Immagine che contiene Carattere, testo, Elementi grafici, logo

Descrizione generata automaticamente

MSc in Computer Engineering

Electronics and Communications Systems

**Project Report: The Caesar Cypher**

Francesco Bruno

**Academic Year 2023-2024**

Content

[1 Introduction 3](#_Toc164969203)

[2 Architecture Design Flow 4](#_Toc164969204)

[2.1 Input-Output Port Description 4](#_Toc164969205)

[2.2 Analyze common bits in ASCII-Code 5](#_Toc164969206)

[2.3 The choice of LUT content 6](#_Toc164969207)

[3 A view of the Logic Circuit 7](#_Toc164969208)

[3.1 Analysis of Encryption / Decryption functions 7](#_Toc164969209)

[3.2 The Input ASCII Error Detector Function 8](#_Toc164969210)

[4 Correct Behavior Circuit Check 9](#_Toc164969211)

[5 Synthesis/Implementation on Vivado 11](#_Toc164969212)

[5.1 Resource utilization 12](#_Toc164969213)

[5.2 Timing and evaluation 12](#_Toc164969214)

[5.3 Critical Path 13](#_Toc164969215)

[5.4 Power Consumption 14](#_Toc164969216)

[6 Conclusion 14](#_Toc164969217)

# Introduction

The goal of this project was to design a circuit that implements The Caesar's Cypher. It is one of the oldest known cryptographic algorithms.   
This cypher is based on a very simple principle, called monoalphabetic substitution:  
each letter of the text to be ciphered is replaced with another letter of the alphabet.   
The characteristic feature of the algorithm is that the offset factor between the input (plaintext) and output (ciphertext) letters is constant throughout the message.

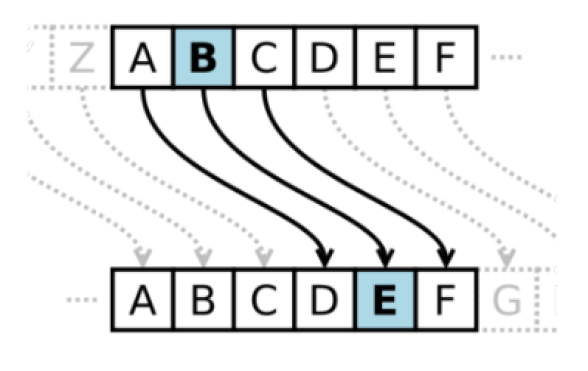


Figure 1 - Caesar Cypher Concept

By keeping in mind the above picture, every cyphered character is the result of the “sum” between its integer representation and the key. A simple formula to compute the cyphered character is shown in the next chapter.

So, by thinking the alphabet as a linear array, with that formula, the alphabet is managed as a circular array.

# Architecture Design Flow

Here follows the description of the project architecture and the choices that I made for the circuit architecture.

## Input-Output Port Description

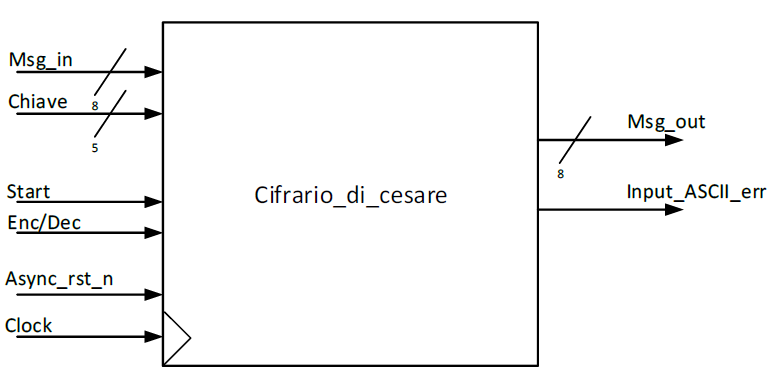


Figure 2 - Input and Output Ports

Input ports:

* Msg\_in : plaintext letter that I want to cypher, it’s on 8 bit
* Chiave : it’s the cypher-key. It represents the number of shifts of the output cyphered-letter. It’s on 5 bits because the maximum shift is 26 letters.
* Start : it’s only an “enable” signal to the cypher process.
* Enc/Dec : it’s one signal to choice the circuit functionality (Encryption or Decryption)
* Async\_rst\_n : it’s the reset signal
* Clock : circuit clock

Output ports:

* Msg\_out : encrypted letter. It’s the result of input letter and the key
* Input\_ASCII\_err : this is an Error signal to inform the downstream circuit that there is an error on the input character, i.e. Out-Of-Range Input Character.

