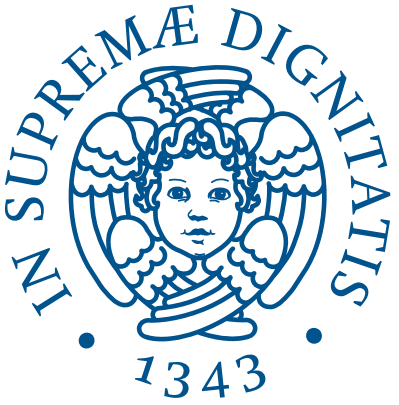

UNIVERSITÀ DI PISA

Master of Science in Cybersecurity

Project Report

Vigenère Cipher Decryption

Hardware and embedded security

E. Casapieri, A. Di Donato

Academic Year 2020/2021

Indice

Capitolo 1

Specification analysis

The Vigenère cipher is a polyalphabetic substitution cipher based on the use of a table $\boldsymbol{T}$(Figure 1.1). In $\boldsymbol{T}$, of size 26x26, each row(and column) i, with $1 \leq i \leq 26$, contains the alphabet of twenty-six letters rotated to the left by i-1 positions. The encoding(decoding) is done by adopting, along with the table $\boldsymbol{T}$, a key k which is generated by starting with a keyword and repeating it in a circular fashion until the length of k coincides with the length of the text to be encoded(decoded).

The purpose of this project was to implement decoding of the newly introduced cipher.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figura 1.1: Vigenère Table

1.1 Encryption

The first letter of the plaintext is matched with the first letter of the key. It then uses the first one to identify the line of $\boldsymbol{T}$, and the second one for the column; in this way the first letter of the ciphertext can be located. Similarly, for the second letter of the plaintext, the second letter of the key is used. The rest of the plaintext is encrypted in a similar way.

1.2 Decryption

Decoding is performed as follows:

1. You locate the first letter of the key in the left column and on the line the first letter of the message to be decoded.
2. Go up the column to read the first letter of the decrypted message.
3. Continue with the next letters of the message and the next letters of the key; when you have reached the end of the key, go back to the first letter of the key.

1.3 Algebraic implementation

It is possible to identify an algebraic implementation of the cipher in which the letters [A-Z] are converted into numbers [0-25]. In this case given E the ciphertext, P the plaintext and K the key, the encoding of the i-th letter is done as follows:

$$E_i = (P_i + K_i) \bmod 26$$

Decoding, on the other hand, is performed as follows:

$$D_i = (E_i - K_i + 26) \bmod 26$$

Below you can see the python implementation of the functions used in the testbench phase to generate the test files; they refer to the algebraic implementation of the cipher.

```
def generateKey(string, key):
        """ Generate the key cyclically until its length is equal to the
        ↪  length of the original text """
        key = list(key)
        if len(string) == len(key):
                return key
        else:
                for i in range(len(string) - len(key)):
                        key.append(key[i % len(key)])
        return "" . join(key)


```

```
13 def cipherText(string, key):
14         """ Returns the ciphertext using the key """
15         cipher_text = []
16         for i in range(len(string)):
17                 # Converte nell'intervallo 0-25
18                 x = (ord(string[i]) + ord(key[i])) % 26
19                 # Converte in ASCII
20                 x += ord('A')
21                 cipher_text.append(chr(x))
22         return "" . join(cipher_text)
23
24
25 def originalText(cipher_text, key):
26         """ Decrypts the ciphertext and returns the original text """
27         orig_text = []
28         for i in range(len(cipher_text)):
29                 # Converte nell'intervallo 0-25
30                 x = (ord(cipher_text[i]) - ord(key[i]) + 26) % 26
31                 # Converte in ASCII
32                 x += ord('A')
33                 orig_text.append(chr(x))
34         return("" . join(orig_text))
```

Capitolo 2

Block diagram and design choices

At the highest level, the structure to be used for decoding a character using the Vigenére cipher is as follows:

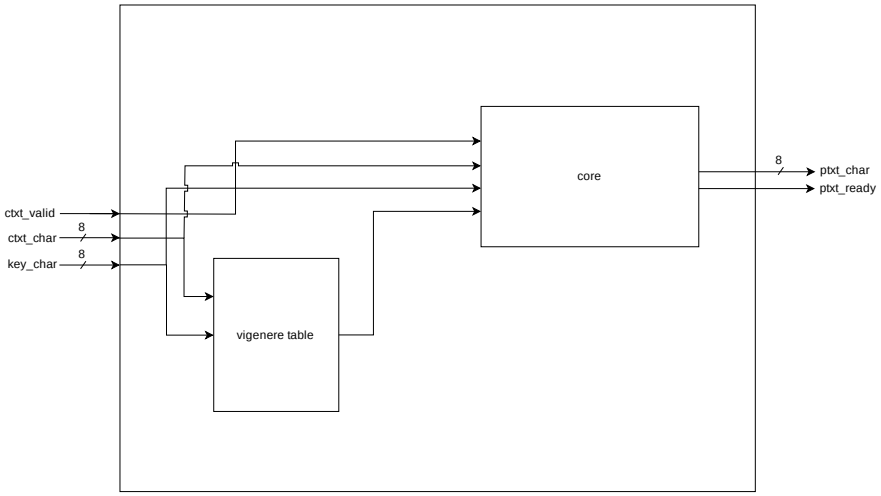

Figura 2.1: Block diagram vigenère decipherer

dove:

- **Input:**
  - *ctx_char*: is the 8-bit ASCII code of the character to be deciphered.
  - *ctx_valid*: is an active high line, set to 1 when the supplied input is valid.
  - *key_char*: is the 8-bit ASCII code of the character to be used as key.
- **Output:**.
  - *ptxt_char*: is the 8-bit ASCII code of the decrypted character.
  - *ptxt_ready*: a high active line, raised to 1 when the output *ptxt_char* provided is valid and can be consumed by the logical resources connected to the vigenère decipherer.

2.1 Vigenere table

The following cipher is based on the use of the $\boldsymbol{T}$ table shown in Figure 1.1 and accessed by using the character to be decrypted and the key to be used. In this case the table is implemented inside a ROM, which has two inputs of 8 bits each, which are *ctx_ char* and *key_ char* respectively, and which are used to compose the address used to take the value of the decrypted character; these values are already saved in the ROM and are represented by 8 bits with hexadecimal encoding.

2.2 Core

This component is of fundamental importance since it contains all the control logic necessary for the correct implementation of the decipherer. Its main functions are:

- Management of the initial state of the circuit: in particular a value of invalid is imposed on the output *ptxt_ ready*.
- Control of the 8 bit ASCII character upper case.
- Control of the condition of the 8-bit uppercase ASCII character.
- Management of the decrypted character. It is the Core that sends out *ptxtxt_ char*.
- Management of the signal *ptxt_ ready* to indicate the validity (or invalidity) of the decrypted character.

Capitolo 3

Expected waveforms

From the specifications it is expected that:

- The ciphertext character and the key character can be only 8-bit ASCII character representing upper-case letters.
- The ctxt_valid input signal is asserted upstream when the input ciphertext character (ctx_char signal) is valid and has to be consumed by the Vigenère Cipher. It must be equal to zero otherwise.
- The ptxt_ready output signal is asserted when the plaintext character is available at the corresponding output port (ptxt_ready port) and has to be consumed by the logic resources linked to Vigenère Cipher. It must be equal to zero otherwise.
- The Vigenère Cipher shall have an asynchronous active-low reset port.
- After the reset is deasserted the ctx_char and the key_char signals assume new ASCII characters at each clock cycle.
- The Vigenère Cipher shall decrypts (only decryption function is required) one cyphertext character per clock cycle.

More specifically, proposing some examples of possible inputs it is required to obtain the following waveform:

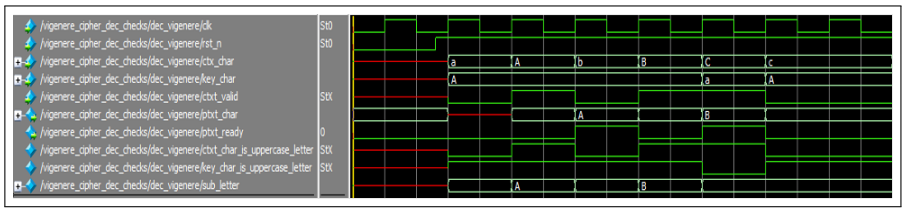

Figura 3.1: Vigenère decipherer: internal structure

- At the first interesting clock cycle the input ciphertext character is a lowercase "a" that is not valid. The key character instead is a valid character. It is expected that the signal ctxt_char_is_uppercase_letter, which checks the corresponding condition, is set to 0 while the signal key_char_is_uppercase_letter is set to 1. As a result at the next rising edge of the clock the output signal ptxt_ready will be 0 since the output plaintext character is not valid (NULL char).

- At the next clock cycle, both inputs are valid and the ctxt_valid signal is 1. As a result at the next rising edge of the clock the output signal ptxt_ready will be 1 and the output plaintext character is read from the Vigenère table.
- After some clock cycles for example the inputs are the uppercase ciphertext character “C”, the ctxt_valid signal equal to 1 and the lowercase key character “a”. In this case it is expected that the signal ctxt_char_is_uppercase_letter is set to 1 while the signal key_char_is_uppercase_letter is set to 0. As a result at the next rising edge of the clock the output signal ptxt_ready will be 0 since the output plaintext character is not valid (NULL char).

