NTNU – Trondheim
Norwegian University of
Science and Technology

TFE4141 Design of digital systems 1
Project Report

RSA encryption implemented in VHDL

Group 02:

Anders Nilsen
Torgeir Leithe

May 29, 2018

1. Overview

The main goal for the exercise was to implement a 128-bit RSA encryption/decryption module on an FPGA, using an algorithm the group found to be the most fitting in regards to speed, power consumption and area. The module was to be designed using Verilog with VHDL code, and tested using a testing server set up for the project, running the code on an FPGA.

1.1. RSA

The RSA algorithm has its name from its creators; Ron Rivest, Adi Shamir and Len Adleman. Public-key cryptography, also known as asymmetric cryptography, uses two different but mathematically linked keys, one public and one private. Both the public and the private keys can encrypt a message, and the opposite key must be used to decrypt it.

A key pair consists of three parts:
Public key: {e, n}
Private key: {d, n}
Where n is the product of two primes, and e and d are two coprimes[1] smaller than n.

The method to encrypt:

$$C = M^e \ \% \ n, M < n \tag{1.1}$$

To decrypt:

$$M = C^d \ \% \ n \tag{1.2}$$

Observing the similarity of equation 1.1 and 1.2, reveals that the algorithms for encryption and decryption are mathematically equivalent.

1.2. Requirements

A series of requirements was given in regards to the RSA implementation:

- The design must implement the RSA encryption algorithm
- Encrypt/decrypt a message of length 128 bits as fast as possible
- The target frequency is 50MHz

[1] Two integers having no positive integer factors in common, aside from 1

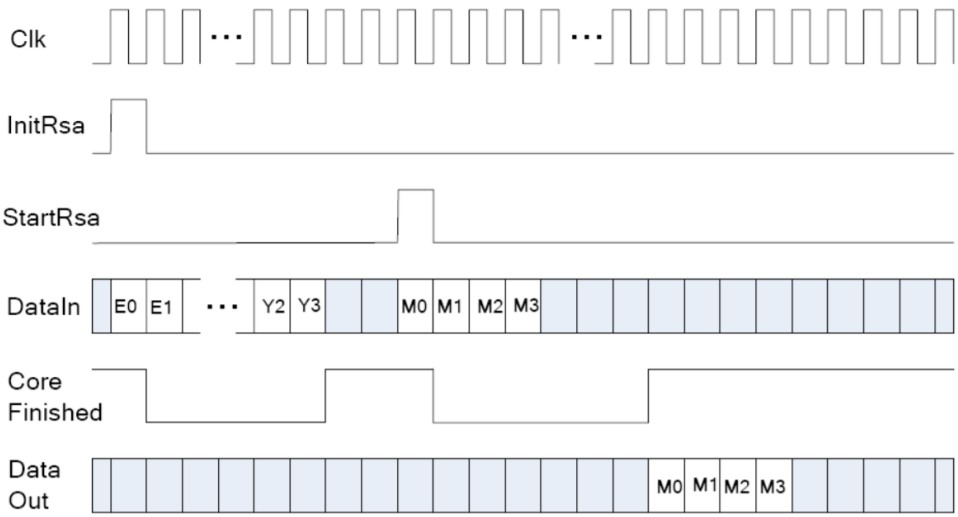

Figure 1.1.: interface

- The design must use less that 50% of the resources in a Xilinx Zynq®-7000 device
- The design entity declaration must match listing 1.1
- The design must implement the interface in figure 1.1

Listing 1.1: entity declaration

```
component RSACore is
  port (
    Clk             :  in std_logic;
    Resetn          :  in std_logic;
    InitRsa         :  in std_logic;
    StartRsa        :  in std_logic;
    DataIn          :  in std_logic_vector(31 downto 0);
    DataOut         : out std_logic_vector(31 downto 0);
    CoreFinished    : out std_logic
  );
end component RSACore;
```

2. RSA algorithms

The naive method to solving equation 1.1 would be to first compute $tmp = m^e$ and then find the reminder $C = tmp \ \% \ n$. This is because the temporary variable could in the worst case be $2^{128} \cdot 128 \approx 4 \cdot 10^{40}$bits. Therefore, a more efficient method for computing equation 1.1 is needed.

Different efficient algorithms for solving this problem already exists.

2.1. Binary method

The simplest improvement to this method is to do the modulus operation for each step when calculating M^e, by calculating $(C = M * C \bmod n)$, e number of times. This removes the problem of getting extremely lager numbers, but still requires many multiplications to get to the answer.

$$M^{55} = M^1 \rightarrow M^2 \rightarrow M^3 \rightarrow M^4 \rightarrow M^5 \rightarrow M^6 \rightarrow M^7 \cdots \rightarrow M^{55} \tag{2.1}$$

This can be further improved by either squaring C, or multiplying by M:

$$M^{55} = M^1 \rightarrow M^2 \rightarrow M^3 \rightarrow M^6 \rightarrow M^{12} \rightarrow M^{13} \rightarrow M^{26} \rightarrow M^{27} \rightarrow M^{54} \rightarrow M^{55} \tag{2.2}$$

A algorithm for determining this behavior is given in [1, p. 10].

Listing 2.1: Simplified brute force exponentiation

```
def ModExp(M, e, n)
    if e[k-1]: C = M
    else       C = 1

    for i in range(k-2 to 0):
        C=C*C mod n
        if e[i]:
            C=C*M mod n
    return C
```

The speed of this method is determined by the speed of computing $A*B \ mod \ n$. Blakely's method[1, p. 45] is a way of computing this.

