# VHDL PROJECT

DANIR Nouhaila - ZAHI Meriem - ZARADA Wadie

May 2024

## 1 Introduction

In this project, we originally planned to implement the RSA algorithm using the VHDL language for synthesis on an FPGA. Although the RSA code is functional in theory, we encountered major problems during its practical implementation on the FPGA.

The main problem lies in the operations required to calculate the PGCD, modular inverse and modular exponentiation. These operations use integers, which cannot be directly synthesized in VHDL because they require complex arithmetic operations. FPGAs are designed to perform arithmetic operations via specialized hardware circuits, but integer operations require transformations into bit-by-bit operations and loops, making them complex to implement.

Because of these technical difficulties, we decided to change our approach, opting for Caesar cipher. Although the latter is less secure than RSA, it is much simpler to implement in VHDL.

So, although the RSA code works, the constraints of synthesizing on FPGA led us to choose a more practical solution, adapted to the resources available, using Caesar cipher.

## 2 RSA Algorithm

RSA (Rivest-Shamir-Adleman) is one of the most widely used algorithms for public-key cryptography. It is named after its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman, who introduced the algorithm in 1977. RSA is used to secure sensitive data, particularly when being sent over an insecure network like the internet.

RSA relies on the mathematical properties of prime numbers and modular arithmetic. Here's a high-level overview of how the RSA algorithm functions:

-**Prime Numbers**: Select two large prime numbers, $p$ and $q$.
- **Modulus**: Compute $n = p \times q$. The number $n$ is used as the modulus for both the public and private keys.
- **Totient**: Compute the totient $\phi(n) = (p-1) \times (q-1)$.
- **Public Key Exponent**: Choose an integer $e$ such that $1 < e < \phi(n)$ and $e$ is co-prime with $\phi(n)$.

- **Private Key Exponent**: Compute $d$ as the modular multiplicative inverse of $e$ modulo $\phi(n)$, i.e., $d \times e \equiv 1 \ (mod \ \phi(n))$.
The public key consists of the pair $(e, n)$, and the private key consists of the pair $(d, n)$.
- To encrypt a message $M$, convert $M$ to an integer $m$ such that $0 \leq m < n$.
- Compute the ciphertext $c$ using the public key: $c \equiv m^e \ (mod \ n)$.
- To decrypt the ciphertext $c$, compute the original message $m$ using the private key: $m \equiv c^d \ (mod \ n)$.
- Convert the integer $m$ back to the original message $M$.

The security of RSA is based on the computational difficulty of factoring the product of two large prime numbers. As of now, no efficient algorithm exists for factoring large numbers into their prime factors, making RSA secure when sufficiently large primes are used.

Implementing RSA in hardware, such as on an FPGA, presents several challenges:
- **Integer Arithmetic**: RSA involves complex operations on large integers, including modular exponentiation, which are not directly supported by standard VHDL operations.
- **Resource Utilization**: Efficiently using FPGA resources to perform large integer arithmetic can be challenging and may require custom-designed hardware modules.

Due to these challenges, simpler algorithms like the Caesar cipher are sometimes used for educational purposes or where cryptographic strength is not a critical concern. The Caesar cipher involves only basic arithmetic operations, making it much easier to implement on hardware platforms.

# 3 RSA VHDL Code

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity testx is
6      port (
7          message    : in  std_logic_vector(87 downto 0);  --
               11 characters * 8 bits
8          p          : in  unsigned(7 downto 0);
9          q          : in  unsigned(7 downto 0);
10         n          : buffer unsigned(15 downto 0);
11         phi_n      : buffer unsigned(15 downto 0);
12         e          : buffer unsigned(7 downto 0);
13         d          : buffer unsigned(7 downto 0);
```

```
14          resultat  : out std_logic_vector(87 downto 0)  --
                Encrypted message
15      );
16  end entity testx;
17
18  architecture behavioral of testx is
19
20      function gcd(a, b : integer) return integer is
21          variable temp_a : integer := a;
22          variable temp_b : integer := b;
23      begin
24          for i in 1 to 100 loop
25              if temp_a = 0 then
26                  return temp_b;
27              elsif temp_b = 0 then
28                  return temp_a;
29              elsif temp_a < temp_b then
30                  temp_b := temp_b - temp_a;
31              else
32                  temp_a := temp_a - temp_b;
33              end if;
34          end loop;
35          return -1; -- Indication d'un d passement de la
                limite d'it ration
36      end function;
37
38      function mod_inverse(a : integer; m : integer) return
            integer is
39          variable y : integer := 0;
40          variable x : integer := 1;
41          variable m0 : integer := m;
42          variable a_temp : integer := a;
43          variable iteration_count : integer := 0;  --
                Compteur d'it rations
44      begin
45          while a_temp > 1 loop
46              iteration_count := iteration_count + 1;  --
                    Incr menter le compteur d'it rations
47              if iteration_count > 100 then  -- V rifier si
                    le nombre d'it rations a atteint la limite
48                  return -1;  -- Indication d'un d passement
                        de la limite d'it ration
49              end if;
50
51              y := x - (a_temp / m) * y;
52              x := y + x;
53              a_temp := m0 - (m0 / a_temp) * (m0 mod a_temp);
54              m0 := a_temp;
55          end loop;
56          if x < 0 then
```