## Analyze common bits in ASCII-Code

As a project constraint, the cypher circuit can work only with lower case letter, so my first step was to analyze the features of the characters ascii code of that range.   
The *Table1* is the ASCII table in which I have also added the relative binary of the ***ASCII-Code***. The feature extracted are the common bits between all the letters, as I’ve highlighted in red.

|  |  |  |
| --- | --- | --- |
| **Letter** | **ASCII\_Code** | **Binary** |
| a | 97 | 01100001 |
| b | 98 | 01100010 |
| c | 99 | 01100011 |
| d | 100 | 01100100 |
| e | 101 | 01100101 |
| f | 102 | 01100110 |
| g | 103 | 01100111 |
| h | 104 | 01101000 |
| i | 105 | 01101001 |
| j | 106 | 01101010 |
| k | 107 | 01101011 |
| l | 108 | 01101100 |
| m | 109 | 01101101 |
| n | 110 | 01101110 |
| o | 111 | 01101111 |
| p | 112 | 01110000 |
| q | 113 | 01110001 |
| r | 114 | 01110010 |
| s | 115 | 01110011 |
| t | 116 | 01110100 |
| u | 117 | 01110101 |
| v | 118 | 01110110 |
| w | 119 | 01110111 |
| x | 120 | 01111000 |
| y | 121 | 01111001 |
| z | 122 | 01111010 |

Table 1 – Lower Case Alphabet ASCII Code

Since all lower-case letters have the three Most Significant Bits equal to 011, I’ve chosen to ignore them in the cypher process, and to use them only in *Error detection module.*  
So, when a character enters in the circuit, will be considered only its 5 Least Significant Bits, for the encryption/decryption functionalities.

## The choice of LUT content

As mentioned in the previous paragraph, I chose to add or subtract (depending on the working mode of the circuit) to the key, the 5LSB of the input character, and in the end calculate the remainder or the quotient, again depending on the circuit working mode.  
By doing this I am sure that the result is always representable on 5 bits, and then I use that result as address to the LUT, since the result and address are aligned.

By choosing the LUT content as shown below, I have created a circular array.

|  |  |  |
| --- | --- | --- |
| **Lookup Table** | | |
| Address | Stored Value | ASCII-Char |
| 00000 | 01111010 | z |
| 00001 | 01100001 | a |
| 00010 | 01100010 | b |
| 00011 | 01100011 | c |
| 00100 | 01100100 | d |
| 00101 | 01100101 | e |
| 00110 | 01100110 | f |
| 00111 | 01100111 | g |
| 01000 | 01101000 | h |
| 01001 | 01101001 | i |
| 01010 | 01101010 | j |
| 01011 | 01101011 | k |
| 01100 | 01101100 | l |
| 01101 | 01101101 | m |
| 01110 | 01101110 | n |
| 01111 | 01101111 | o |
| 10000 | 01110000 | p |
| 10001 | 01110001 | q |
| 10010 | 01110010 | r |
| 10011 | 01110011 | s |
| 10100 | 01110100 | t |
| 10101 | 01110101 | u |
| 10110 | 01110110 | v |
| 10111 | 01110111 | w |
| 11000 | 01111000 | x |
| 11001 | 01111001 | y |
| 11010 | 01111010 | z |

As you can notice, to close the circle, first and last LUT-address, have the same value that is the ASCII-Code of the ‘z’.

