École nationale supérieure en systèmes avancés et réseaux

Report SE520

Subject: Implementation of RSA on FPGA

Minh Quang Nguyen

Bessem BEN AMOR

5A EIS & Mistre

Année universitaire 2023/2024

I. Introduction

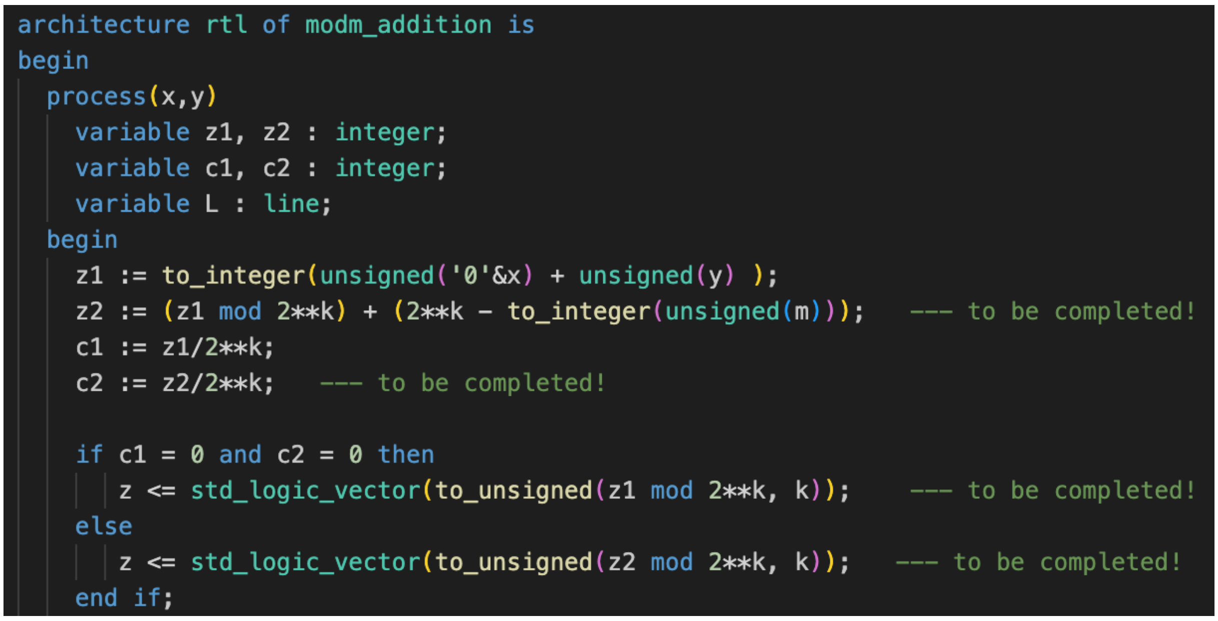
The goal of this lab is to implement the RSA encryption and decryption algorithm on a Basys-3 FPGA board. The work includes RTL and circuit designs of the arithmetic modules involved in the 16-bit RSA computations, representing encrypted and decrypted message with 7-segment display on FPGA, and analysis of brute-force attack on the private key used for the encryption process.

II. Lab code and result

1. Mod m adder

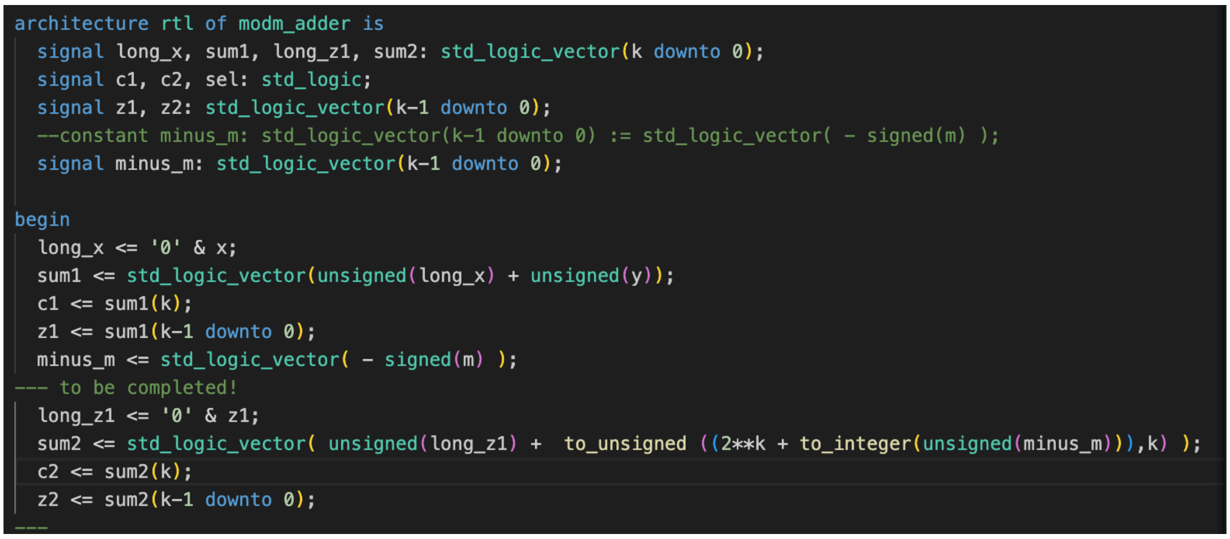
1.1. Behavioral description

modm_addition.vhd



```vhdl
architecture rtl of modm_addition is
begin
  process(x,y)
    variable z1, z2 : integer;
    variable c1, c2 : integer;
    variable L : line;
  begin
    z1 := to_integer(unsigned('0'&x) + unsigned(y) );
    z2 := (z1 mod 2**k) + (2**k - to_integer(unsigned(m)));   --- to be completed!
    c1 := z1/2**k;
    c2 := z2/2**k;   --- to be completed!

    if c1 = 0 and c2 = 0 then
       z <= std_logic_vector(to_unsigned(z1 mod 2**k, k));    --- to be completed!
    else
       z <= std_logic_vector(to_unsigned(z2 mod 2**k, k));   --- to be completed!
    end if;
```

modm_adder.vhd



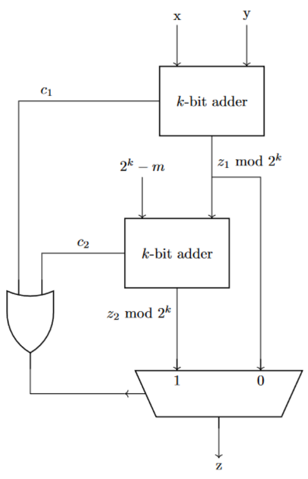
```vhdl
architecture rtl of modm_adder is
  signal long_x, sum1, long_z1, sum2: std_logic_vector(k downto 0);
  signal c1, c2, sel: std_logic;
  signal z1, z2: std_logic_vector(k-1 downto 0);
  --constant minus_m: std_logic_vector(k-1 downto 0) := std_logic_vector( - signed(m) );
  signal minus_m: std_logic_vector(k-1 downto 0);

begin
  long_x <= '0' & x;
  sum1 <= std_logic_vector(unsigned(long_x) + unsigned(y));
  c1 <= sum1(k);
  z1 <= sum1(k-1 downto 0);
  minus_m <= std_logic_vector( - signed(m) );
--- to be completed!
  long_z1 <= '0' & z1;
  sum2 <= std_logic_vector( unsigned(long_z1) +  to_unsigned ((2**k + to_integer(unsigned(minus_m))),k) );
  c2 <= sum2(k);
  z2 <= sum2(k-1 downto 0);
---
```