Capitolo 4

Testbench

In order to verify the functional requirements, i.e. the correctness of the results provided by the circuit according to the specifications, it is developed some testbench code to simulate the Vigenère Cipher drive by using the ModelSim environment. In this environment, two 8-bit ASCII characters are supplied as input, corresponding to the ciphertext character and the key character, for each clock cycle along with the additional 1-bit value for the ctxt_valid signal. The other inputs are appropriately driven by giving the clock a period of 10 ns and making the reset go to '1' after 12.8 ns.
The following tests has been developed:

1. The objective of the first test is to obtain the expected waveform described at the point 3 giving to the module the same inputs. After the simulation, it has been verified that all the signals behave according to the specifications.

2. The objective of the second test is to verify the proper behavior of the ptxt_ready signal in presence of incorrect ctxt_valid signal drive. In particular it is verified the robustness of the module in presence of some errors in the upstream logic. For example giving as input a lowercase ciphertext character "a" along with ctxt_valid equal to 1, the output will be not valid and the ptxt_ready signal is set to zero.

3. Other simulations have been made in order to check the correct decoding of every uppercase character of the alphabet with all possible key. In this way it has been verified that the approach using the value in the stored table is equivalent to the approach using the analytical formula of the Vigenère cipher. For this purpose a testbench that automates this verification have been created: by using the Fork – Join construct provided by SystemVerilog, multiple procedural blocks have been spawned off at the same time in order to perform 26 different tests. Each one takes as key always a specific character (corresponding to a row in the Vigenère table) and as ciphertext all the uppercase characters of the alphabet (corresponding to all the columns of the selected row) presented at each clock. The output plaintext is then compared to the plaintext coming from the Vigenère analytical formula. Eventually all the tests have been successful.

4. A python model of the Vigenère cipher decryption has been developed in order to generate test vectors. Given a randomly generated list of 260-character plaintexts, for each one the model is able to perform the encryption and to create a text file where the first row contains the original plaintext, the second row contains the key that is generated in a cyclic manner starting from an initial keyword of

10 characters until its length is equal to the length of original plaintext, and the third row contains the ciphertext produced by the model. The following snippet shows the developed python code.

```python
import random
import string
from vigenere import generateKey
from vigenere import cipherText
from vigenere import originalText


for j in range(100):
        plaintext = ""
        input_ptx = []
        for i in range(260):
            input_ptx.append(string.ascii_uppercase
            [random.randint(0,25)])
        plaintext = "".join(input_ptx)
        keyword = "AYUSHAIOPS"
        key = generateKey(plaintext, keyword)
        cipher_text = cipherText(plaintext,key)
        str_ptx = originalText(cipher_text, key) + "\n"
        str_key = key + "\n"
        str_ctx = cipher_text + "\n"
        f= open("./Tests_Files/testTextTarget_"+ str(j)
        ↪  +".txt","w+")
        f.write(str_ptx)
        f.write(str_key)
        f.write(str_ctx)
```

Even in this case the testbench has been automated by using the Fork – Join construct. Each multiple procedural block performs a test that uses one of those different text files as reference. The testbench SystemVerilog code is created by a python script whose code is the following:

```python
import string

for number in range(100):
    code = """

            @(posedge clk);

            fork

            begin: textCompleteCTX_%s

                FP_PTXT =
↪  $fopen("../scripts/Tests_Files/testTextTarget_%s.txt", "r");
                if (FP_PTXT == 0) begin
                    $display("data_file handle was NULL");
                    $finish;
```

```
                end
                $display("Processing testTextTarget_%s.txt");

                while($fscanf(FP_PTXT, "%%c", char) == 1) begin
                    if(char !== "\\n\") begin
                        value_read = int'(char);
                        if (counter < 261)
                            PTXT.push_back(value_read);
                        else if (counter >= 261 && counter < 521)
                            KEY.push_back(value_read);
                        else
                            CTXT.push_back(value_read);
                    end
                    counter = counter + 1;
                end

                counter = 0;
                for(int i = 0; i < 260; i+=1) begin
                    key_char = KEY.pop_front();
                    ctx_char = CTXT.pop_front();
                    EXPECTED_CHECK = PTXT.pop_front();
                    @(posedge clk);
                    @(posedge clk);
                    $display("%%c %%c %%-5s", ptxt_char,
↪   EXPECTED_CHECK, EXPECTED_CHECK === ptxt_char ? "OK" : "ERROR");
                    if(EXPECTED_CHECK !== ptxt_char) $stop;
                end
            end: textCompleteCTX_%s

            join

    """

    print(code % (number, number, number, number))
```

Each block opens the specific text file, parse the rows and save the contained strings in 3 appropriate registers. Subsequently at each clock cycle each character of the ciphertext and of the key is passed to the Vigenère cipher decryption module. At the next positive edge of the clock the output of the module is taken and compared to the original plaintext read from the text file. If every characters of the 2 plaintexts are equal the test is passed otherwise the simulation is stopped. It is verified that all the tests have been successful on all the 100 considered text files.

