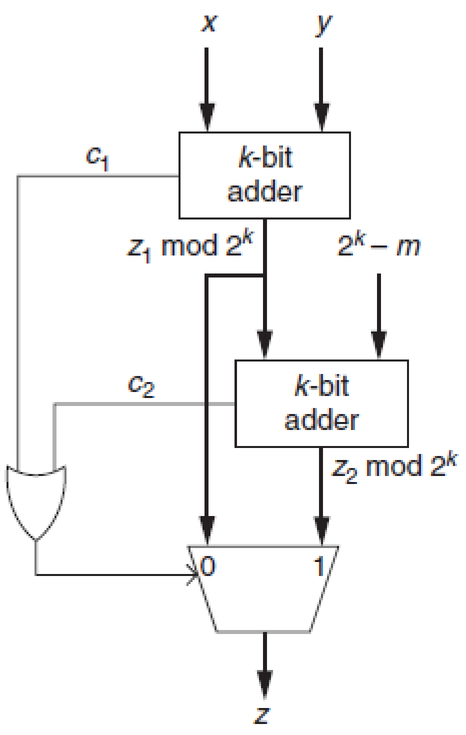
SE520
Cryptography and secure protocols for embedded systems

Submitted by :
Karim Hamod
MD. Shahnauze AHSAN

Introduction:

The goal of this lab is to implement on fpga arithmetic operations based on Z/nZ where the final operator, which is the modular exponentiation, is the main element of the RSA public key encryption algorithm.

1 mod m adder:



1.1) the algorithm used for this adder is:

```
long_x <= '0' & x;
   sum1 <= long_x + y;
   c1 <= sum1(K);
   z1 <= sum1(K-1 downto 0);
-- to be completed
  long_z1 <= '0' & z1;
   sum2 <= long_z1 + minus_M;
   c2 <= sum2(K);
   z2 <= sum2(K-1 downto 0);
   sel <= c1 or c2;
   with sel select z <= z1 when '0', z2 when
others;



constant K: integer := 8;
   constant logK: integer := 3;
   constant integer_M: integer := 239;
   constant M: std_logic_vector(k-1 downto 0) :=
conv_std_logic_vector(integer_M, K);
   constant minus_M: std_logic_vector(K-1 downto 0)
:= conv_std_logic_vector(2**k - integer_M, K);
   constant ZERO: std_logic_vector(logK-1 downto 0)
:= (others => '0');
```

Where here z1 = x+y and z2 = z1 mod 2 power k + 2 power k - m

and result will be finally z= z2 mod 2k when sel not 0

1.2)the complete modm adder code:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;


package dar_modm_multiplier_parameters is
  constant K: integer := 8;
  constant logK: integer := 3;
  constant integer_M: integer := 239;
  constant M: std_logic_vector(k-1 downto 0) :=
conv_std_logic_vector(integer_M, K);
  constant minus_M: std_logic_vector(K-1 downto 0) :=
conv_std_logic_vector(2**k - integer_M, K);
  constant ZERO: std_logic_vector(logK-1 downto 0) := (others =>
'0');
end dar_modm_multiplier_parameters;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.ALL;
use work.dar_modm_multiplier_parameters.all;

entity modm_adder_to_be_completed is
port (
  x, y: in std_logic_vector(K-1 downto 0);
  z: out std_logic_vector(K-1 downto 0)
);
end modm_adder_to_be_completed;

architecture rtl of modm_adder_to_be_completed is
  signal long_x, sum1, long_z1, sum2: std_logic_vector(K downto
0);
  signal c1, c2, sel: std_logic;
  signal z1, z2: std_logic_vector(K-1 downto 0);