Datapath by completing

The algorithm performs addition between two integers, x and y, with modulo m, and k representing the number of bits.

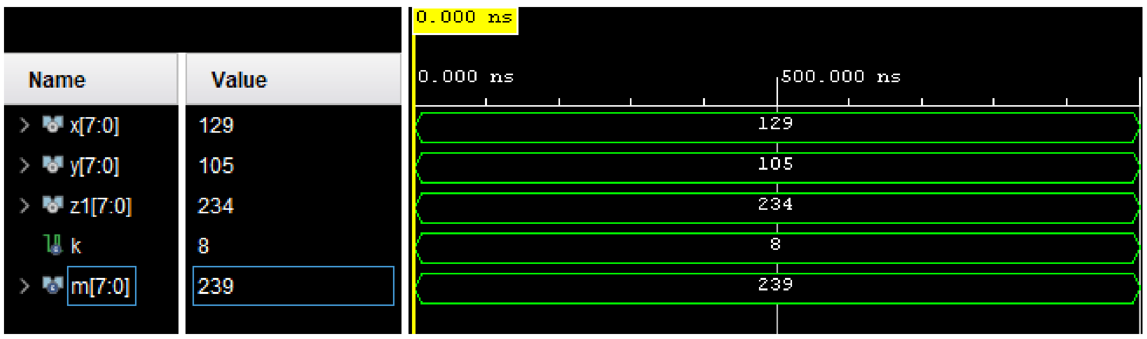
- z1 = x + y
- z2 = z1 mod $2^k + 2^k - m$

The result will be equal to : z2 mod 2^k if [(z1 ≥ m) or (z2 ≥ m)],
Otherwise, it will be z1 mod 2^k.



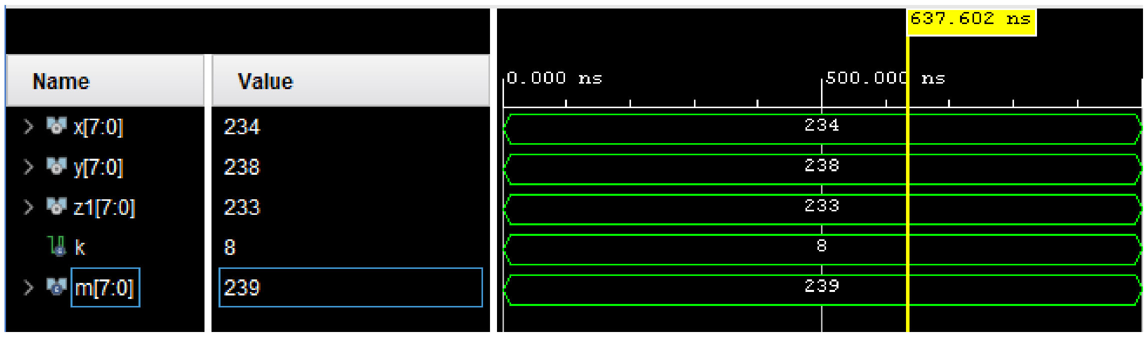
Example :

In this simulation, when we input x = 129 and y = 105 with m = 239, with k=8.
we obtain the following result:



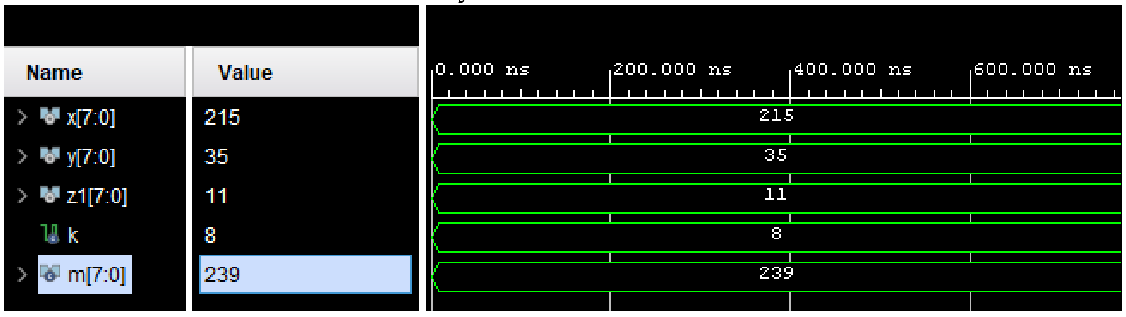
Name	Value
x[7:0]	129
y[7:0]	105
z1[7:0]	234
k	8
m[7:0]	239