Listing 2.2: Blakely's method

```
def ModMult(A,B,n)
    P=0
    for i in range(0,k-1):
        if B[k-1-i]:
            P= 2P + A
        else:
            P = 2P
        if P >= n:
            P -= n
        if P >= n:
            P -= n
    return P
```

2.2. Montgomery reduction algorithm

In 1985, P.L. Montgomery introduced an efficient algorithm for computing $AB \ \% \ n$. This algorithm replaces the need for the division by n operation with division by a power of two. This can be done because the original $AB \ \% \ n$ has been mapped to a "Montgomery plane". Given two n-residues $\bar{a}$ and $\bar{b}$, the Montgomery product is defined as the n-residue.

$$MonPro(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot r^{-1} (mod \ n) \tag{2.3}$$

where r^{-1} is such that

$$r^{-1} \cdot r = 1 (mod \ n) \tag{2.4}$$

and $r = 2^k$, where k is the number of bits in n.[1]

An efficient binary add-shift algorithm for computing MonPro(A, B) is given in listing 2.3.[2, p. 23]

Listing 2.3: Montgomery product

```
def MonPro(A, B, n):
    u = 0
    for i in range(0 to k-1):
        if a[i]:
            u += B
        if (u % 2) = 1:
            u += n
        u = u/2
```

[1]This is the actual number of bits needed to represent n, leading zeros are not counted

```
    if u > n:
        return u-n
    return u
```

An important property to note is that whether n needs to be added can be pre-calculated by using $u[0] \oplus A[i]B[0]$

A method for utilizing the Montgomery product to calculate $M^e \ mod \ n$ is given in [1, p. 48]:

```
def ModExp(message,e,n,r)
    M=message*r % n
    x=r % n
    for i in range (k-1 downto 0)
        x=MonPro(x,x,n)
        if e[i]:
            x=MonPro(M,x,n)
    return MonPro(x,1,n)
```

However, several changes can be made to improve this method. Due to $r \ mod \ n$(r_n) and $r^2 \ mod \ n$(rr_n) being given as variables in the project, the calculation of M can be changed to:

$$M = message * r \ mod \ n = message * r * r * r^{-1} \ mod \ n$$
$$M = MonPro(message, rr\_n, n)$$

By changing the direction of the for loop, it can be rewritten as:

```
    for i in (0 to k-1):
        if e[i]:
            x = MonPro(M,x,n)
        M = MonPro(M,M,n)
```

This has the advantage that for one passage of the loop, the inputs to one MonPro instance is not dependent one the output of the other. A result of this is that when implemented on an FPGA the two MonPro instances can be run in parallel. This allows for a theoretical doubling of speed, depending on the number of '1's in e.

Another way to speed up the calculation is to note that once the loop has passed the most significant '1' in e, all leading zeros do not change x. Therefor the loop can be stopped earlier when e is not 128 bits long.

A final change that has no speed difference, but simplifies the design, is to calculate x as

$$x = r \ mod \ n = r * r * r^{-1} \ mod \ n$$
$$x = MonPro(1, rr\_n, n)$$

This is possible because there already are two instances of MonPro, and the calculation of x can be run in parallel with the calculation of M. This replaces the separate register for r_n with a MUX on the input to the MonPro.

After implementing these changes, the new ModExp algorithm can be written as shown in listing 2.4:

Listing 2.4: Montgomery Exponentiation

```
def ModExp(message, e, n, rr_n):
    M = MonPro(message, rr_n, n)
    x = MonPro(1,       rr_n, n)

    for i in ragne(0 to len(e)):
        if e[i]:
            x = MonPro(M, x, n)
        M = MonPro(M,M, n)

    return MonPro(x,1,n)
```

3. Implementation and verification

The Montgomery method was chosen because it seamed like the fastest and most interesting way of computing the RSA-encryption algorithm. The first step to implementing the design was testing the Montgomery algorithms using Python. The python code is included as appendix C. This was both to verify the function of the algorithms as well as increase our understanding of them. It was also very useful to have the Python models to compare against the VHDL simulations.

3.1. Design

The next effort was put into creating block schematics based on the Python code. The design processes started with implementing and verifying the innermost function.

3.1.1. MonPro

In this case the MonPro function, the schematic is shown in figure 3.1. A separate testbench was written in VHDL to verify the MonPro function. In the final implementation the two MonPro blocks are combined together, because they share some of the same control logic.

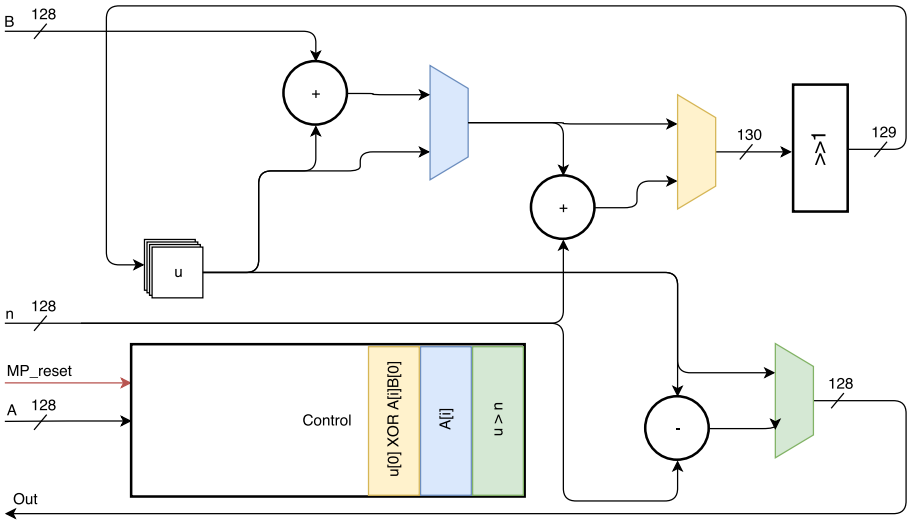

Figure 3.1.: MonPro schematic

3.1.2. ModExp

The next function to be made was the ModExp function, utilizing the already made MonPro function, figure 3.2. The "first" MonPro is the signal used to MUX in the start values to calculate x and M, and the "first or last" MonPro signal is used both to calculate the start value, and the final output value when converting away from the Montgomery plane. Since the MonPro function was already tested, any difference between the VHDL implementation and the Python model was known to be in the ModExp function. This made the debugging simpler.