begin
  long_x <= '0' & x;
  sum1 <= long_x + y;
  c1 <= sum1(K);
  z1 <= sum1(K-1 downto 0);
-- to be completed
 long_z1 <= '0' & z1;
  sum2 <= long_z1 + minus_M;
  c2 <= sum2(K);
  z2 <= sum2(K-1 downto 0);
  sel <= c1 or c2;
  with sel select z <= z1 when '0', z2 when others;
```

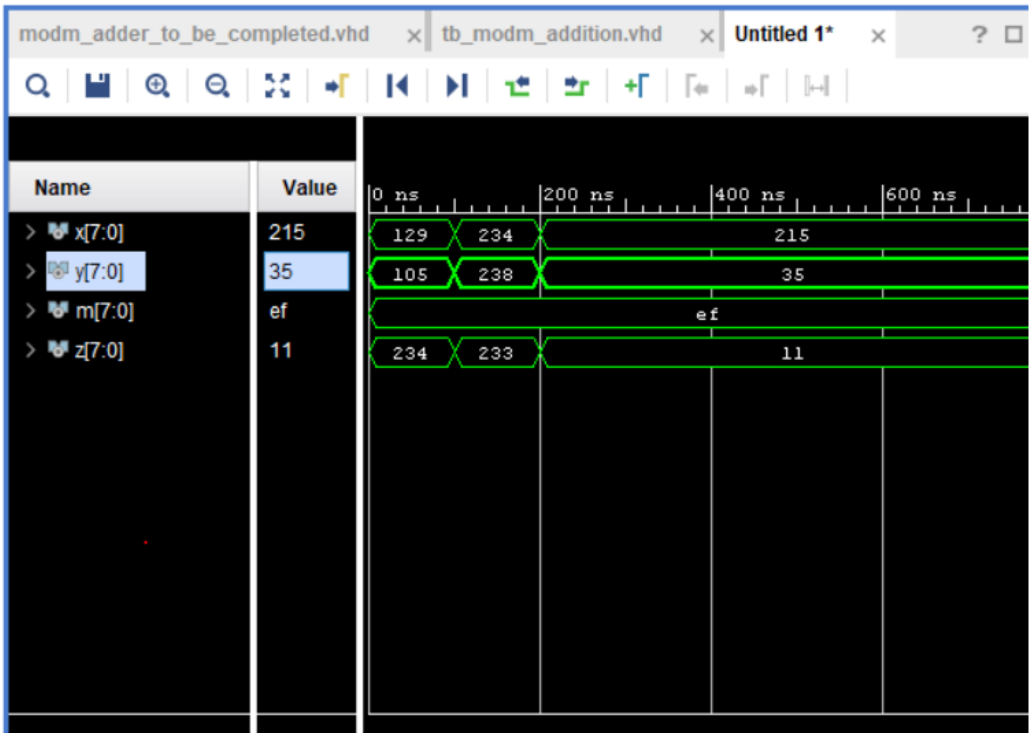
```
end rtl;
```

1.3)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.numeric_std.ALL;

ENTITY tb IS
END tb;

ARCHITECTURE behavior OF tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT modm_adder_to_be_completed
    PORT(
         x : IN  std_logic_vector(7 downto 0);
         y : IN  std_logic_vector(7 downto 0);
         z : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal x : std_logic_vector(7 downto 0) := (others => '0');
   signal y : std_logic_vector(7 downto 0) := (others => '0');
   signal m : std_logic_vector(7 downto 0) := (others => '0');
   signal z : std_logic_vector(7 downto 0) := (others => '0');


BEGIN

     -- Instantiate the Unit Under Test (UUT)
   uut: modm_adder_to_be_completed PORT MAP (
          x => x,
          y => y,
          z => z
        );


   -- Stimulus process
   stim_proc: process
   begin
         x <= std_logic_vector(to_unsigned(129, 8));
```

```
            y <= std_logic_vector(to_unsigned(105, 8));
            m <= std_logic_vector(to_unsigned(239, 8));
      wait for 100 ns;
            x <= std_logic_vector(to_unsigned(234, 8));
            y <= std_logic_vector(to_unsigned(238, 8));
      wait for 100 ns;
            x <= std_logic_vector(to_unsigned(215, 8));
            y <= std_logic_vector(to_unsigned(35, 8));

      wait;
   end process;

END;
```

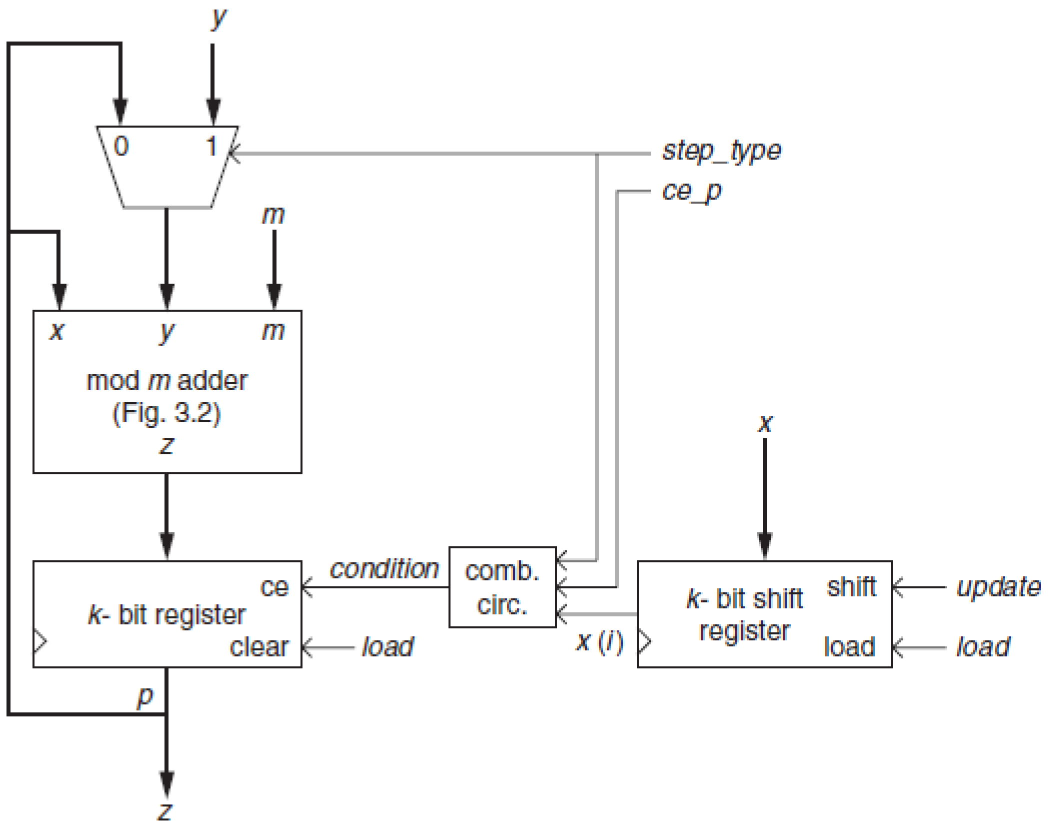
This is the code of the test bench for the modm adder completed
First inputs are x=129,y=105,m=239
Second inputs are x=234,y=238
3rd inputs are x= 215 ,y=35
129+105 mod 239=234 in decimal which is equal to (ea) in base 16
234+238 mod 239 =233
215+35 mod 239 =11