Another example with : x= 234, y = 238



Name	Value
x[7:0]	234
y[7:0]	238
z1[7:0]	233
k	8
m[7:0]	239

Also for modm_adder with x=215, y =35



Name	Value
x[7:0]	215
y[7:0]	35
z1[7:0]	11
k	8
m[7:0]	239

2. Modulo m multiplier

2.1. Finite State Machine :

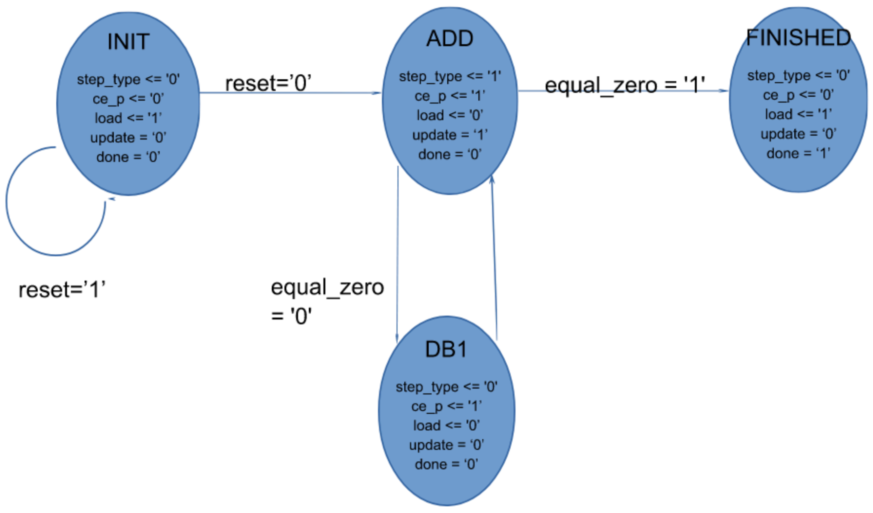
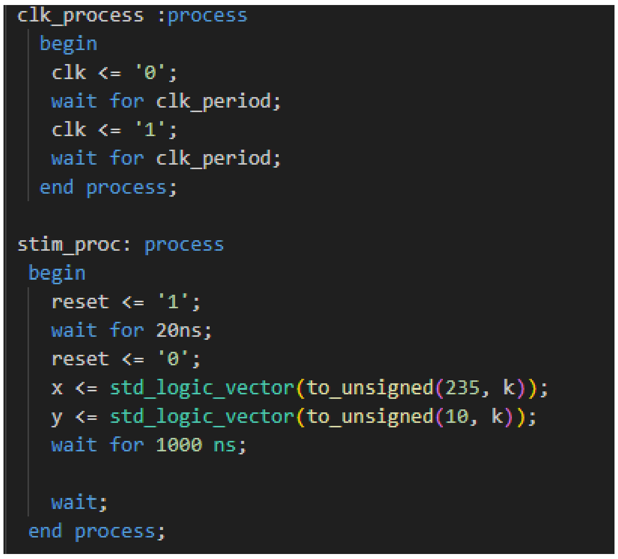


Figure x: FSM

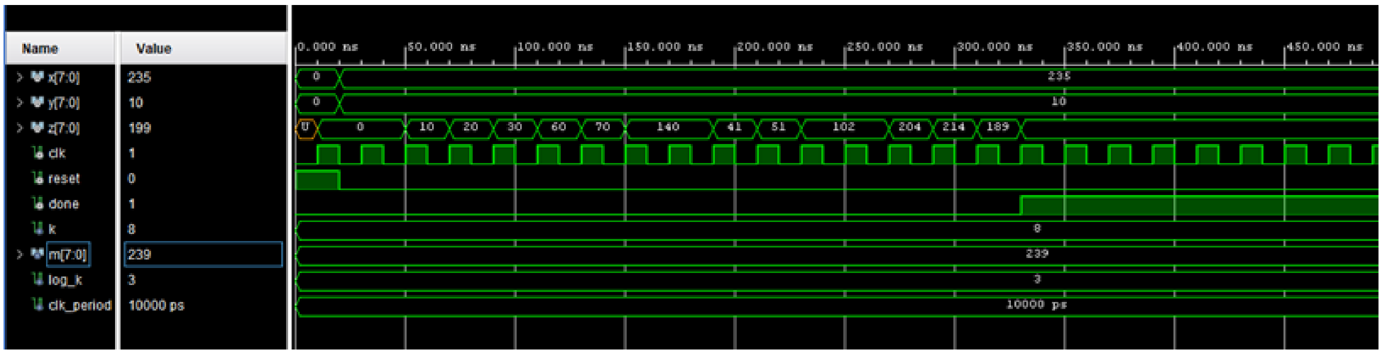
2.2. testbench modm_multiplier.vhd

tb_modm_multiplier.vhd



```
clk_process :process
  begin
   clk <= '0';
   wait for clk_period;
   clk <= '1';
   wait for clk_period;
  end process;

stim_proc: process
 begin
   reset <= '1';
   wait for 20ns;
   reset <= '0';
   x <= std_logic_vector(to_unsigned(235, k));
   y <= std_logic_vector(to_unsigned(10, k));
   wait for 1000 ns;

   wait;
 end process;
```

In this simulation of the testbench with inputs : x=235, y=10 and k=8.



Name	Value
x[7:0]	235
y[7:0]	10
z[7:0]	199
clk	1
reset	0
done	1
k	8
m[7:0]	239
log_k	3
clk_period	10000 ps