Capitolo 5

Implementation of RTL design on FPGA and results

Since satisfactory results were obtained during the testing phase in the ModelSim environment, we moved on to the synthesis and implementation phase. This was carried out using Quartus Prime 20.1.1 Build 720 11/11/2020 SJ Lite Edition.
The steps that have been performed here are:

- **Analysis & synthesis:** logic synthesis to minimize the logic usage of the design, and performs technology mapping to implement the design logic using device resources such as logic elements. It generates a single project database integrating all the design files in a design. The following is the internal structure of the Vigenère decipherer resulting from the RTL analysis:

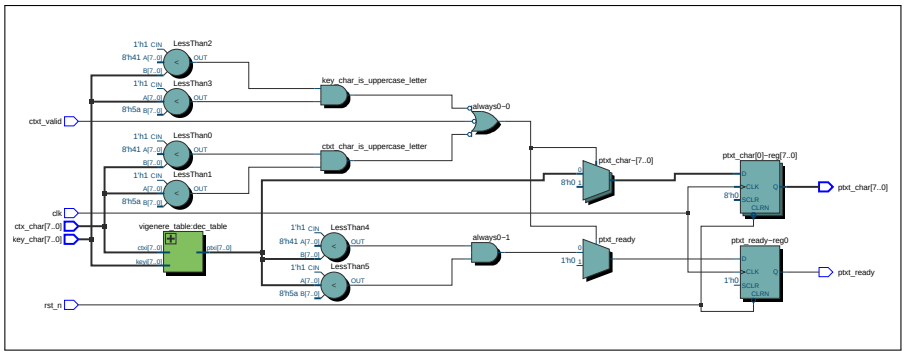

Figura 5.1: Vigenère decipherer: internal structure

- **Fitter (Place & Route):** In this phase, following the successful completion of the analysis and synthesis step, we fit the logic of the design into the design's target device: 5CGXFC9D6F27C7 (Cyclone V Family).

Below is the final report of the two phases carried out:

Flow Status	Successful - Sat Nov 6 12:28:24 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	vigenere_cipher_dec
Top-level Entity Name	vigenere_cipher_dec
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	403 / 113,560 (< 1 %)
Total registers	7
Total pins	28 / 378 (7 %)
Total virtual pins	0
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 %)

Figura 5.2: Quartus report

Some comments:

- Logic usage is quite low, below 1%.
- The number of pins used is 28, this corresponds to the total number of inputs and outputs.
- The number of registers is 7, and not 9 (8 plaintext characters + the plaintext ready signal). This is because the representation of all uppercase ascii characters that can be output share 3 bits at the top, but it is also possible to have a sequence of 8 bits of all 0, so the bits actually shared are 2 and not 3.

Capitolo 6

Static Timing Analysis(STA)

In order to perform the STA, a timing constraints file is produced where a clock period of 8 ns is specified. Since the clock is asynchronous, a directive for the STA engine of Quartus is also added so that it does not check that the reset signal paths respect the clock constraint. Moreover, to solve the warning of Quartus about the unconstrained paths, input and output delays are added, specifying the minimum and the maximum delays following a rule of thumb, i.e 10% of the clock period is used for the minimum and the 20% for the maximum.

```
1 create_clock -name clk -period 8 [get_ports clk]
2 set_false_path -from [get_ports rst_n] -to [get_clocks clk]
3 set_input_delay -min 0.8 -clock [get_clocks clk] [get_ports {rst_n
  ↪  ctx_char[*] key_char[*] ctxt_valid}]
4 set_input_delay -max 1.6 -clock [get_clocks clk] [get_ports {rst_n
  ↪  ctx_char[*] key_char[*] ctxt_valid}]
5 set_output_delay -min 0.8 -clock [get_clocks clk] [get_ports
  ↪  {ptxt_char[*] ptxt_ready}]
6 set_output_delay -max 1.6 -clock [get_clocks clk] [get_ports
  ↪  {ptxt_char[*] ptxt_ready}]
```

Since the implemented design can be seen as a subsystem of a more complete and general system, this information has to be given to the synthesis engine of Quartus specifying the input/output ports as virtual pins, except for the clock. This allows to neglect additional combinatory delays due to the fact that the output ports are mapped onto the output pins of the FPGA and not on internal logic resources.

Taking into account just the Slow 1100 mV 85C Model, within Quartus time requirements has been successfully respected with the following results:

- Setup summary report shows that the worst slack is 0.524 ns.
- The critical path is the path that goes from the inputs key_char to the outputs ptxt_char.
- Hold summary report shows that the worst slack is 0.579 ns.
- The maximum clock frequency is 133.66 MHz.