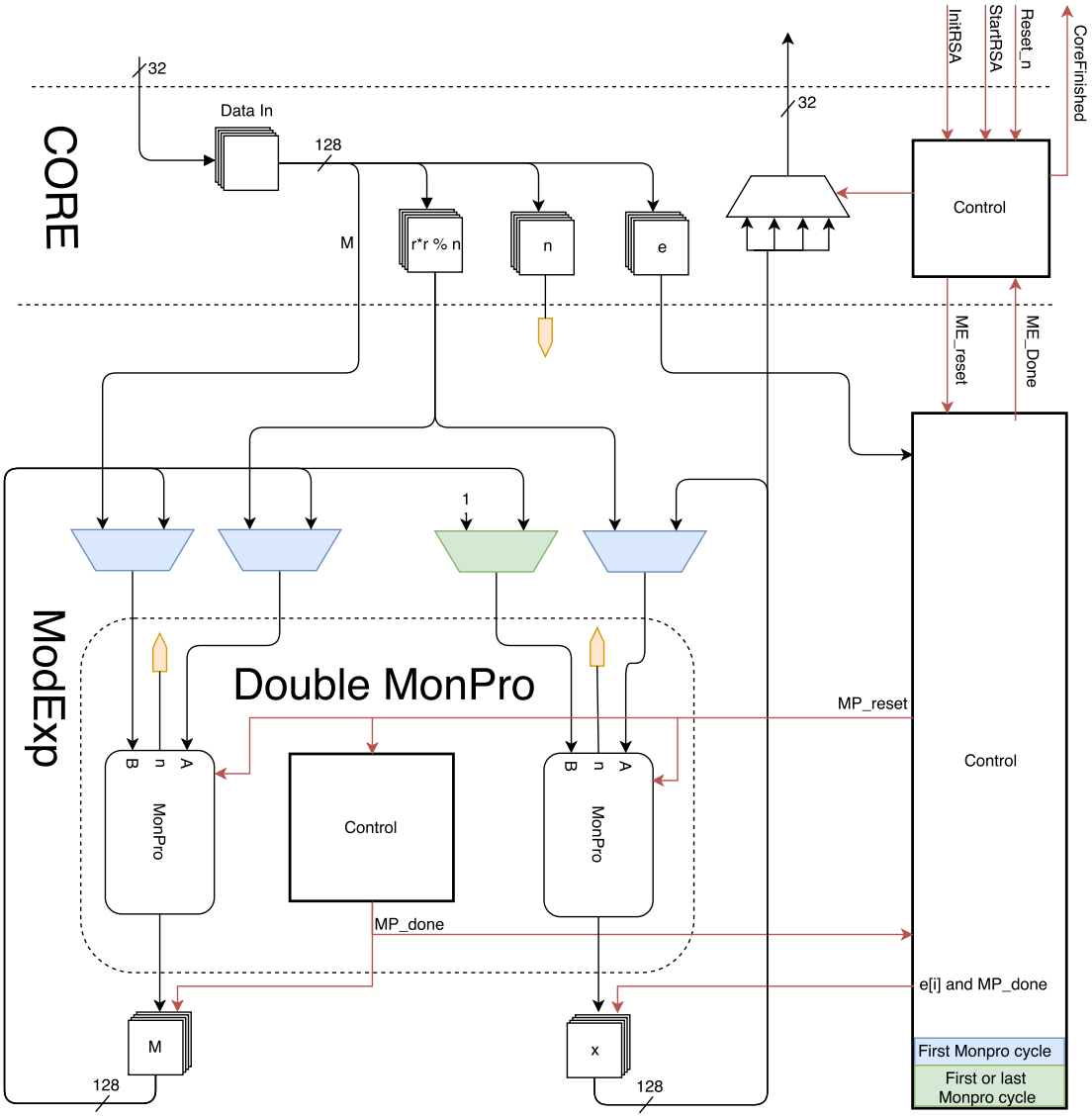

Figure 3.2.: ModExp and core schematic

3.1.3. Core

Finally the core module had to be made, it handled the input and output of data, and the assembling of the 32-bit input data into the correct 128-bit registers. The first input register also works as the M register, which is done by stopping the read in of the input data at the right time.

Since the core model does not do any mathematical processing of the data, there was not made a Python model of this block.

The interface between the modules is part of figure 3.2. The inner modules are reset when the reset signal is high, and start computing once the signal transitions low. To save space the modules do not have registers on the data inputs, therefore the inputs should not be changed while the module is running. When a module is done, its done signal is set high, and at that clock cycle the output value is correct. There is no designated output register, so the higher level module needs to store the value when needed.

3.2. Estimation of performance

The MonPro loop can be preformed in one clook cycle per pass. An additional clock cycle is used to calculate the return value. This gives 129 clock cycles for one use of the MonPro-function, hereafter called MonPro-cycle. The ModExp function uses up to 130 MonPro-cycles. 1 for the initial calculation of x and M, up to 128 MonPro-cycles for the main loop, and 1 MonPro-cycles to calculate the return value. The total is then $129 \cdot 130 = 16770$ clock cycles. In addition to this, a few cycles are needed for data read-in and read-out.