Here we can the simulation of the test bench where the z is displayed in unsigned decimal value. This simulation shows that the modm adder works correctly and the calculation is done correctly.

<u>2 Double, add, and reduce multiplier:</u>

$$condition = ce\_p \wedge (\overline{step\_type} \vee x(i))$$

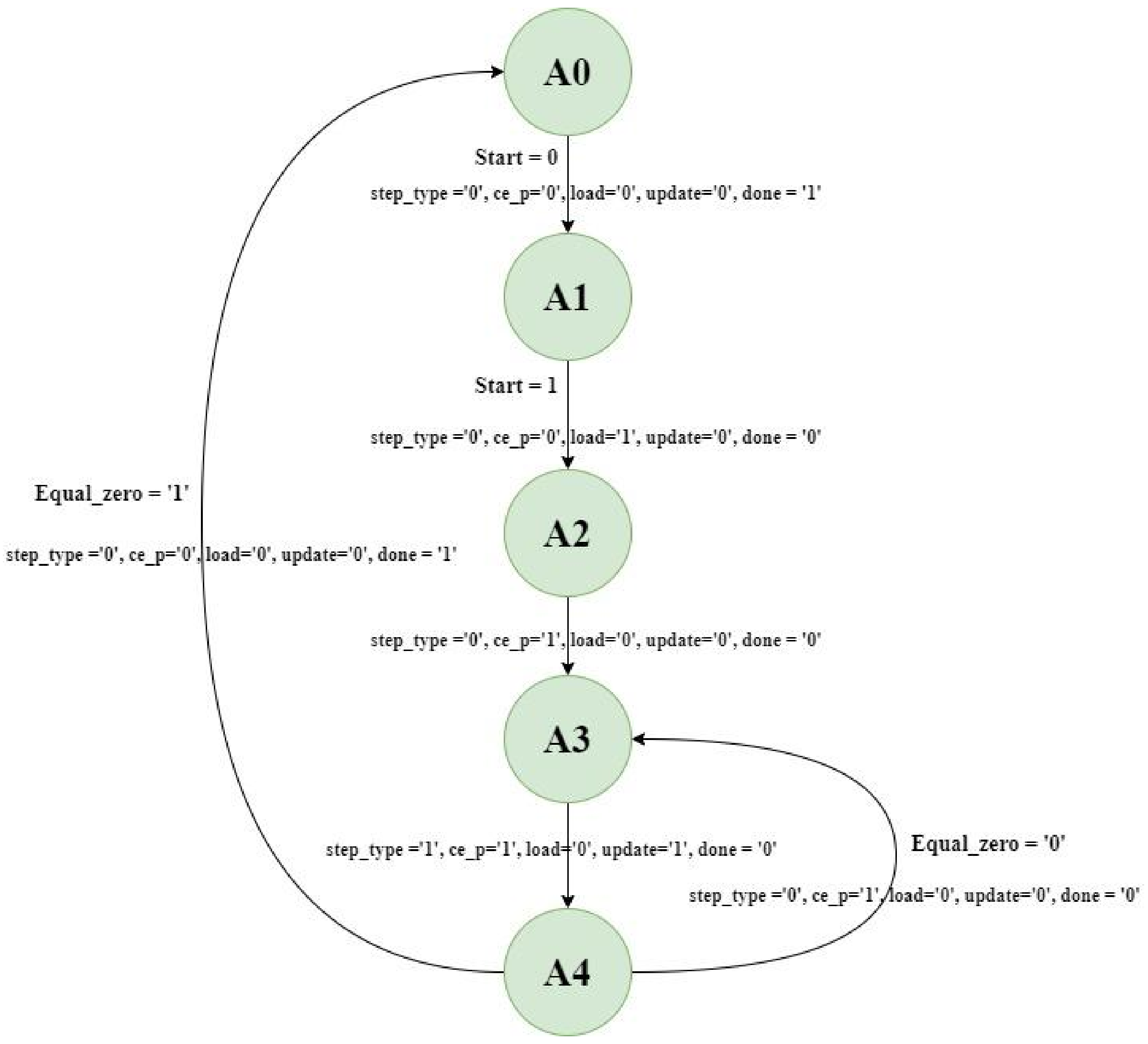


<u>2.1) Finite state machine:</u>

This section describes circuits based on the Interleaving Multiplication Algorithm ([RSDK06]). Given a k-bit natural x and a natural y the product z = xy can be computed as follows:
xy = (xk - 1 2k - 1 + xk - 2 2k - 2 + . . . + x0 20)y = (. . .((0.2 + xk - 1y)2+ xk - 2y)2+ . . . + x1y)2 + x0y

If all operations (addition and doubling) are executed mod m, the result is product = x.y mod m. The corresponding (left to right) algorithm is:

```
p := 0 ;