```
57              x := x + m;
58          end if;
59          return x;
60      end function;
61
62      function mod_exp(base : integer; exponent : integer;
            modulus : integer) return integer is
63          variable result : integer := 1;
64          variable b : integer := base mod modulus;
65          variable e : integer := exponent;
66          variable iteration_count : integer := 0;  --
                Compteur d'it rations
67      begin
68          while e > 0 loop
69              iteration_count := iteration_count + 1;  --
                    Incr menter le compteur d'it rations
70              if iteration_count > 100 then  -- V rifier si
                    le nombre d'it rations a atteint la limite
71                  return -1;  -- Indication d'un d passement
                        de la limite d'it ration
72              end if;
73
74              if (e mod 2) = 1 then
75                  result := (result * b) mod modulus;
76              end if;
77              e := e / 2;
78              b := (b * b) mod modulus;
79          end loop;
80
81          return result;
82      end function;
83
84  begin
85
86      process(p, q, message)
87          variable temp_p : integer;
88          variable temp_q : integer;
89          variable temp_n : integer;
90          variable temp_phi_n : integer;
91          variable temp_e : integer := -1; -- Initialisation
                avec une valeur par d faut
92          variable temp_d : integer;
93          variable temp_m : integer;
94          variable temp_x : integer;
95          variable temp_result : std_logic_vector(87 downto 0)
                ;
96      begin
97          temp_p := to_integer(p);
98          temp_q := to_integer(q);
99
```

```
100         temp_n := temp_p * temp_q;
101         temp_phi_n := (temp_p - 1) * (temp_q - 1);
102
103         n <= to_unsigned(temp_n, 16);
104         phi_n <= to_unsigned(temp_phi_n, 16);
105
106         -- Trouver e, un nombre premier avec phi(n)
107         for i in 2 to 1000 loop
108             if gcd(i, temp_phi_n) = 1 then
109                 temp_e := i;
110                 exit;
111             end if;
112         end loop;
113
114         if temp_e = -1 then
115             -- Aucun nombre premier trouv , g rer l'erreur
116             -- Vous pouvez choisir une autre valeur par
                    d faut ou signaler une erreur
117             null;
118         else
119             e <= to_unsigned(temp_e, 8);
120
121             -- Calculer l'inverse modulaire d
122             temp_d := mod_inverse(temp_e, temp_phi_n);
123             d <= to_unsigned(temp_d, 8);
124
125             -- Chiffrer le message
126             for i in 0 to 10 loop
127                 temp_m := to_integer(unsigned(message(87 -
                        8*i downto 80 - 8*i)));
128                 temp_x := mod_exp(temp_m, temp_e, temp_n);
129                 temp_result(87 - 8*i downto 80 - 8*i) :=
                        std_logic_vector(to_unsigned(temp_x, 8));
130             end loop;
131
132             resultat <= temp_result;
133         end if;
134     end process;
135
136 end architecture behavioral;
```

# 4 Caesar cipher CODE

Due to the constraints outlined in the introduction of our project, we have made the decision to employ the Caesar cipher , as discussed earlier. This encryption method presents several advantages in terms of security and practicality, particularly within the context of the specific constraints we have identified. Consequently, it stands as the most suitable choice for our application.

Here is the code we used :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cesar is
    port (
        m : in  std_logic_vector(87 downto 0);  -- 11 caract res * 8 bits pour le message d'entr e
        k : in  integer;                        -- Cl  de d calage pour le chiffrement
        s : out std_logic_vector(87 downto 0)  -- Message chiffr
    );
end entity cesar;

architecture Behavioral of cesar is
begin
    process(m, k)
        variable temp_output : std_logic_vector(87 downto 0);
        variable current_char : std_logic_vector(7 downto 0);
        variable char_value : integer;
        variable new_value : integer;
        constant LOWER_A : integer := 97;
        constant LOWER_Z : integer := 122;
        constant UPPER_A : integer := 65;
        constant UPPER_Z : integer := 90;
    begin
        for i in 0 to 10 loop
             -- Extraire le caract re courant
            current_char := m((i+1)*8-1 downto i*8);

            -- Convertir le caract re courant en valeur enti re
            char_value :=
                to_integer(unsigned(current_char));

            -- Chiffrer le caract re s'il est alphab tique
            if char_value >= LOWER_A and char_value <= LOWER_Z then
                new_value := LOWER_A + (char_value - LOWER_A + k) mod 26;
            elsif char_value >= UPPER_A and char_value <= UPPER_Z then
                new_value := UPPER_A + (char_value - UPPER_A + k) mod 26;
```

```
37              else
38                  new_value := char_value;  -- Les
                        caract res non alphab tiques restent
                        inchang s
39              end if;
40
41              -- Convertir la nouvelle valeur enti re en
                    std_logic_vector
42              temp_output((i+1)*8-1 downto i*8) :=
                    std_logic_vector(to_unsigned(new_value, 8));
43          end loop;
44
45          -- Assigner la sortie temporaire au port de sortie
                r el
46          s <= temp_output;
47      end process;
48  end architecture Behavioral;
```