3.3. Estimation of area

The main bulk of area is used by the 128 bit wide data path, the control logic is going to be negligible in comparison.
4 128-bit registers are needed to store the input data and 1 to store the output data. Two 128-bit registers are needed to store M and x in the ModExp function, and one 129-bit registers in each of the MonPro instances. In total 1154 registers. the other main contributing factor are the adders and subtractor in the data path. For each MonPro-block there are 2 129-bit adders, and one 129-bit subtractor.

4. Synthesis and FPGA test

The results from synthesizing in Vivado can be seen in table 4.1, the requirement was less than 50% usage, which this design clearly passes. When synthesized on the FPGA server, the footprint was slightly larger, table 4.2. This might be because of different FPGA technologies or different optimizations during synthesis.

The requirement to achieve a clock frequency of 50 Mhz was fulfilled with a max frequency of 57.86 Mhz on the FPGA server. The Vivado synthesis showed a worst negative slack of 13.973 ns. This was a sign that in Vivado the frequency could be much higher. After testing it by changing the constraints, a clock speed of 140 Mhz was achieved before the worst negative slack became negative.

The average number of clock cycles per block from the FPGA server (table 4.2) was 9624. This number aws less than the worst-case calculation. This was probably because e is less than 128 bits.

Resource	Utilization	Available	Utilization %
LUTs	1455	78600	1.85
FlipFlops	1218	157200	0.77
IO	69	163	42.33
BUFG	1	32	3.13

Table 4.1.: Summary of post-synthesis resource utilization

Average cycles per block:	9624
Number of latches:	0
Logic cells used:	3278
fmax:	57.86 MHz
Throughput:	6012 blocks/second

Table 4.2.: Summary of performance and usage on FPGA

5. Conclusion

When testing with small variations of the design on the FPGA server, it became apparent that seemingly unimportant changes could affect the speed by several MHz.

The design made in this paper satisfies the requirements of the task. It has decent performance, and is not using a huge area. The speed difference between Montgomery and the other solutions is probably not that drastic at 128 bit, but when using larger primes, the difference most likely becomes more significant.

In addition, the time spent on programming the algorithm in Python proved time well spent, as it made testing and verification of the system in VHDL much easier, as well as providing valuable insight in how the algorithm works.

A. Source code

A.1. MonPro.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 18.10.2017 19:49:54
6  -- Design Name:
7  -- Module Name: MonPro - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20
21 library IEEE;
22 use IEEE.STD_LOGIC_1164.ALL;
23 use ieee.numeric_std.all;
24
25 --MonPro-loop is one pass of the for loop in MonPro
26 entity MonPro_loop is
27     Port (
28    clk        : in  STD_LOGIC;
29    reset_n    : in  STD_LOGIC;
30    n_in       : in  STD_LOGIC_VECTOR (127 downto 0);
31    a_in       : in  STD_LOGIC_VECTOR (127 downto 0);
32    b_in       : in  STD_LOGIC_VECTOR (127 downto 0);
33    a_bit      : in  STD_LOGIC_VECTOR (7 downto 0);
34    u_ut       : out STD_LOGIC_VECTOR (127 downto 0));
35 end MonPro_loop;
36
37 architecture Behavioral of MonPro_loop is
38    signal loop_reg     : STD_LOGIC_VECTOR (128 downto 0);
39    signal loop_nxt     : STD_LOGIC_VECTOR (128 downto 0);
40 begin
41
42
43 --This is very ugly, but hella fast, and any atemt to improve it, makes it slower.
44     data_beregning: process(n_in,a_in,b_in,a_bit,loop_reg)
45     variable u_tmp : std_logic_vector(129 downto 0);
46     begin
47     if (((a_in(to_integer(unsigned(a_bit(6 downto 0)))) = '1') AND (b_in(0) =
       '1')) XOR (loop_reg(0) = '1')) then
48       if(a_in(to_integer(unsigned(a_bit(6 downto 0)))) = '1') then
```

```
49          u_tmp := std_logic_vector(( '0' & unsigned(loop_reg))+ unsigned(b_in) +
    unsigned(n_in));
50        else
51          u_tmp := std_logic_vector(( '0' & unsigned(loop_reg))+ unsigned(n_in));
52        end if;
53      elsif(a_in(to_integer(unsigned(a_bit(6 downto 0)))) = '1') then
54          u_tmp := std_logic_vector(unsigned(b_in) + unsigned('0' & loop_reg));
55      else
56          u_tmp := '0' & loop_reg;
57      end if;
58
59      loop_nxt <= u_tmp(129 downto 1);
60      end process;
61
62      u_reg: process(clk,reset_n,loop_nxt) begin
63        if(clk'event and clk = '1') then
64          if(reset_n = '1') then
65            loop_reg <= (others => '0');
66          else
67            loop_reg <= loop_nxt;
68          end if;
69        end if;
70      end process;
71
72     process(loop_reg,n_in)
73     variable u_ut_tmp : std_logic_vector(128 downto 0);
74     begin
75          if (unsigned(loop_reg) > unsigned(n_in)) then
76              u_ut_tmp := std_logic_vector(unsigned(loop_reg) - unsigned(n_in));
77          else
78              u_ut_tmp := loop_reg;
79          end if;
80          u_ut <= u_ut_tmp(127 downto 0);
81      end process;
82
83  end Behavioral;
84
85  library IEEE;
86  use IEEE.STD_LOGIC_1164.ALL;
87  use ieee.numeric_std.all;
88
89  entity MonPro is
90      Port (
91              clk       : in  std_logic;
92              reset_n   : in  std_logic;
93              n_in      : in STD_LOGIC_VECTOR (127 downto 0);
94              a_in1     : in STD_LOGIC_VECTOR (127 downto 0);
95              b_in1     : in STD_LOGIC_VECTOR (127 downto 0);
96              a_in2     : in STD_LOGIC_VECTOR (127 downto 0);
97              b_in2     : in STD_LOGIC_VECTOR (127 downto 0);
98              MP_done   : out std_logic;
99              u_out1    : out STD_LOGIC_VECTOR (127 downto 0);
100             u_out2    : out STD_LOGIC_VECTOR (127 downto 0));
101
102 end MonPro;
103
104 architecture Behavioral of MonPro is
105    signal u_int_ut1 : STD_LOGIC_VECTOR (127 downto 0);
106    signal u_int_ut2 : STD_LOGIC_VECTOR (127 downto 0);
107
108    signal a_bit    : STD_LOGIC_VECTOR (7 downto 0);
109
```