for i in 0 .. k-1 loop
p := (p*2 + x(k-i-1)*y) mod m;
end loop;
product := p;
```


A0
Start = 0
step_type ='0', ce_p='0', load='0', update='0', done = '1'
A1
Start = 1
step_type ='0', ce_p='0', load='1', update='0', done = '0'
A2
step_type ='0', ce_p='1', load='0', update='0', done = '0'
A3
step_type ='1', ce_p='1', load='0', update='1', done = '0'
A4
Equal_zero = '1'
step_type ='0', ce_p='0', load='0', update='0', done = '1'
Equal_zero = '0'
step_type ='0', ce_p='1', load='0', update='0', done = '0'

The finite state machine is a mealy finite state machine and we where able to deduce that through the provided code of the double add reduce multiplier where the output is depended on the present state and on the inputs.

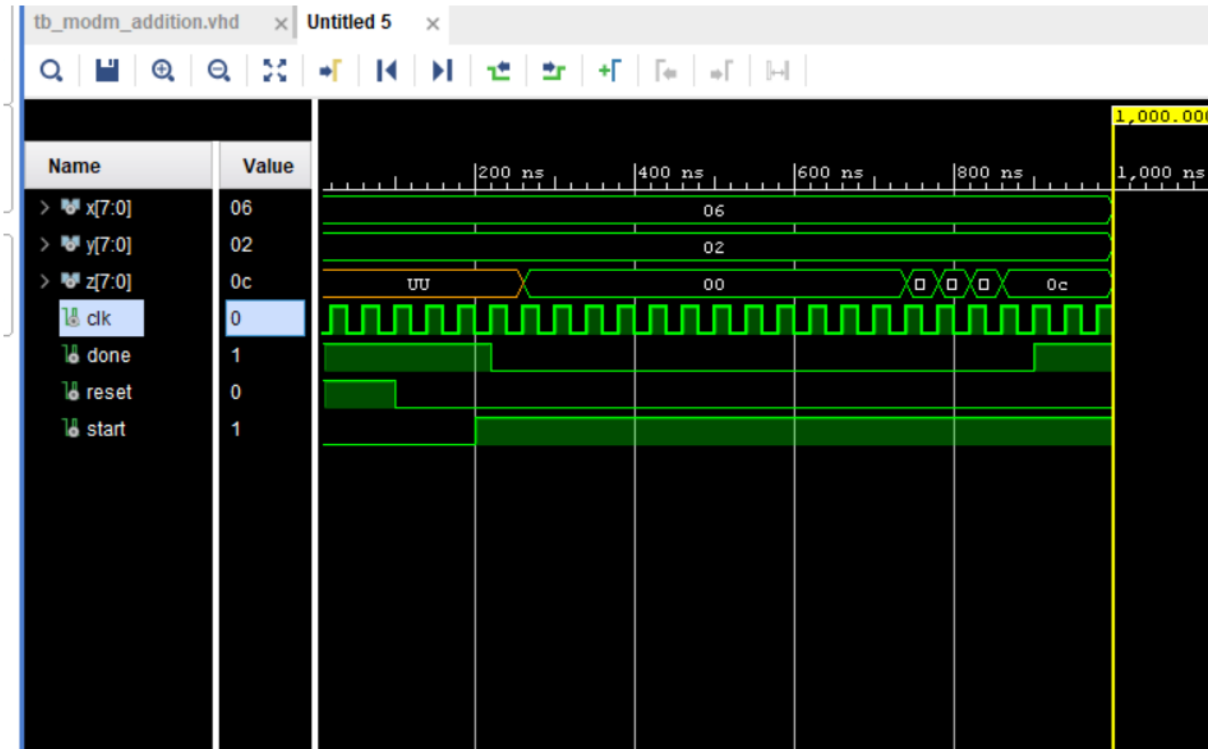
Regarding the reset its not affected by the clock thus its asynchronous

