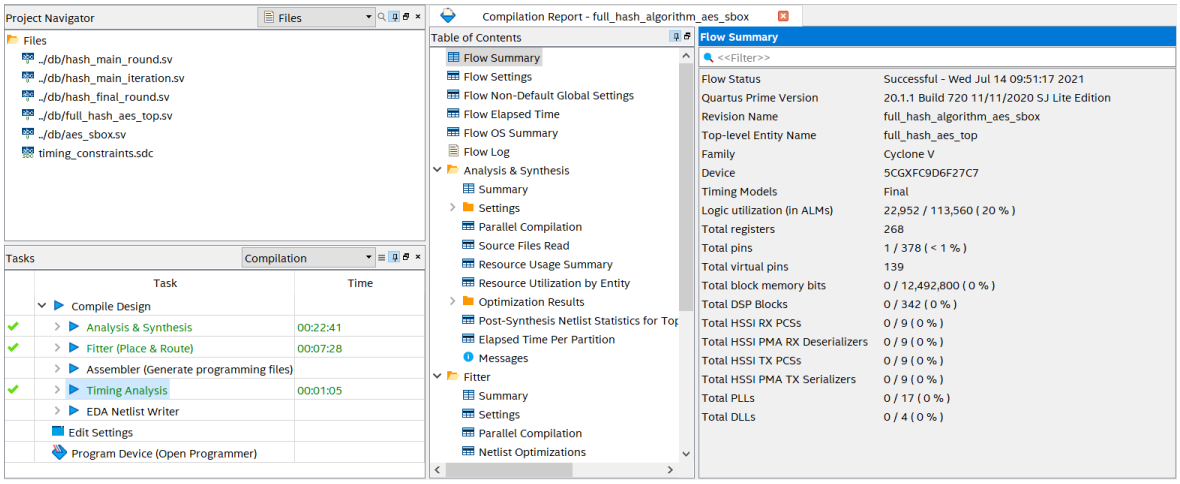

Figure 5.1: Synthesis report of the design.

From the flow summary, we can observe that there is a total utilization of 140 pins (1 real pin, and 139 virtual pins). This is coherent with the design; as each pin (excluding the clock) corresponds to a single bit of the inputs and outputs of the design. As the input and output ports of the final top-level design is the following:

```
module full_hash_aes_top (
    input              clk
   ,input              rst_n
   ,input              M_valid
   ,input       [63:0] C_in
   ,input        [7:0] M
   ,output reg         hash_ready
   ,output reg [63:0] digest_out
);
```

we can say that this result reflects the initial expectations.

Chapter 6

Static Timing Analysis

After making sure that the synthesis process could be performed without any errors, the final step was the timing analysis of the synthesized design.

As a first attempt, I created a Synopsys Design Constraints (.sdc) file with a clock period constraint of 10 ns (100 MHz frequency), and constrained all inputs and outputs with a minimum and maximum delay. Such values were chosen using the empirical rule of picking the 10% of the clock period as the minimum delay, and the 20% of it as the maximum.

I did not expect these constraints to be respected by the design, however; as all the computations, including the 64 iterations of the first round, are computed in a single clock cycle, the critical path (i.e. the longest possible register-to-register path) is very long. This implies a considerably high propagation delay, that severely limits the maximum clock frequency of the design.

In the following figure are reported the results obtained in the worst case (Slow 1100mV 85C model).

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	5.7 MHz	5.7 MHz	clk

Slow 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	-165.541	-10498.332

Slow 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	-0.265	-0.959

Figure 6.1: Timing report summary (F_{clk} = 100 MHz).

The negative slack (in red) represents the margin by which the timing requirements are not met. Furthermore, in the first tab we can observe that the maximum frequency

for the clock is 5.7 MHz, which is far from the 100 MHz constraint. Additionally, it's worth mentioning that this analysis was performed prior to assigning the virtual pins of the FPGA device.

After these considerations, the .sdc file was modified, specifying a clock period of 200 ns (so a frequency of 5 MHz). All the other constraints were also modified accordingly. Before re-launching the analysis, the virtual pins were assigned as follows:

	Status	From	To	Assignment Name	Value	Enabled	Entity
1	Ok		hash_ready	Virtual Pin	On	Yes	full_hash_aes_top
2	Ok		rst_n	Virtual Pin	On	Yes	full_hash_aes_top
3	Ok		C_in	Virtual Pin	On	Yes	full_hash_aes_top
4	Ok		M	Virtual Pin	On	Yes	full_hash_aes_top
5	Ok		digest_out	Virtual Pin	On	Yes	full_hash_aes_top
6	Ok		M_valid	Virtual Pin	On	Yes	full_hash_aes_top

Figure 6.2: Virtual pins assignments.

At this point, I launched the Timing Analysis once again, and this time all the requirements were met. No parts of the Timing Analyzer summary were highlighted in red, and success is confirmed by the Timing Closure Recommendations tab, which reports: "This design does not contain any failing setup paths. The worst-case slack is 13.343 ns.", and "No paths fail setup timing.".

The final results are reported in the following figure (6.3).

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	5.36 MHz	5.36 MHz	clk

Slow 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	13.343	0.000

Slow 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.426	0.000

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	5.38 MHz	5.38 MHz	clk

Slow 1100mV 0C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	14.009	0.000

Slow 1100mV 0C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.437	0.000

Fast 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	102.299	0.000

Fast 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.179	0.000

Fast 1100mV 0C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	112.032	0.000

Fast 1100mV 0C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.171	0.000

Figure 6.3: Timing analysis summary for Slow and Fast models (0 and 85C).

As a final consideration, it's worth mentioning that the low maximum clock frequency is due to the design choice of letting the user provide a character at every clock cycle. As

said before, this choice creates a long critical path, that in turn implies a considerable propagation delay for the circuit.

An alternative design to obtain a higher maximum clock frequency would have been to implement the 64 iterations of the main round in more stages (i.e. implement a pipeline by placing registers between one or more rounds of the main iteration). While increasing the maximum clock frequency, this would also increase the latency of the circuit. For example, by placing a register every 2 or 4 rounds would mean that every input byte would need 32 or 16 clock cycles, respectively, to be processed. In this case, the latency formula would become the following:

$$L = T_{clk} \times ((N_b \times (64 \div N_s)) + 2),$$

where N_s is the number of stages.