```
110    shared variable n_less_than_128_bit : std_logic := '0'; --Set to 1 to allow "n"
       less than 128-bit (n(msb) == "0").
111                                                          --Design is smaler when
       only supporting n = 128 bit
112 begin
113
114 -- Find when monpro is done
115
116   process(clk,a_bit,reset_n) begin
117     if(clk'event and clk = '1') then
118       if(reset_n = '1') then
119         a_bit <= (others => '0');
120       else
121         a_bit <= std_logic_vector(unsigned(a_bit) + "1");
122       end if;
123     end if;
124   end process;
125
126   process(n_in,a_bit,reset_n)
127     variable a_test : STD_LOGIC_VECTOR(127 downto 0);
128   begin
129    if (n_less_than_128_bit = '1') then
130    a_test := (others => '0');
131    a_test(to_integer(unsigned(a_bit(6 downto 0)))) := '1';
132
133     if (reset_n = '1') then
134       MP_done <= '0';
135     elsif(unsigned(a_test(127 downto 0)) > unsigned(n_in) or (unsigned(a_bit) > "
       01111111")) then
136       MP_done <= '1';
137     else
138       MP_done <= '0';
139     end if;
140   else
141     if (reset_n = '1') then
142       MP_done <= '0';
143     else
144       MP_done <= a_bit(7); --is '1' if a_bit > "01111111"
145     end if;
146   end if;
147   end process;
148
149   loopti_loop1: entity work.MonPro_loop
150   port map (
151     -- Data input interface
152     clk     => clk,
153     reset_n => reset_n,
154     n_in    => n_in,
155     b_in    => b_in1,
156     a_in    => a_in1,
157     a_bit   => a_bit,
158     u_ut => u_int_ut1
159   );
160
161   loopti_loop2: entity work.MonPro_loop
162   port map (
163     -- Data input interface
164     clk     => clk,
165     reset_n => reset_n,
166     n_in    => n_in,
167     b_in    => b_in2,
168     a_in    => a_in2,
```

```
    a_bit   => a_bit,
    u_ut => u_int_ut2
    );

  u_out1 <= u_int_ut1;
  u_out2 <= u_int_ut2;