2.2)The multiplication is carried out by multiplying a binary number with 2 and shifting one bit from left to right.
2.3) the test bench for the dar modm multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.ALL;
use work.dar_modm_multiplier_parameters.all;
ENTITY tb IS
END tb;

ARCHITECTURE behavior OF tb iS

    -- Component Declaration for the Unit Under
Test (UUT)

 component dar_modm_multiplier is
    port (
      x, y: in std_logic_vector(K-1 downto 0);
      clk, reset, start: in std_logic;
      z: out std_logic_vector(K-1 downto 0);
      done: out std_logic
    );
 end component;
   --Inputs
   signal x : std_logic_vector(7 downto 0) :=
(others => '0');
   signal y : std_logic_vector(7 downto 0) :=
(others => '0');

   signal z : std_logic_vector(7 downto 0) :=
(others => '0');
   signal clk,done,reset, start:  std_logic :=
'0';

BEGIN
     clk <= NOT(clk) AFTER 10 NS;
    -- Instantiate the Unit Under Test (UUT)
  uut: dar_modm_multiplier PORT MAP (
            x => x,
            y => y,
            z => z,

            reset=> reset,
            start => start,
            clk => clk,
            done => done

          );

   -- Stimulus process
    stim_proc: process
    begin
         reset <= '1';
         start <= '0';
         x <= std_logic_vector(to_unsigned(6, 8));
         y <= std_logic_vector(to_unsigned(2, 8));

       wait for 100 ns;
```

```
         reset <= '0';
       wait for 100 ns;
         start <= '1';
       wait;

   end process;

END;
```

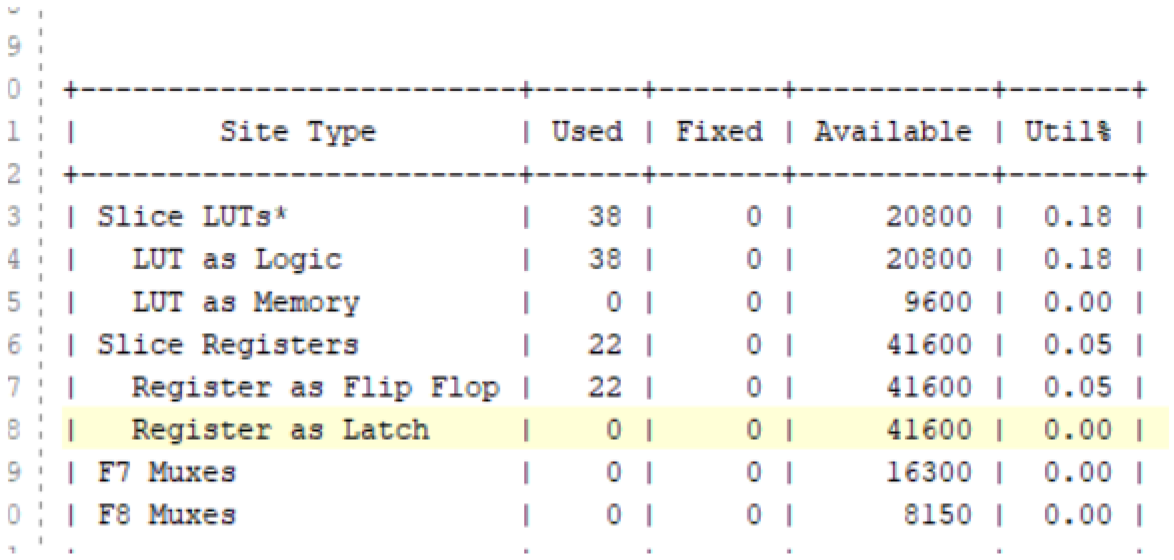
2.4)number of clock cycles required is 2K+1 here K=8 thus it needs 17 clock cycles to finish computation



Name	Value
x[7:0]	06
y[7:0]	02
z[7:0]	0c
clk	0
done	1
reset	0
start	1

2.5)according to simulation result it indeed took 17 clock cycles to reach the final value.

2.6)