**note**:
•Alphabetic Character Wrapping:
For lowercase letters ('a' to 'z'), the ASCIl range is 97 to 122. The wrapping is done by subtracting 97, applying the shift, taking the modulo 26, and then adding 97 back.
For uppercase letters ('A' to 'Z'), the ASCIl range is 65 to 90. The wrapping is done by subtracting 65, applying the shift, taking the modulo 26, and then adding 65 back.
• Non-Alphabetic Characters:
Non-alphabetic characters remain unchanged.
•Shift Adjustment:
The shift is adjusted to be within the range of 0 to 25 for proper alphabetic wrapping.

# 5 SIMULATION

*EXAMPLE 1 :*

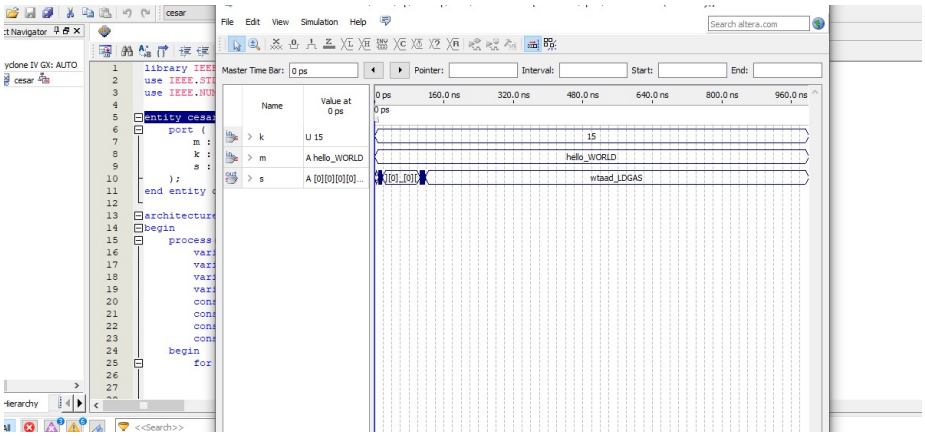


Figure 1: Description of Example 1.

*EXAMPLE 2 :*

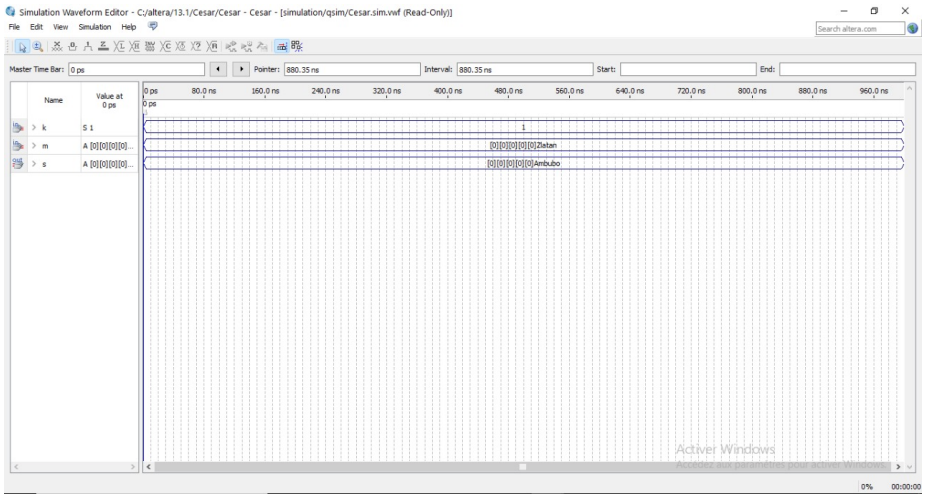


Figure 2: Description of Example 2.

# 6 CONCLUSION

The RSA algorithm, while powerful, presents particular challenges when implemented in FPGA hardware due to the limitations of integer arithmetic operations in hardware description languages like VHDL. Here is an explanation of the main issues encountered and a simple alternative solution, the Caesar cipher.
In VHDL, integer operations are not synthesizable because they require complex arithmetic operations, which need to be implemented by specific hardware circuits on an FPGA. To make these functions synthesizable, they must be rewritten using bitwise operations and loops, which significantly increases complexity.
To circumvent these complex challenges, we used the Caesar cipher, which is much simpler to implement in FPGA hardware.
the Caesar cipher offers a much simpler alternative. The Caesar cipher, though basic and not secure for serious applications, is an excellent example of a substitution cipher that is easy to understand and implement.