end Behavioral;
```

A.2. ModExp.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 21.10.2017 20:45:36
-- Design Name:
-- Module Name: ModExp - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------

library IEEE;
library work;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity ModExp is
    Port (
    clk : in STD_LOGIC;
    reset_n : in STD_LOGIC;

    M_in : in STD_LOGIC_VECTOR (127 downto 0);
    e_in : in STD_LOGIC_VECTOR (127 downto 0);
    n_in : in STD_LOGIC_VECTOR (127 downto 0);

    rr_n : in STD_LOGIC_VECTOR (127 downto 0);

    ME_done : out STD_LOGIC;
    M_out : out STD_LOGIC_VECTOR (127 downto 0));
end ModExp;

architecture Behavioral of ModExp is
    signal reset_monpro  : STD_LOGIC;
    signal MP_done_first : STD_LOGIC;
    signal MP_done       : STD_LOGIC;

--Signals for MonPro 1
    signal a_in_1        : STD_LOGIC_VECTOR (127 downto 0);
    signal b_in_1        : STD_LOGIC_VECTOR (127 downto 0);
    signal u_out_1       : STD_LOGIC_VECTOR (127 downto 0);
    signal u_reg_1       : STD_LOGIC_VECTOR (127 downto 0);
```

```
51
52 --Signals for MonPro 2
53     signal a_in_2         : STD_LOGIC_VECTOR (127 downto 0);
54     signal b_in_2         : STD_LOGIC_VECTOR (127 downto 0);
55     signal u_out_2        : STD_LOGIC_VECTOR (127 downto 0);
56     signal u_reg_2        : STD_LOGIC_VECTOR (127 downto 0);
57
58     signal ME_done_int    : STD_LOGIC_VECTOR (1 downto 0);
59     signal loop_count     : STD_LOGIC_VECTOR (6 downto 0);
60 begin
61
62   process (reset_n,MP_done,clk) begin
63     if (reset_n = '1') then
64       MP_done_first <= '0';
65     elsif(clk'event and clk = '1' and MP_done = '1') then
66       MP_done_first <= '1';
67     end if;
68   end process;
69
70   REG_1:process(reset_n,clk,u_out_1,MP_done)
71   begin
72     if (clk'event and clk = '1' and MP_done = '1') then
73       u_reg_1 <= u_out_1;
74     end if;
75   end process;
76
77   Start_MUX:process(M_in,u_reg_1,MP_done_first,rr_n)
78   begin
79       if (MP_done_first = '1') then --First MonPro cycle
80         b_in_1 <= u_reg_1;
81         a_in_1 <= u_reg_1;
82       else                          --Main MonPro cycles
83         b_in_1 <= M_in;
84         a_in_1 <= rr_n;
85       end if;
86   end process;
87
88   process(clk, MP_done,reset_n,reset_monpro) begin
89     if (reset_n = '1') then
90       reset_monpro <= '1';
91     elsif(clk'event and clk = '1')then
92       if (MP_done = '1' and reset_monpro = '0') then
93         reset_monpro <= '1';
94       else
95         reset_monpro <= '0';
96       end if;
97     end if;
98   end process;
99
100    Dobbel_MonPro: entity work.MonPro
101    port map(
102    clk => clk,
103    reset_n => reset_monpro,
104    n_in => n_in,
105
106    MP_done => MP_done,
107    a_in1 => a_in_1,
108    b_in1 => b_in_1,
109    u_out1 => u_out_1,
110
111    a_in2 => a_in_2,
112    b_in2 => b_in_2,
```

```
113     u_out2 => u_out_2
114     );
115
116   --ME done fiks
117   process(MP_done,loop_count,e_in,ME_done_int,reset_n,clk)
118       variable loop_test : std_logic_vector(128 downto 0);
119   begin
120     loop_test := (others => '0');
121     if (MP_done_first = '1') then
122       loop_test(to_integer("0" & unsigned(loop_count) + "1")) := '1';
123     end if;
124
125     if(reset_n = '1') then
126       ME_done_int <= "00";
127     elsif (clk'event and clk = '1' and MP_done = '1') then
128       if (ME_done_int = "01") then
129         ME_done_int <= "11";
130       elsif(unsigned(loop_test) > unsigned(e_in)) then
131         if (ME_done_int = "00") then
132           ME_done_int <= "01";
133         end if;
134       else
135         ME_done_int <= "00";
136       end if;
137     end if;
138   end process;
139
140   MUX_2:process(M_in,u_reg_1,u_reg_2,MP_done_first,rr_n,ME_done_int) begin
141     if (ME_done_int(0) = '1')then    -- One MonPro cycle until done
142        a_in_2 <= x"00000000000000000000000000000001";
143        b_in_2 <= u_reg_2;
144     elsif (MP_done_first = '1') then -- Main part
145        a_in_2 <= u_reg_1;
146        b_in_2 <= u_reg_2;
147     else                             -- First monpro cycle
148       a_in_2 <= x"00000000000000000000000000000001";
149       b_in_2 <= rr_n;
150     end if;
151   end process;
152
153   ME_done <= ME_done_int(1) OR(MP_done and ME_done_int(0));
154
155   REG_2:process(MP_done,u_out_2,loop_count,e_in,MP_done_first,clk)
156   begin
157       if(clk'event and clk = '1' and MP_done = '1') then
158         if (MP_done_first = '0' or e_in(to_integer(unsigned(loop_count))) = '1')
      then
159           u_reg_2 <= u_out_2;
160         end if;
161       end if;
162   end process;
163
164   Loop_counting:process(MP_done,reset_n,loop_count,clk) begin
165     if(reset_n = '1') then
166         loop_count <= (others => '1');
167     elsif(clk'event and clk = '1' and MP_done = '1') then
168         loop_count <= std_logic_vector(unsigned(loop_count) + "1");
169     end if;
170   end process;
171    M_out <= u_out_2;
172
173 end Behavioral;
```

A.3. RSACore.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date: 24.10.2017 12:48:59
6  -- Design Name:
7  -- Module Name: RSACore - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool Versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20
21 --78dui47
22
23 library IEEE;
24 use IEEE.STD_LOGIC_1164.ALL;
25 use IEEE.numeric_std.all;
26
27 entity RSACore is
28     Port (
29  DataIn          : in STD_LOGIC_VECTOR (31 downto 0);
30  DataOut         : out STD_LOGIC_VECTOR (31 downto 0);
31  Clk             : in STD_LOGIC;
32  InitRSA         : in STD_LOGIC;
33  StartRSA        : in STD_LOGIC;
34  CoreFinished    : out STD_LOGIC;
35  Resetn          : in STD_LOGIC);
36 end RSACore;
37
38 architecture Behavioral of RSACore is
39 signal reset_ME : STD_LOGIC;
40
41 signal M_in     : STD_LOGIC_VECTOR (127 downto 0);
42 signal e_in     : STD_LOGIC_VECTOR (127 downto 0);
43 signal n_in     : STD_LOGIC_VECTOR (127 downto 0);
44
45 signal rr_n     : STD_LOGIC_VECTOR (127 downto 0);
46
47 signal ME_done : STD_LOGIC;
48 signal M_out   : STD_LOGIC_VECTOR (127 downto 0);
49 signal Data_in_reg : STD_LOGIC_VECTOR (127 downto 0);
50
51 signal Data_out_reg: STD_LOGIC_VECTOR (127 downto 0);
52
53 signal state    : STD_LOGIC_VECTOR (1 downto 0);
54 signal out_state: std_logic;
55 signal counter      : STD_LOGIC_VECTOR (4 downto 0);
56 signal counter_nxt : STD_LOGIC_VECTOR (4 downto 0);
57
58 begin
59
```