2.3. Computation time

As showing the simulation in 2.3

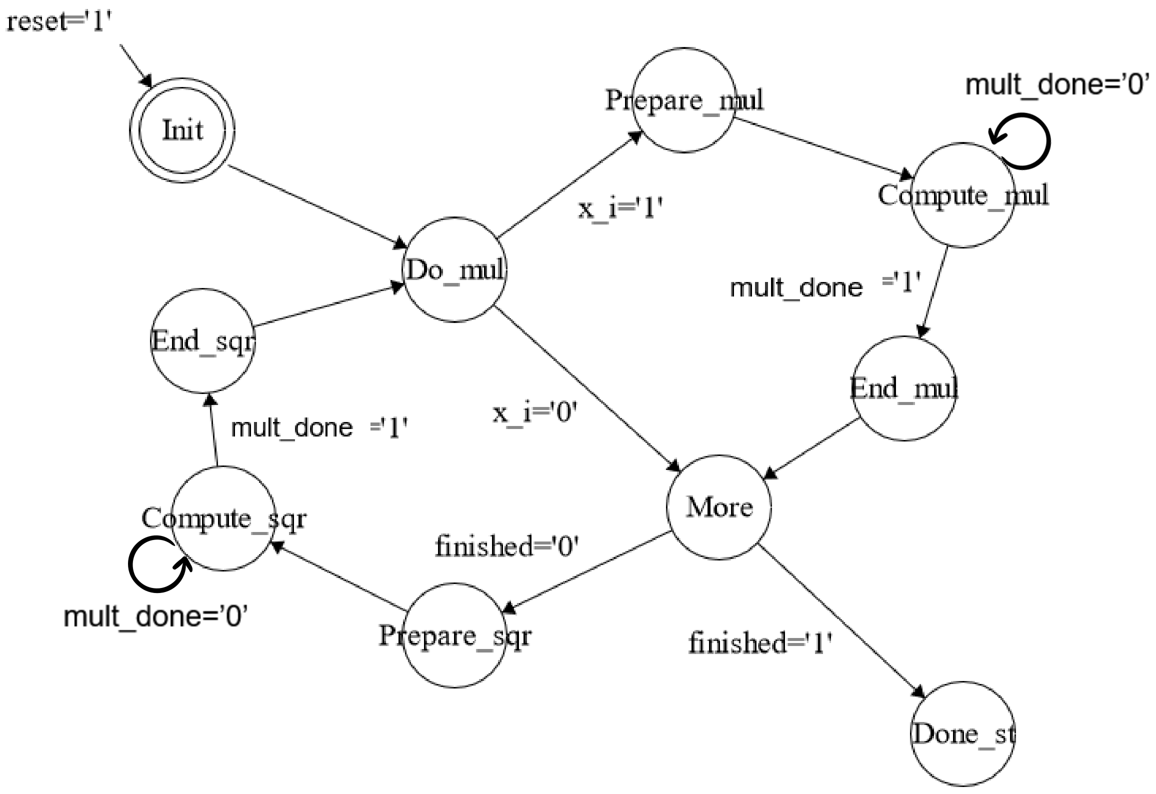
Computation time analysis = 2k-1

Comment: because 2*(k-1) cycles for ADD => DB1, then 1 cycle for the last add

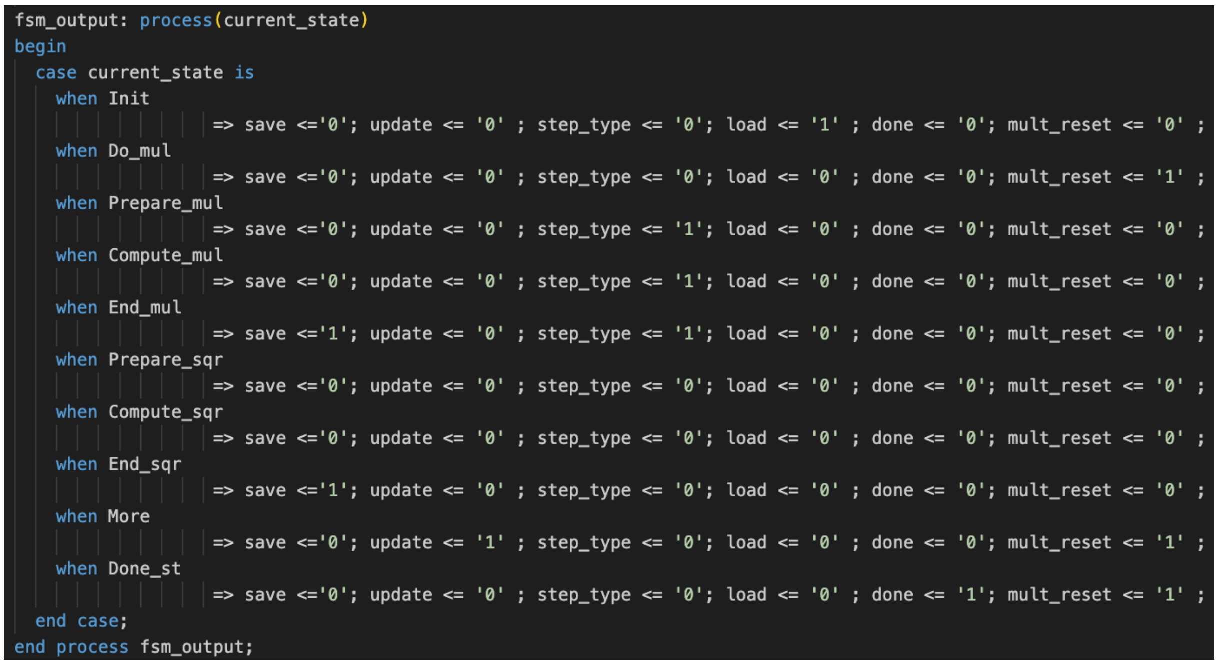
Check : k = 8, time = 15 cycles => correct

3. Modulo m exponentiation

3.1. FSM for the modulo m exponentiation

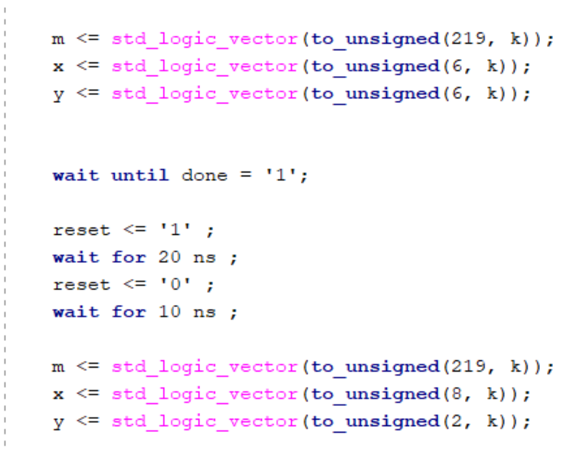


3.2. output values of the FSM

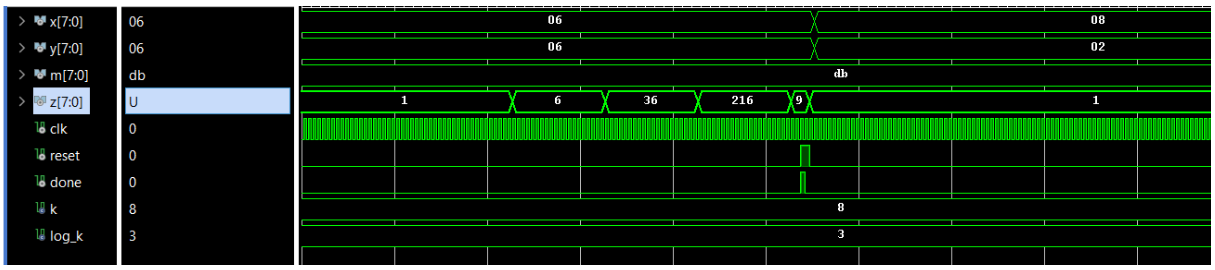


```
fsm_output: process(current_state)
begin
  case current_state is
    when Init
              => save <='0'; update <= '0' ; step_type <= '0'; load <= '1' ; done <= '0'; mult_reset <= '0' ;
    when Do_mul
              => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '1' ;
    when Prepare_mul
              => save <='0'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Compute_mul
              => save <='0'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when End_mul
              => save <='1'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Prepare_sqr
              => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Compute_sqr
              => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when End_sqr
              => save <='1'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when More
              => save <='0'; update <= '1' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '1' ;
    when Done_st
              => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '1'; mult_reset <= '1' ;
  end case;
end process fsm_output;
```