# A view of the Logic Circuit

Here, there is my idea of the electronic circuit to perform the Caesar Cypher algorithm, and this represent the hardware I’ve described with the VHDL language.



Figure 3 - Logic Circuit

## Analysis of Encryption / Decryption functions

The function of circuit is driven by the value of the Enc/Dec input signal. In details:

* If ***Enc\_Dec = ‘0’***, the circuit works in ***CYPHERING-MODE***:  
  The 5LSB are first extended on 6 bits, and then they are summed with the key (also extended). Below is shown a summarized formula, only to understand the computation of the cyphered char out, so without the LUT-address part.

) )

The complete operation is a bit different, because with the sum result,   
i’ve computed the REM-26 operator, that is the computation of the division quotient with the constant *26*. This operation is used to compute the correct LUT-address (my “circular-array”) to select the correct cyphered character related to plain-text character (because the circuit is cyphering).

* If ***Enc\_Dec = ‘1’***, the circuit works in ***DECYPHERING-MODE***:  
  In a dual way, the decryption functionality is realized by subtracting at the 5LSB,   
  the key, as you can see in the above formula (without the LUT-address part).

) )

Also, in this case the complete operation involves, after the mathematic operation, the Module operator (MOD-26) with the ‘*26*’ constant. In this way, I can compute the Rest of the division, also for the negative number (since the operation is a subtraction). So, at the end, is computed the LUT-address to select the correct plain-text character related to the input cyphered character (because the circuit is decrypting).

## The Input ASCII Error Detector Function

To verify that the input character is a Lower-Case letter, the circuit has the Error Detector Functionality. So, technically speaking, I have to check if ASCII code of the input character is in the range of [97 – 122], that represents the ASCII CODE range of the lower case alphabet.

By exploiting the *msg\_in* separation in 5LSB and 3MSB, I choose to divide the error detection functionality in two sub-functionalities:

* The *Error on 3MSB functionality* is in charge of check if there was an error on 3MSB of the input character. If you look the *Table1*, you can notice that the lower-case letters characters have always the 3MSB equal to “011”.   
  With a simple use of the *bool algebra*, it’s trivial manipulate the equation and obtain a simple error detector logic function:
* *The Error on 5LSB functionality* is responsible for check if there was an error on the other part of the input character, that is its 5LSB.  
  The correct character input range by considering only its 5LSB is [00001 – 11010], that corresponds to [1, 26] decimal range. So, it’s sufficient to check if the 5LSB of the input letter are out of that range.   
  This is the equivalent to verify these two situations:
  + If the 5LSB are all equal to 0 🡪 it’s trivial to do in hardware.
  + If the 5LSB are greater or equal than 27 🡪 I have chosen to subtract the constant 27 to the 5LSB, and then if the result sign bit is not 1, then, this means that the 5LSB are greater or equal to 27, and then are out of range.

So, if at least one of these two situation is happened, that there was an error on the input letter.

By merging the result of the two previously situation, to obtain the *Error Detection Circuit* is sufficient to put in OR the two outputs of the two sub-functionalities. In this way if at least one of the two signal is ‘1’, then the *Input ASCII Error* signal is ‘1’ as well.See the VHDL code to better understand.

# Correct Behavior Circuit Check

In this paragraph, I show how I verified the correct circuit behavior in different working mode (encryption – decryption – input character out of range).

* First test: the circuit works in **Encryption Mode** (*Enc\_Dec = ‘0’)*, with the key equal to 13, and the input letters are (in the correct range). So, what I expect is that, with the character ‘*a’* as input char, it wil be cyphered in the thirteenth character after *‘a’* in the alphabet, that is the *‘n’*.   
  If the input char will be the *‘b,* it will be cyphered in the *‘o’* and so on… as you can see in the below Modelsim screenshot.