```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |   38 |     0 |     20800 |  0.18 |
|   LUT as Logic          |   38 |     0 |     20800 |  0.18 |
|   LUT as Memory         |    0 |     0 |      9600 |  0.00 |
| Slice Registers         |   22 |     0 |     41600 |  0.05 |
|   Register as Flip Flop |   22 |     0 |     41600 |  0.05 |
|   Register as Latch     |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                |    0 |     0 |     16300 |  0.00 |
| F8 Muxes                |    0 |     0 |      8150 |  0.00 |
```

Number of flip flops used is 22
Since we have
8 for x and y
3 for state and count thus 8+8+3+3= 22
Which is also verified in simulation

<u>3 Exponentiation: yx mod m:</u>

Exponentiation is the basic function of the RSA public key encryption algorithm. Assume that m is a k-bit number and that x is represented in base 2, that is x = xk-1 2k-1 + xk-2 2k-2 + . . . + x0 · 20. Then z can be written in the form
z = ((. . . (12 yxk-1)2 yxk-2)2 . . .)2 yx1)2 yx0 mod m
to which corresponds the following algorithm:
Base 2 mod m exponentiation, MSB-first

```
e := 1;
for i in 1 .. k loop
e := (e*e) mod m ;
if binary_x(k-i) = 1 then
e := (e*y) mod m;
end if;
end loop;
z := e;
```

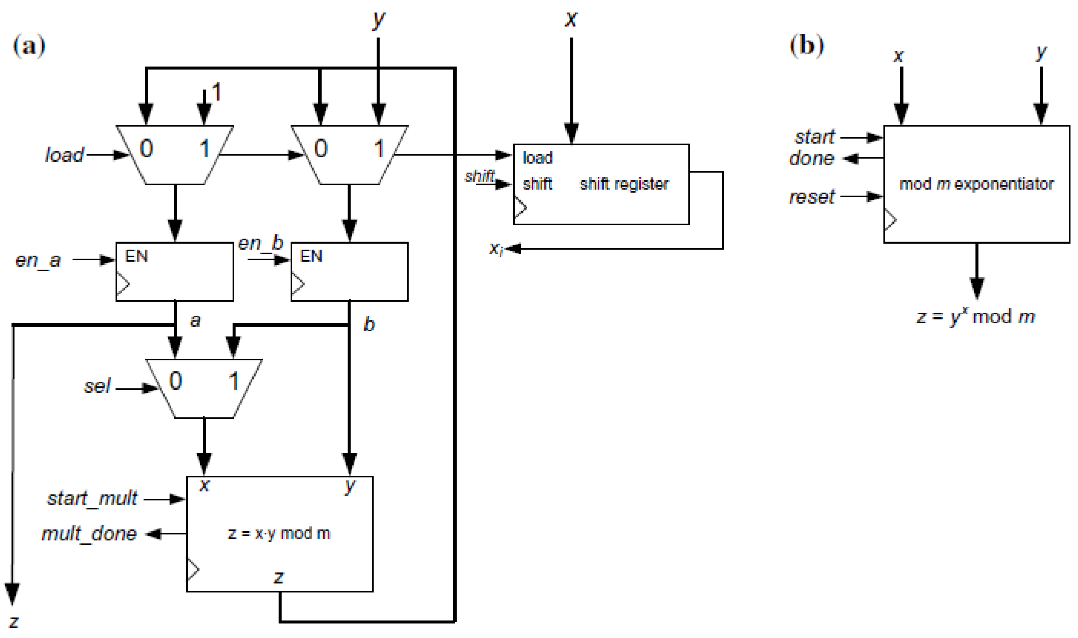
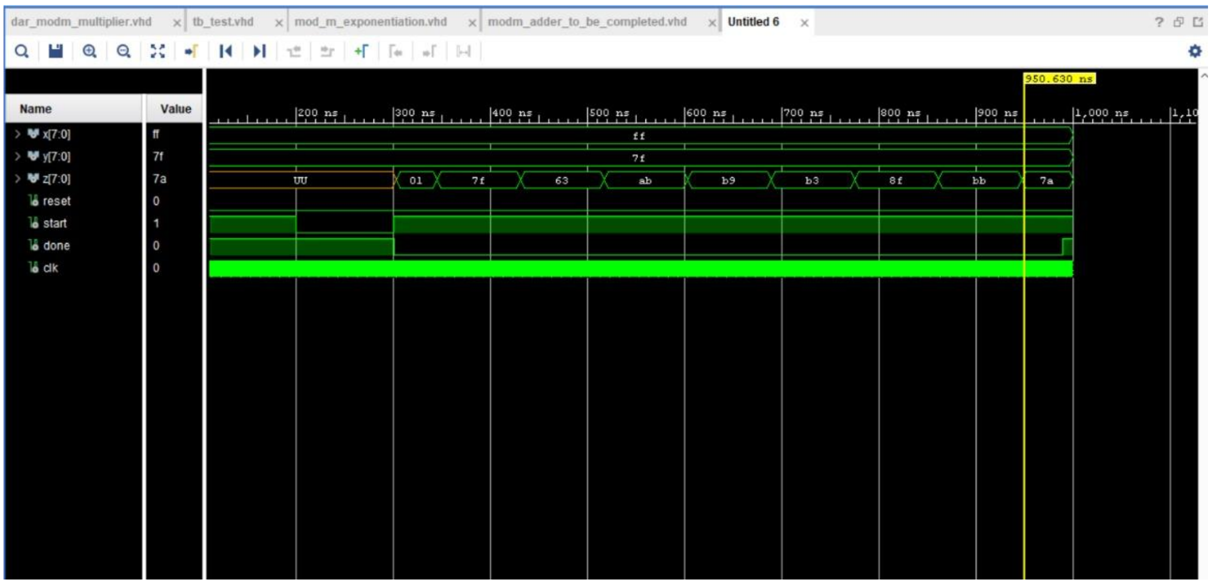


Fig. 6.10 Data path and symbol of a circuit that computes $z = y^x \bmod m$

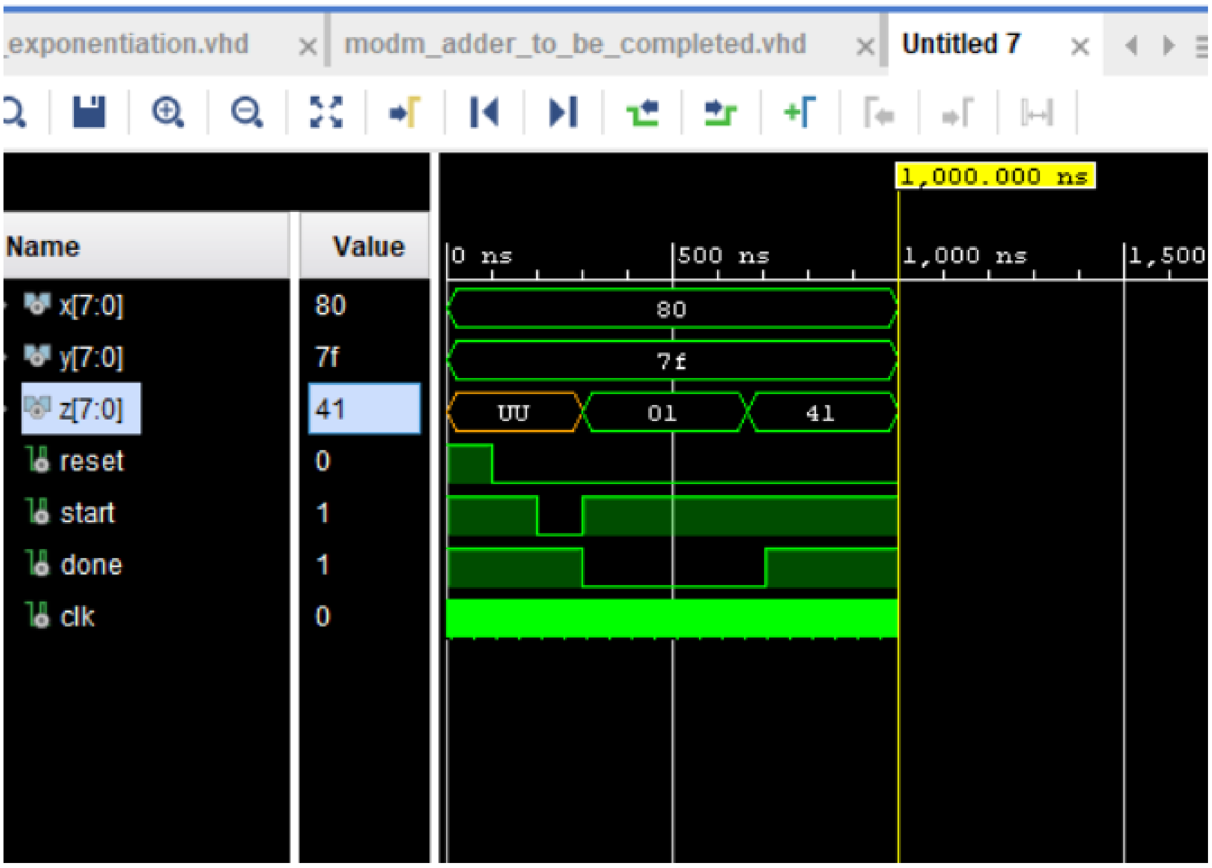
3.1)This is the test bench for the exponentiation

```
LIBRARY IEEE; USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.mod_mm_exponentiation_parm.ALL;
ENTITY test_mod_mm_exponentiation IS END test_mod_mm_exponentiation;