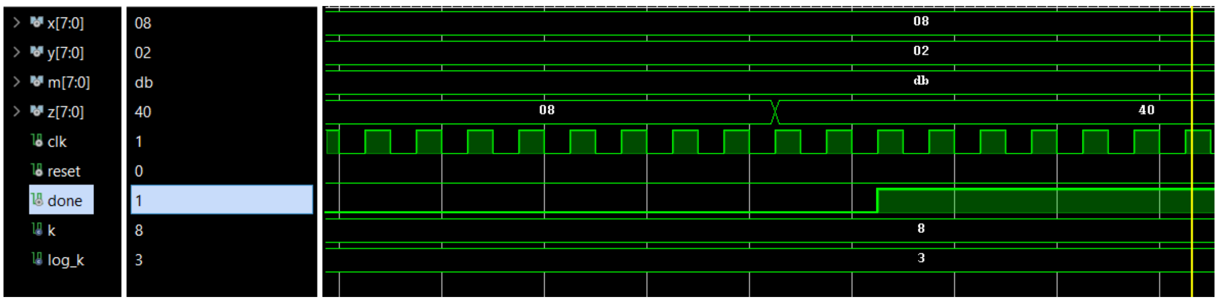
3.3. Design code, testbench and simulation



```
    m <= std_logic_vector(to_unsigned(219, k));
    x <= std_logic_vector(to_unsigned(6, k));
    y <= std_logic_vector(to_unsigned(6, k));


    wait until done = '1';

    reset <= '1' ;
    wait for 20 ns ;
    reset <= '0' ;
    wait for 10 ns ;

    m <= std_logic_vector(to_unsigned(219, k));
    x <= std_logic_vector(to_unsigned(8, k));
    y <= std_logic_vector(to_unsigned(2, k));
```

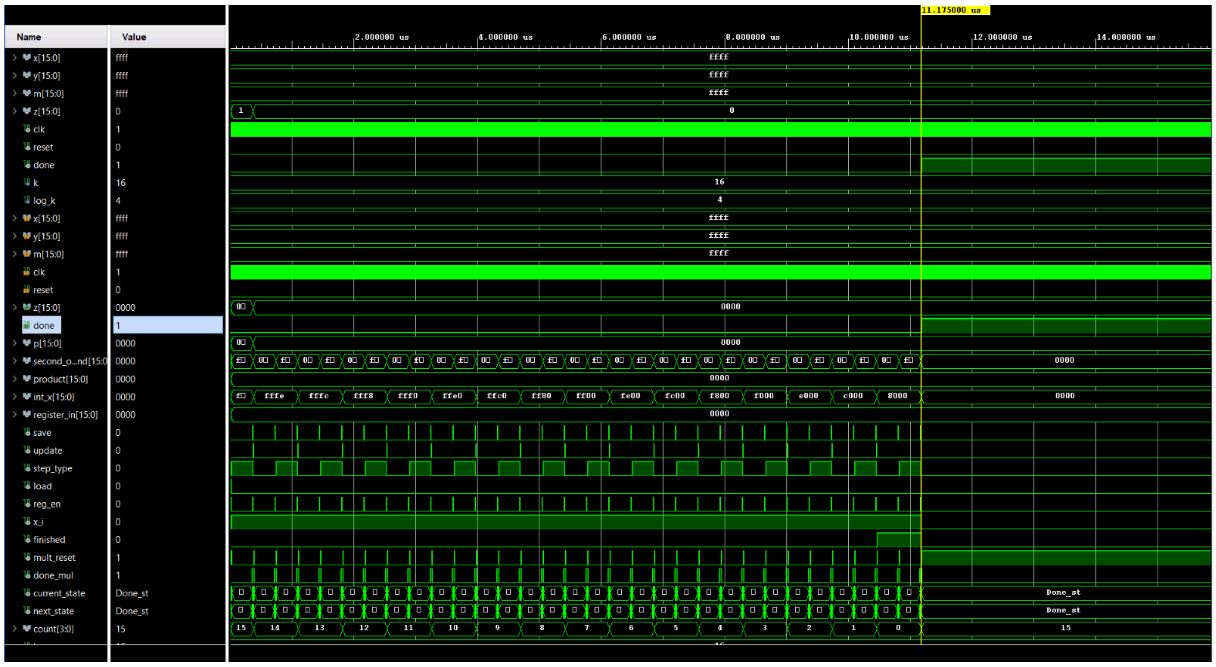


```
x[7:0]    06
y[7:0]    06
m[7:0]    db
z[7:0]    U
clk       0
reset     0
done      0
k         8
log_k     3
```



```
x[7:0]    08
y[7:0]    02
m[7:0]    db
z[7:0]    40
clk       1
reset     0
done      1
k         8
log_k     3
```

```
m <= std_logic_vector(to_unsigned(65535, k));
x <= std_logic_vector(to_unsigned(65535, k));
y <= std_logic_vector(to_unsigned(65535, k));
```

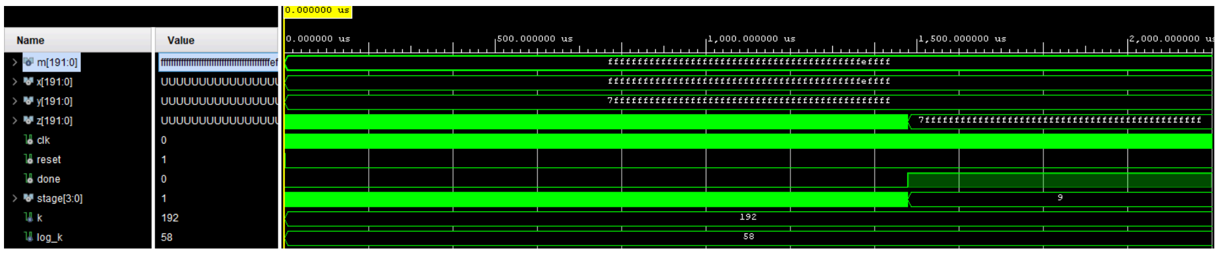


Name	Value
x[15:0]	ffff
y[15:0]	ffff
m[15:0]	ffff
z[15:0]	0
clk	1
reset	0
done	1
k	16
log_k	4
x[15:0]	ffff
y[15:0]	ffff
m[15:0]	ffff
clk	1
reset	0
z[15:0]	0000
done	1
p[15:0]	0000
second_o...nd[15:0]	0000
product[15:0]	0000
int_x[15:0]	0000
register_in[15:0]	0000
save	0
update	0
step_type	0
load	0
reg_en	0
x_i	0
finished	0
mult_reset	1
done_mul	1
current_state	Done_st
next_state	Done_st
count[3:0]	15

3.4. Big number

```
constant k: integer := 192;
constant log_k: integer := 58;

constant m : std_logic_vector(k-1 downto 0) := (191 downto 17 => '1', 15 downto 0 => '1', others => '0');

signal x, y, z: std_logic_vector(k-1 downto 0);
signal clk, reset, done: std_logic := '0';
signal stage: std_logic_vector(3 downto 0);

-- Stimulus process
stim_proc: process
begin

reset <= '1';
wait for 10 ns ;
reset <= '0';
--(2**192)-1-(2**16)
x <= m;
y <= (190 downto 0 => '1', others => '0');
wait until done = '1';
wait;
end process stim_proc;
```