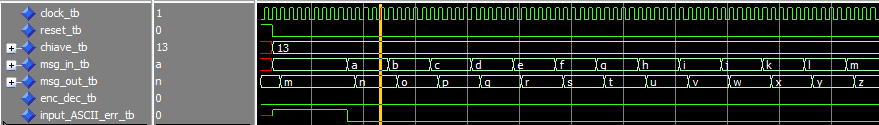


Figure 4: Encryption Mode Test

* Second Test: the circuit works in **Decryption Mode** (*Enc\_Dec = ‘1’),* with the same encryption key, and the input letters are the output of the previous encryption.   
  So, I expect that, if I give in input the previous cyphered characters, the output is the previous plaint-text input characters.  
  In other words, I expect that, with letter *‘n’* as input, with the key 13, it will be decrypted with the letter *‘a’*. With the letter *'o’* as input char, it will be decrypted in the *'b’*, and so on… as you can notice in the *Figure 5*.

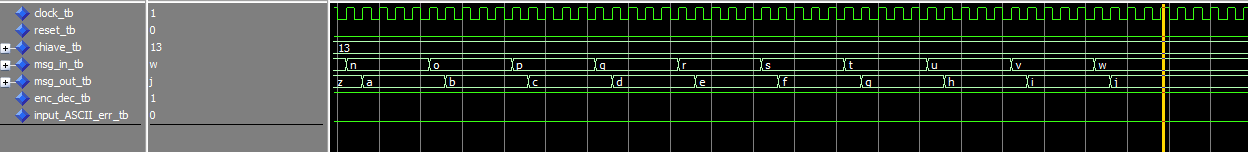


Figure 5: Decryption Mode Test

* Third Test: Error Detection Circuit Test.  
  This test is composed by three sub-tests, just to separately check of the error detection functionality.  
    
  In order to check the *5LSB part* of the error detector, I have given as input an out-of-range character. In detail, the input was ‘*01100000’*, so its 5LSB are all equal to ‘*0’*, and this is an error situation because the character is out of correct range.   
  So, what I expect is that when that input will be presented to the circuit, it will rise the *input\_ASCII\_err* signal, and this is exactly the obtained behavior, as demonstrated in *Figure* *6*.

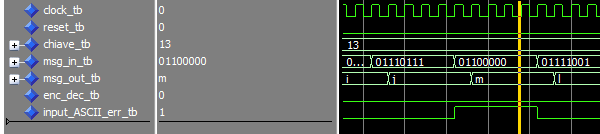


Figure 6: Third test – All 5LSB equal to 0

As a second sub-test of the Error Detector Circuit, I set the input to ‘*01111011’*, that is the *ASCII CODE* relative to *‘{‘ that is the* open bracketcharacter.  
Since the input character is not a lower-case-letter, I expect that the signal *input\_ASCII\_err* is raised up, exactly as show in the *Figure 7 .*

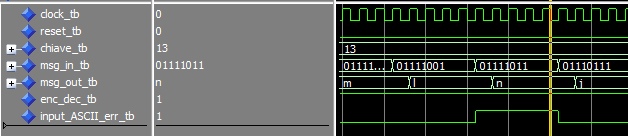


Figure 7: Third Test - 5LSB are greater than 26

As final sub-test of the Error Detector Circuit, I set the input to ‘*11110000’*, to check the error detection on the 3MSB of the input character. So, what I expect was that when the circuit “see” that the 3MSB are different from ‘*011’*, it raises up the *input\_ASCII\_err signal,* and this is what I obtained in the Figure 8.

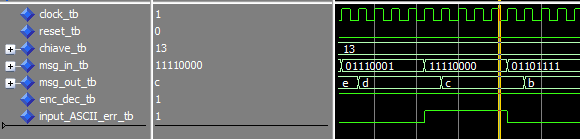


Figure 8: Test Third - 3MSB are equal to ‘111’

In conclusion to the test phase, I can say that the circuit has the expect behavior, in both the encryption and decryption modes. Thanks to this correct behavior, i can work with Vivado tool.

# Synthesis/Implementation on Vivado

The system schematic after the design phase elaboration is shown in *Figure 9*.



Figure 9 - Caesar Cypher Wrapper Schematic



Figure 10 - Caesar Cyphrer Schematic

After elaborating the design, the synthesis is performed with no constraints first to verify that everything is correctly working and that the design is bug-free.   
Then I defined the constraints by declaring the clock period to be 8 nano sec, to achieve a maximum clock frequency of 125MHz.