```
process(InitRSA,StartRSA,ME_done,resetn,clk) begin
    if (clk'event and clk = '1') then
      if (resetn = '0') then
        state <= "00";
      elsif(InitRSA = '1') then
        state <= "01";
      elsif(StartRSA = '1') then
        state <= "10";
      elsif (ME_done = '1') then
        state <= "11";
      end if;
   end if;
end process;

process(state,counter) begin
  if(state = "01" and unsigned(counter) > "01111") then
    CoreFinished <= '1';
  elsif (state = "11") then
    CoreFinished <= '1';
  else
    CoreFinished <= '0';
  end if;
end process;

counter_nxt <= std_logic_vector(unsigned(counter) + "1");

process(clk,state,counter_nxt,InitRSA,StartRSA,counter) begin
  if (InitRSA = '1' or StartRSA = '1' or state = "00" or state = "11") then
      counter <= (others => '0');
  elsif (clk'event and clk = '1') then
    if (counter_nxt <= "11000") then
      counter <= counter_nxt;
    end if;
  end if;
end process;

process(DataIn, clk,Data_in_reg,state,counter_nxt) begin
  if (clk'event and clk = '1') then
    if(state /= "10")then
      Data_in_reg <= DataIn & Data_in_reg(127 downto 32);
    elsif(counter_nxt <= "00011")  then
      Data_in_reg <= DataIn & Data_in_reg(127 downto 32);
    end if;
  end if;
end process;

M_in <= Data_in_reg;

process(state, counter) begin
  if((state = "10") and (counter > "00100")) then --Not start ME before data read
    is done
    reset_ME <= '0';
  elsif(state = "11") then
    reset_ME <= '0';
  else
    reset_ME <= '1';
  end if;
end process;

process(counter_nxt, clk, state) begin
  if (clk'event and clk = '1' and state = "01") then
    case counter_nxt is
```

```
      when "00100" => e_in  <= Data_in_reg;
      when "01000" => n_in  <= Data_in_reg;
      when "10000" => rr_n  <= Data_in_reg;
      when others => null;
    end case;
  end if;
end process;

process(ME_done,clk,M_Out,Data_out_reg,out_state,resetn,StartRSA)begin
  if(resetn = '0' or StartRSA = '1') then
    out_state <= '0';
  elsif(clk'event and clk = '1')then
    if(out_state = '0') then
      Data_out_reg <= M_out;
      if(ME_done = '1') then
        out_state <= '1';
      end if;
    else
      Data_out_reg(31 downto 0)  <= Data_out_reg(63  downto 32);
      Data_out_reg(63 downto 32) <= Data_out_reg(95  downto 64);
      Data_out_reg(95 downto 64) <= Data_out_reg(127 downto 96);
      Data_out_reg(127 downto 96)<= Data_out_reg(31 downto 0);
    end if;
  end if;
end process;

DataOut <= Data_out_reg(31 downto 0);

ModExp: entity work.ModExp
    port map (
clk     => clk,
reset_n => reset_ME,

M_in    => M_in,
e_in    => e_in,
n_in    => n_in,

rr_n    => rr_n,

ME_done => ME_done,
M_out   => M_out
);
end Behavioral;
```

B. FPGA output

```
################################################################################
Congratulations group02!

Your design passed the test!
Your countless sleepless nights have finally paid off!

                    ------
                  /      /\
                 /      /##\
                /      /####\
               /      /######\
              /      /########\
             /      /##########\
            /      /#####/\#####\
           /      /#####/++\#####\
          /      /#####/++++\#####\
         /      /#####/\++++\#####\
        /      /#####/  \++++\#####\
       /      /#####/    \++++\#####\
      /      /#####/      \++++\#####\
     /      /#####/        \++++\#####\
    /      /#####/----------\++++\#####\
   /                         \++++\#####\
  /---------------------------\++++\####/
 \+++++++++++++++++++++++++++++++++\##/
  \+++++++++++++++++++++++++++++++++\/
    ````````````````````````````````
Average cycles per block: 9624
Number of latches: 0
Logic cells used: 3278
fmax: 57.86 MHz
Throughput: 6012 blocks/second
```
```