From the waveform, we can see z = y as desired

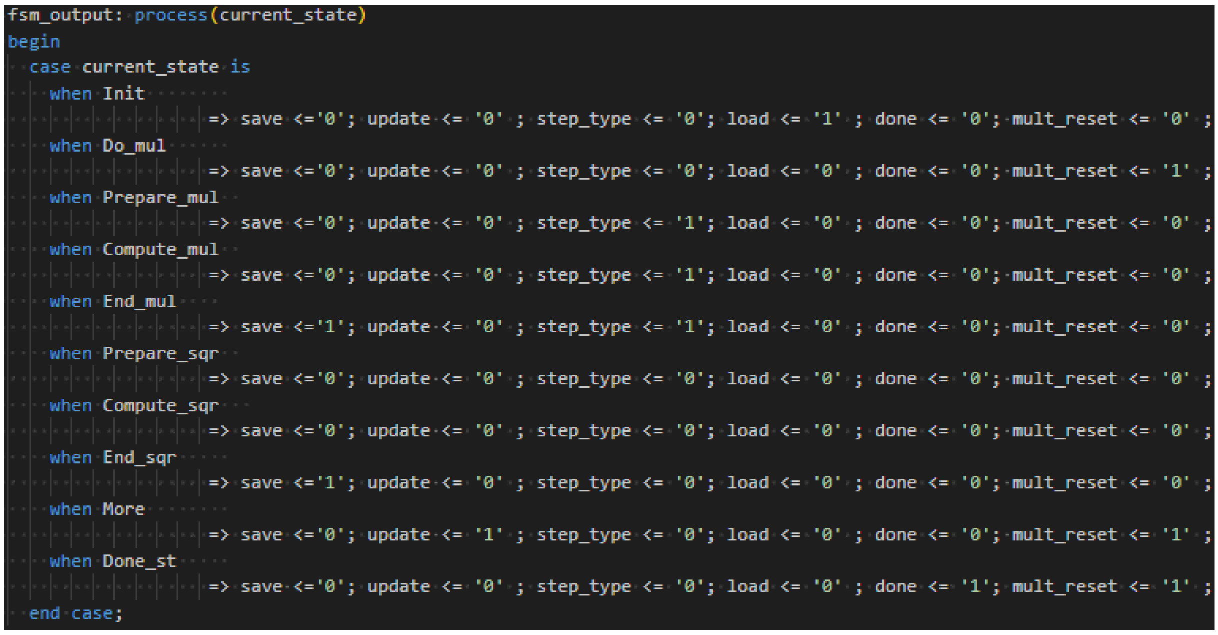

0.000000 us
Name
Value
m[191:0]
x[191:0]
y[191:0]
z[191:0]
clk
reset
done
stage[3:0]
k
log_k
0.000000 us
500.000000 us
1,000.000000 us
1,500.000000 us
2,000.000000 us
ffffffffffffffffffffffffffffffffffffffffffffeffff
ffffffffffffffffffffffffffffffffffffffffffffeffff
7fffffffffffffffffffffffffffffffffffffffffffffff
7fffffffffffffffffffffffffffffffffffffffffffffff
9
192
58

3.5. Compute worst-case execution time

* Computational time is only dependent on k and x, while m and y do not pose an impact.

* We start the exponential computation at 15ns. Clock cycle is 10ns.

* Improved code, so each mul only takes 2k cycles, 2k-1 for the multiplier module to output done, and 1 cycle lag for the exponential function to take the done input and move on to the next state.



```
fsm_output: process(current_state)
begin
  case current_state is
    when Init
                => save <='0'; update <= '0' ; step_type <= '0'; load <= '1' ; done <= '0'; mult_reset <= '0' ;
    when Do_mul
                => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '1' ;
    when Prepare_mul
                => save <='0'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Compute_mul
                => save <='0'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when End_mul
                => save <='1'; update <= '0' ; step_type <= '1'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Prepare_sqr
                => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when Compute_sqr
                => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when End_sqr
                => save <='1'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '0' ;
    when More
                => save <='0'; update <= '1' ; step_type <= '0'; load <= '0' ; done <= '0'; mult_reset <= '1' ;
    when Done_st
                => save <='0'; update <= '0' ; step_type <= '0'; load <= '0' ; done <= '1'; mult_reset <= '1' ;
  end case;
```

* Since x = 11…11, mul is always executed for each bit position, thus 2k cycles for the computation, and 2 for prepare and compute stage. Total mul = k*(2k+2)

* Since square is executed (k-1) times, 2k cycles for the computation, and 2 for prepare and compute stage. Total square = (k-1)*(2k+2)

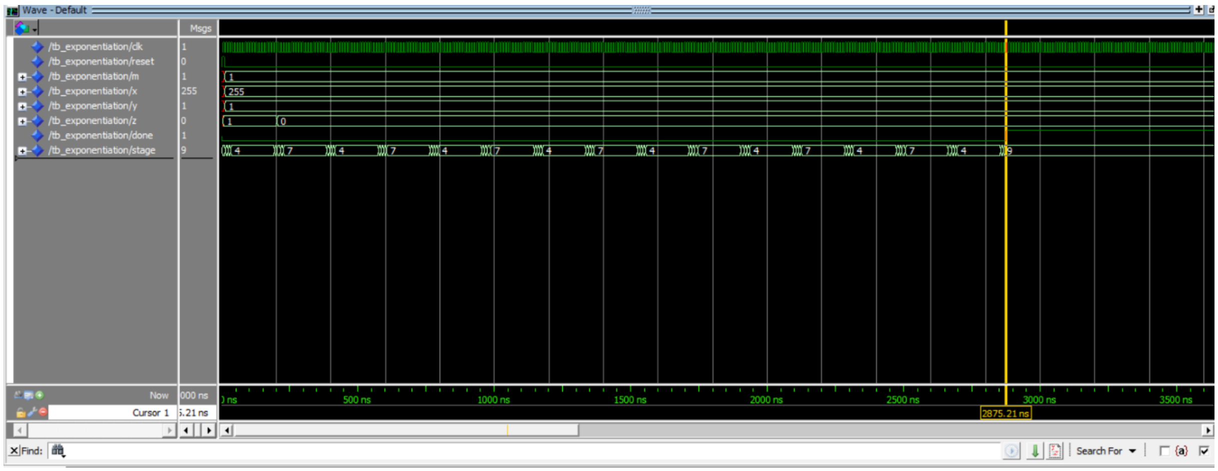
* Transition between mul and square, take (k-1) + (k) – 1 = 2k-2 cycles