ARCHITECTURE test OF test_mod_mm_exponentiation IS
COMPONENT mod_mm_exponentiation IS
  PORT(
    x, y: IN STD_LOGIC_VECTOR(k-1 DOWNTO 0);
    clk, reset, start:IN STD_LOGIC;
    z: OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0);
    done: OUT STD_LOGIC
  );
END COMPONENT;

SIGNAL x, y, z: STD_LOGIC_VECTOR(k-1 DOWNTO 0);
SIGNAL reset, start, done: STD_LOGIC;
SIGNAL clk: STD_LOGIC := '0';

BEGIN

dut: mod_mm_exponentiation
PORT MAP(x => x, y => y, z => z, clk => clk, reset => reset, start =>
start, done => done);

clk <= NOT(clk) AFTER 1 nS;

test_vectors: PROCESS
BEGIN
    x <= "11111111";
    y <= (k-1 => '0', others => '1');
    reset <= '1';
    start <= '1';
    WAIT FOR 100 NS;
    reset <= '0';
    WAIT FOR 100 NS;
    start <= '0';
    WAIT FOR 100 NS;
    start <= '1';
    WAIT;

    WAIT FOR 8000000 NS;
    x <= (k-1 => '1', others => '0');
    start <= '0';
    WAIT FOR 100 NS;
    start <= '1';

    END PROCESS;
    END test;
```

3.2)for x=(1111 1111)



For x=(1000 0000):



For x =1111 1111 the total computation time was 700ns respectively which is 0.7 micro second
For x =1000 0000 total computation time was 300 ns which is 0.3 microsecond
Therefore we observe that even the value of the exponent is a factor that can affect computation time.

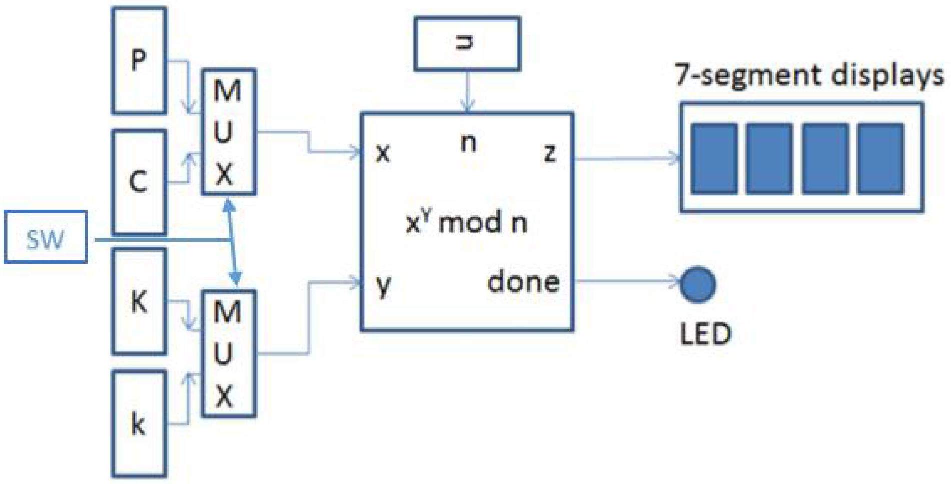
<u>4 Integration of the exponentiation in a RSA</u>

In order to display the outcome on the 7-segment display, we need to implement our method on FPGA after simulating the exponentiation module. The encryption of plaintext and decryption of ciphertext are the objectives.

$$\begin{tikzcd} P \arrow[r, "K"] & C \end{tikzcd}$$

$$\begin{tikzcd} C \arrow[r, "k"] & P \end{tikzcd}$$

To test whether the output of the exponentiation will be equal to the right value of the ciphertext during implementation, we have built a new entity called Imple FPGA that will receive the plaintext and the public key as inputs. The components (Multiplexer, modm exponentiation, and display) were then implemented in a single architecture. The following diagram shows the architecture:



We have added the vhdl modules for the seven segment display and LED of Basys3 board, we also added the constraint files to program FPGA. In order to test the proper functioning of the architecture, we used the values given in the lab:

- P (plaintext): 32768 in decimal, 0x8000 in HEX
- K (public key): 127 in decimal, 0x007F in HEX
- n (modulo): 63383 in Decimal, 0xF797 in HEX

We tried to generate beatstream but due to limited time we could not solve some errors in our top design module. As a result we were unable to successfully implement the beatstream which is the main key to program FPGA. Though we could not implement it in an

FPGA board, we clearly understood the complete process of the RSA encoding algorithm by using the simulation in testbench.

Conclusion:

After finishing this lab we learned how arithmetic operations for RSA encoding algorithms work and the importance of having an optimised code for machine time computation. We've seen how crucial it is to have keys that are hundreds of thousands of bits long in order to fully secure the transmission of data encrypted using this type of technique.