## Resource utilization

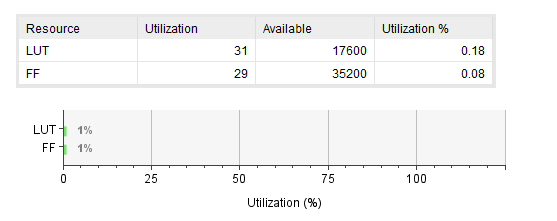


Figure 11 - Resource Utilization Summary

## Timing and evaluation

The timing summary with a clock period of 8ns gave me a WNS of 2.3nS.  
Thus, the optimal clock period is 8 – 2.3 = 5.7ns, then the maximum clock frequency is around 175MHz.

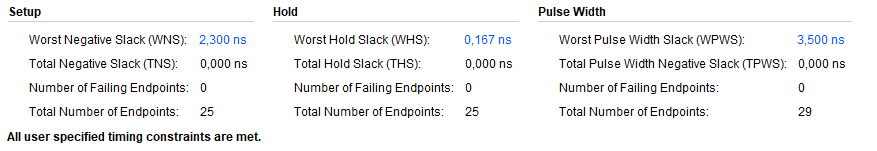


Figure 12 – Report Timing Summary

## Critical Path

By selecting the Worst Negative Slack item, the Vivado tool automatically shows the relative critical path of the circuit. In my case this highlighted path, in the *Figure 11,* is related to the *Chiave* port. This is because after the synthesis phase, the signal coming from *Chiave* must across 5 LUTs.

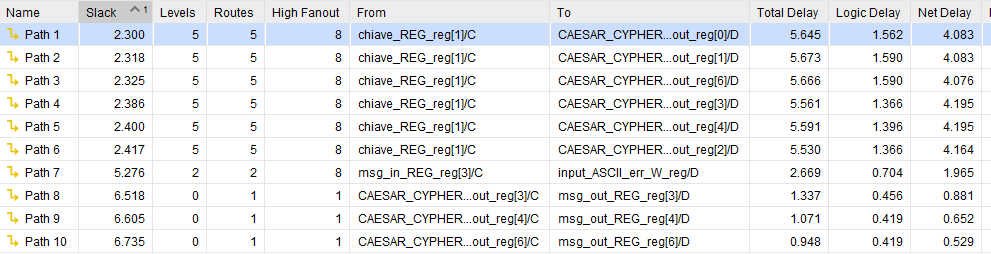


Figure 13 - Critical Path

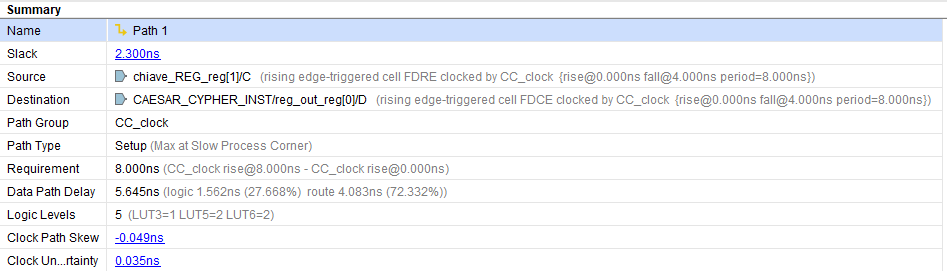


Figure 14 Critical Path Summary

## Power Consumption

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

# Conclusion

By observing that the 72.33% of the Worst-Negative-Slack delay is due to the long route,  
it may be a good idea to follow a Timing-Closure-Design-Methodolody, in order to try to reduce this delay due “only” to the route length.

Alternatively, if there is the need to improve the clock frequency, the circuit as it can reach at most the clock frequency of 175MHz. So, to obtain a better performance, it can be a good idea to break the longest combinatorial path, by inserting in the middle a register. That is the pipeline technique.