* Ending takes 2 cycles

* Thus total = (2k-1)(2k+2) + 2k = 4k^2 + 4k – 2

Verify with examples:

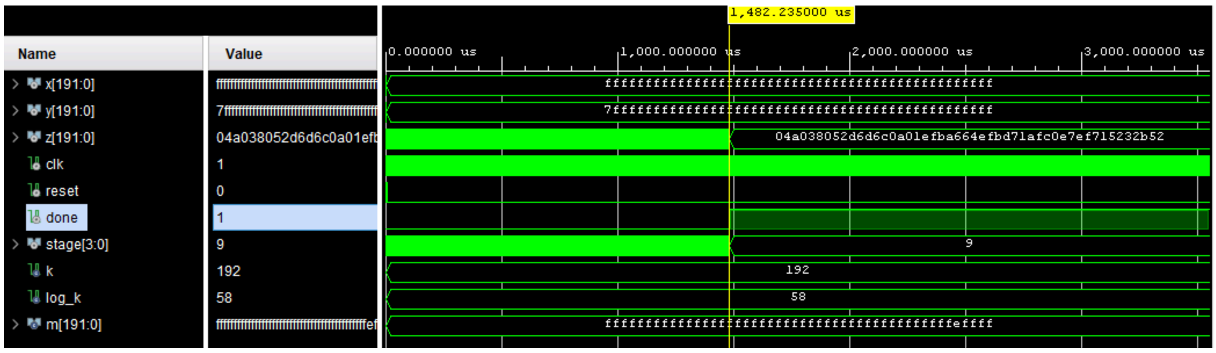
a) x = 255, k = 8. Cycles = 4k^2 + 4k – 2 = 286 = (2875ns-15ns): 10ns



```
Wave - Default
                              Msgs
/tb_exponentiation/clk        1
/tb_exponentiation/reset      0
/tb_exponentiation/m          1
/tb_exponentiation/x          255
/tb_exponentiation/y          1
/tb_exponentiation/z          0
/tb_exponentiation/done       1
/tb_exponentiation/stage      9
```

b) x = 0xFFF…FF , k = 192. Cycles = 4k^2 + 4k – 2 = 148222

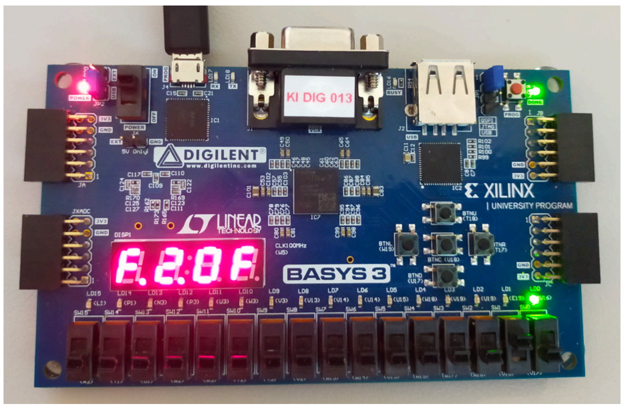
From waveform = 1482220:10= 148222 -> correct

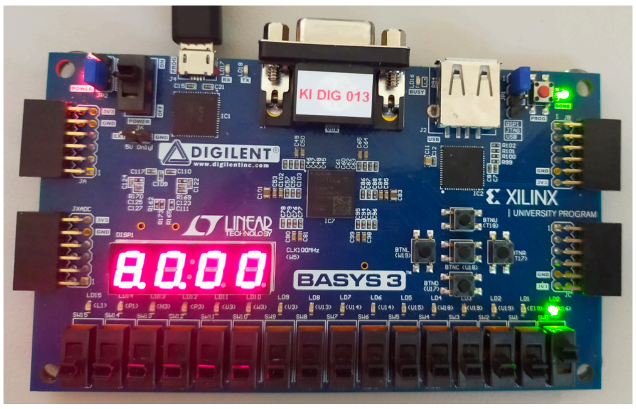


```
Name          Value
x[191:0]      ffffffffffffffffffffffffffffffffffffffffffffffff
y[191:0]      7fffffffffffffffffffffffffffffffffffffffffffffff
z[191:0]      04a038052d6d6c0a01efba664efbd71afc0e7ef715232b52
clk           1
reset         0
done          1
stage[3:0]    9
k             192
log_k         58
m[191:0]      fffffffffffffffffffffffffffffffffffffffffffeffff
```