C. RSA Montgomery Python code

```
# -*- coding: utf-8 -*-
#Python code for computing the RSA algorithm using Montgomery multiplication
#with key generation and the case of even modulo (where one of the primes is 2).
"""
Created on Mon Sep 25 19:36:36 2017
@author: anders & torgeir
"""

import random
from datetime import datetime

'''
Euclid's algorithm for determining the greatest common divisor
Use iteration to make it faster for larger integers
'''
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

'''
Euclid's extended algorithm for finding the multiplicative inverse of two numbers
'''
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi/e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2- temp1* x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi

'''
Tests to see if a number is prime.
'''
def is_prime(num):
    if num == 2:
```

```
53          return True
54      if num < 2 or num % 2 == 0:
55          return False
56      for n in xrange(3, int(num**0.5)+2, 2):
57          if num % n == 0:
58              return False
59      return True
60
61  def generate_keypair(p, q):
62      if not (is_prime(p) and is_prime(q)):
63          raise ValueError('Both numbers must be prime.')
64      elif p == q:
65          raise ValueError('p and q cannot be equal')
66      #n = pq
67      n = p * q
68
69      #Phi is the totient of n
70      phi = (p-1) * (q-1)
71
72      #Choose an integer e such that e and phi(n) are coprime
73      e = random.randrange(1, phi)
74
75      #Use Euclid's Algorithm to verify that e and phi(n) are comprime
76      g = gcd(e, phi)
77      while g != 1:
78          e = random.randrange(1, phi)
79          g = gcd(e, phi)
80
81      #Use Extended Euclid's Algorithm to generate the private key
82      d = multiplicative_inverse(e, phi)
83
84      #Return public and private keypair
85      #Public key is (e, n) and private key is (d, n)
86      return ((e, n), (d, n))
87
88  def MonPro(A, B, n):
89      #Converting A to binary
90      a_bin = bin(A)[2:].zfill(len(bin(n)[2:]))
91
92      u = 0
93      for a_bit in a_bin[::-1]:
94          if a_bit == '1':
95              u += B
96
97          if (u % 2) == 1:
98              u += n
99
100         u = u/2
101
102     if u > n:
103         return u-n
104     return u
105
106 def BinExp(message, e, n):
107     k = 0
108     ebin = str(bin(e))[2:]
109     if len(ebin) == 1:
110         C = message
111         return C
112     for k in range (len(ebin)-1, -1, -1):
113         if k == len(ebin)-1:
114             if ebin[k-1] == 1:
```

```
                C = message
            else:
                C = 1
        for i in range(k-2, -1, -1):
            if ebin[i] == 1:
                C = C * message % n
            else:
                C = C * C % n
    return C

def ModInverse(invvar, j):
   y = 1
   for i in range (2, j+1):
       if (2 ** (i-1)) < ((invvar*y) % 2**i):
           y+=2**(i-1)
       else:
           y=y
   return y


def BinSplit(n):
    binsend = []
    binnum = str(bin(n))[2:]
    n_place = 0
    n_count = len(binnum)-1
    while n_place == 0:
        if binnum[n_count] > 0:
            n_place = n_count
        else:
            n_count -= 1
    binodd = binnum[:n_place]
    bineven = binnum[n_place:]
    binsend.append(len(bineven))
    binsend.append(int(binodd,2))
    return binsend

def extended_gcd(a,b):
    out = []
    t = 1; oldt = 0
    r = b; oldr = a
    while r != 0:
        quotient = oldr / r
        (oldr, r) = (r, oldr - quotient*r)
        (oldt, t) = (t, oldt - quotient*t)
    out.append(oldr); out.append(oldt); out.append(r); out.append(t)
    return out

def ModExp(message, e, n):

    r = 2 ** len(bin(n)[2:]) # r = (r mod n) + n
    n_merket = -extended_gcd(r,n)[1]

    #NOTE! "r mod n" and "r*r mod n" is given in the exercise
    M_strek = MonPro(message,(r*r) % n,n)
    x_strek = r % n

    for i in bin(e)[2:][::-1]:
        if i == '1':
            x_strek = MonPro(M_strek,x_strek,n)
        M_strek = MonPro(M_strek,M_strek,n)

    return MonPro(x_strek,1,n)
```

```
177
178 def torge_crypt(pk, M):
179
180     binret=[]
181     #Unpack the key into it's components
182     e, n = pk
183     #Case of even n
184     if n % 2 == 0:
185         binret=BinSplit(n)
186         j = binret[0]
187         q = binret[1]
188         x1 = ModExp(M, e, q)
189         x2val = 2**j
190         x2_1 = M % x2val
191         x2_2 = e % 2**(j-1)
192         x2 = BinExp(x2_1, x2_2, x2val)
193         q_inv = ModInverse(q,j)
194         y = (x2 - x1)*q_inv % x2val
195         x = x1 + q*y
196         return x
197
198     else:
199         return ModExp(M, e, n)
200
201
202
203 if __name__ == '__main__':
204     '''
205     Detect if the script is being run directly by the user
206     '''
207     print "RSA Encrypter/ Decrypter"
208     p = int((raw_input("Enter a prime number (17, 19, 23, etc): ")).replace(",","")
    ))
209     q = int((raw_input("Enter another prime number (Not one you entered above): ")
    ).replace(",",""))
210 #     p = 43
211 #     q = 17
212 #     message = 19
213
214     print "Generating your public/private keypairs now . . ."
215     public, private = generate_keypair(p, q)
216     print "Your public key is ", public ," and your private key is ", private
217     print "Enter a number to encrypt with your private key: "
218     print_string = "Needs to be less then " + str(private[1])+": "
219     message_str = raw_input(print_string)
220
221     message = int(message_str)
222
223 #     private = (5,119)
224 #     public = (77,119)
225
226     print "Message is:", message
227     print "encrypting message with private key", private, ". . ."
228     starttime = datetime.now()
229     encrypted_msg = torge_crypt(private, message)
230     print "Kryptert:",encrypted_msg
231     print "Decrypting message with public key ", public ," . . ."
232     decrypted_msg = torge_crypt(public, encrypted_msg)
233     print "Dekryptert", decrypted_msg
234     print "Tid:", datetime.now() - starttime
235     print ""
236     print "Fasit"
```

```
237     print ""
238 #    Fasit_encrypted_msg = (message ** private[0]) % private[1]
239 #    print Fasit_encrypted_msg
240 #    Fasit_decrypted_msg = (Fasit_encrypted_msg ** public[0]) % public[1]
241 #    print Fasit_decrypted_msg
242 #
243 #    print ""
244 #    if encrypted_msg == Fasit_encrypted_msg and decrypted_msg ==
    Fasit_decrypted_msg:
245 #        print "Woohooo, all pass!!"
246 #    else:
247 #        print "Buuhuu, noe gikk galt"
248
249     endtime = datetime.now()
250
251     print "Time:", endtime-starttime
252
253     print ""
254     print "Fasit"
255     print ""
256     Fasit_encrypted_msg = (message ** private[0]) % private[1]
257     print Fasit_encrypted_msg
258     Fasit_decrypted_msg = (Fasit_encrypted_msg ** public[0]) % public[1]
259     print Fasit_decrypted_msg
260
261     print ""
262     if encrypted_msg == Fasit_encrypted_msg and decrypted_msg ==
    Fasit_decrypted_msg:
263         print "Woohooo, all pass!!"
264     else:
265         print "Buuhuu, noe gikk galt"
```

Bibliography

[1] Çetin Kaya Koç. *High-speed RSA implementation.* RSA labratories, 1994.

[2] Çetin Kaya Koç. *RSA Hardware Implementation.* RSA labratories, 1995.