4. RSA in the Basys 3

4.1. RSA encryption and decryption

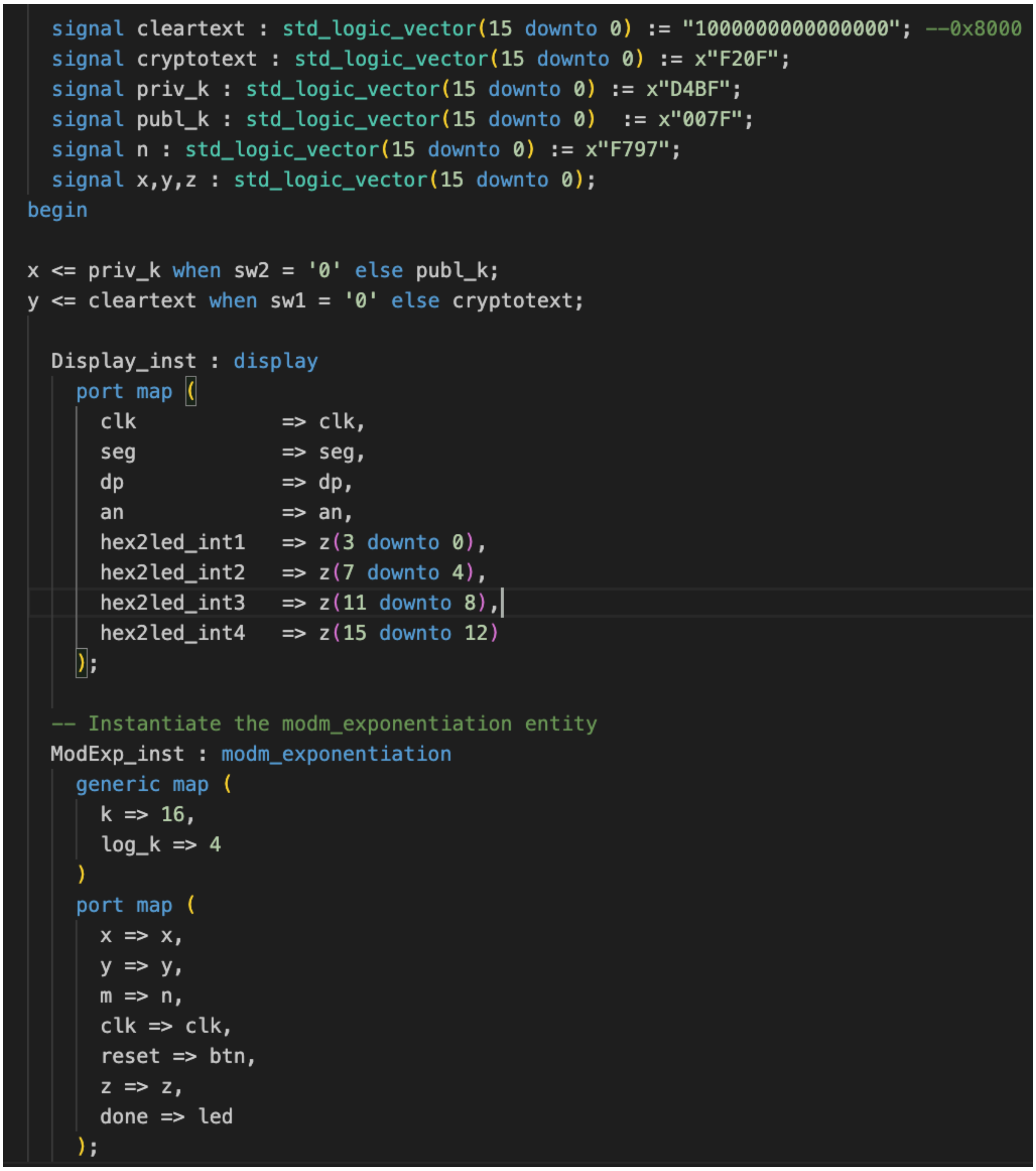




In the constraints file we have declared sw1 to be V17, and sw2 to be V16 of Basys 3

When sw1=0, sw2=1 => encrypting the cleartext to ciphertext F20F using public key

When sw1=0, sw2=1 => decrypting the ciphertext to cleartext 8000 using private key



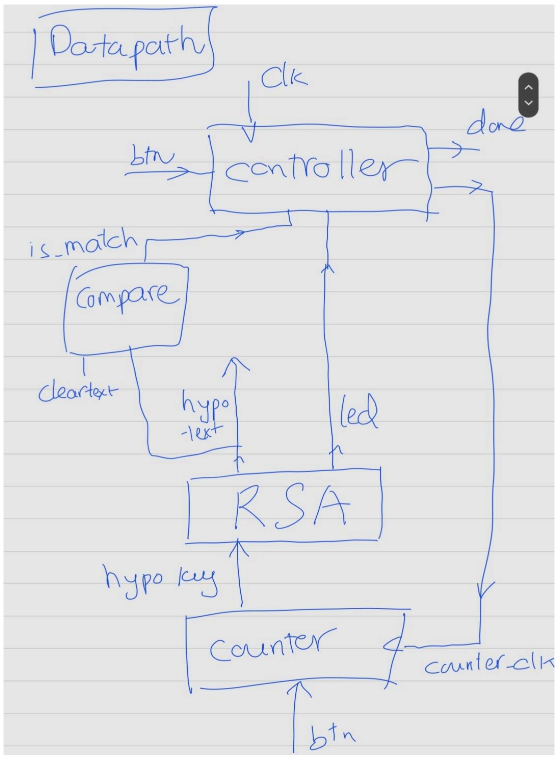
```vhdl
  signal cleartext : std_logic_vector(15 downto 0) := "1000000000000000"; --0x8000
  signal cryptotext : std_logic_vector(15 downto 0) := x"F20F";
  signal priv_k : std_logic_vector(15 downto 0) := x"D4BF";
  signal publ_k : std_logic_vector(15 downto 0)  := x"007F";
  signal n : std_logic_vector(15 downto 0) := x"F797";
  signal x,y,z : std_logic_vector(15 downto 0);
begin

x <= priv_k when sw2 = '0' else publ_k;
y <= cleartext when sw1 = '0' else cryptotext;

  Display_inst : display
    port map (
      clk             => clk,
      seg             => seg,
      dp              => dp,
      an              => an,
      hex2led_int1    => z(3 downto 0),
      hex2led_int2    => z(7 downto 4),
      hex2led_int3    => z(11 downto 8),
      hex2led_int4    => z(15 downto 12)
    );

  -- Instantiate the modm_exponentiation entity
  ModExp_inst : modm_exponentiation
    generic map (
      k => 16,
      log_k => 4
    )
    port map (
      x => x,
      y => y,
      m => n,
      clk => clk,
      reset => btn,
      z => z,
      done => led
    );
```

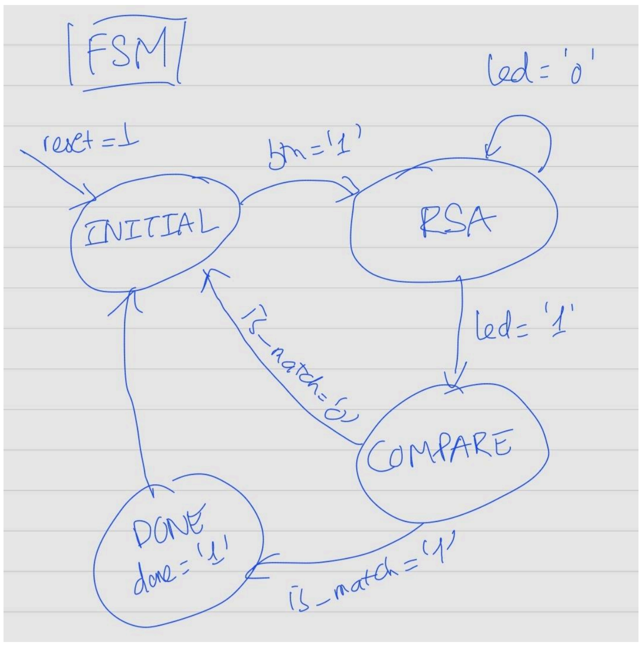
4.2. Brute-force attack

We have made the datapath and FSM diagram of the scheme. Unfortunately, we have not realized this logic on board.

The datapath consists of the RSA module from 4.1, a comparator to compare the cleartext with decrypted message produced by the bruteforce attack from the RSA. If there is a match, we alert the controller to output this value.

From the FSM, we can see our design could continue to output different "matching" private keys as button btn are being pressed. Only if there is a match that the output is displayed.





III. Conclusion

Through the lab, we have gained hands-on experience in implementing RSA encryption and decryption logic on FPGA. We have also gained insights in designing VHDL modules and testing them throughout. Lastly, we learned about the difficulty and feasibility of brute-force attack on this RSA implementation.