Hardware Implementation of Modular Arithmetic for the RSA algorithm

Yann Kieffer – based on previous work by Vincent Beroulle and Fabien Sagnel.

Preliminaries

The goal of this lab is to work on arithmetic primitives involved in RSA computations, as VHDL code representing circuit designs, which will be simulated first, and then implemented on an FPGA.

The implementation will be done on a Basys-3 FPGA board, carrying a Xilinx Artix-7 FPGA.

This lab is spanning 3 lab sessions.

Introduction

The final objective of this lab is the implementation in an FPGA of the RSA encryption and decryption algorithms.

In order to get there, you will work on:

- a modulo m adder;
- a modulo m multiplier;
- a modulo m exponentiation.

A lot of VHDL code is provided to you; whenever code is provided, your first job is to get to know this code, understand how it works and why it works; and validate its correctness with appropriate testbenches. The first testbench is provided, the others you will write yourself.

Evaluation/assessment/deliverables

Even though most of the code is given to you for the first 3 parts, there is still a lot to do in this lab. The only way to get to the end is to work efficiently during the session, and also work between the sessions.

To help you stay on schedule, I will come and see you at the start of sessions 2 and 3, and ask you to show me:

- session 2: your testbench from question 1.3, and the FSM diagram from question 2.1;
- session 3: the output function for the exponentiation in question 3.2.

Please note that these checkpoints are no guarantee that you can finish the project by the end of session 3. It is highly recommended to advance further than the checkpoints in order to take full advantage of the next lab session.

Your project is complete when you have finished question 4.1. Please note that you cannot bring a Basys 3 at home, so you will need to work at school to finish part 4. Part 4.2 is optional, and present here for your personal interest only; please work on it if you have

finished 4.1 before the end of session 3. Presenting the results of part 4.2 in your report may or may not bring you additional points.

The assessment of your work will be summarized in a grade out of 20 points. Here is the detail of the points:

- 4 points for each checkpoint above, for a total of 8 points. You may still reap points if you present work for the previous checkpoint, but you cannot get more than 2 points in this case;
- 4 points are devoted to your attitude and commitment, both during the session, and during the checkpoint interviews. This is the only aspect that is assessed on an individual basis;
- 8 points are devoted to the final report you will post on Chamilo at most 5 working days after the last lab session. The quality of your code organization and presentation, the quality of your reporting and that of your reasoning will all contribute to this part of the grading. A project containing the answers to all questions up to and including question 4.1 is considered complete.

The final deliverables are:

- A unique final report including all your results (with some comments);
- An archive with all your source code (VHDL (source) files, XDC (constraint) files, and project definition files).

They are both to be posted on Chamilo, at most 5 working days after the last lab session.

Additional resources

You may find useful some resources provided in the “Links” section of the SE520 on Chamilo. You will find there:

- Some VHDL tutorials

- A pointer to a page giving a clear view of arithmetic operators in VHDL.

1. Modulo m adder

In this first section, you will write a VHDL description of a modular adder.

We want to compute $\bar{z} = \bar{x} + \bar{y}$ in Z / mZ. The classes $\bar{x}$, $\bar{y}$ and $\bar{z}$ will be represented by numbers taken in {0, …, m-1}. We choose k as the smallest number of bits needed to represent the values m, x, y and z.

Binary mod m addition algorithm

```
z1 := x + y;
z2 := (z1 mod 2**k) + (2**k - m);
c1 := z1/2**k; c2 := z2/2**k;
if c1 = 0 and c2 = 0
  then z := (z1 mod 2**k);
  else z := (z2 mod 2**k);
end if;
```

Examples:

Assume that k = 8 (2^8 = 256) and m =239; then 2^k – m = 236 – 239 = 17.

For x = 129 and y = 105:

- z1 = 129 + 105 = 234
- z2 = 234 + 17 = 251
- c1 = 0, c2 = 0
- z = z1 mod 256 = 234

For x = 234 and y = 238:

- z1 = 234 + 238 = 472
- z2 = 216 + 17 = 233
- c1 = 1, c2 = 0
- z = z2 mod 256 = 233

For x = 215 and y = 35:

- z1 = 215 + 35 = 250
- z2 = 250 + 17 = 267
- c1 = 0, c2 = 1
- z = z2 mod 256 = 11

1.1. Complete the behavioral description of this algorithm in the file *modm_addition.vhd*; verify it with a testbench (*tb_modm_adder.vhd*) using the previous given computation examples.

The next figure represents the architecture of the datapath implementing the previous algorithm.

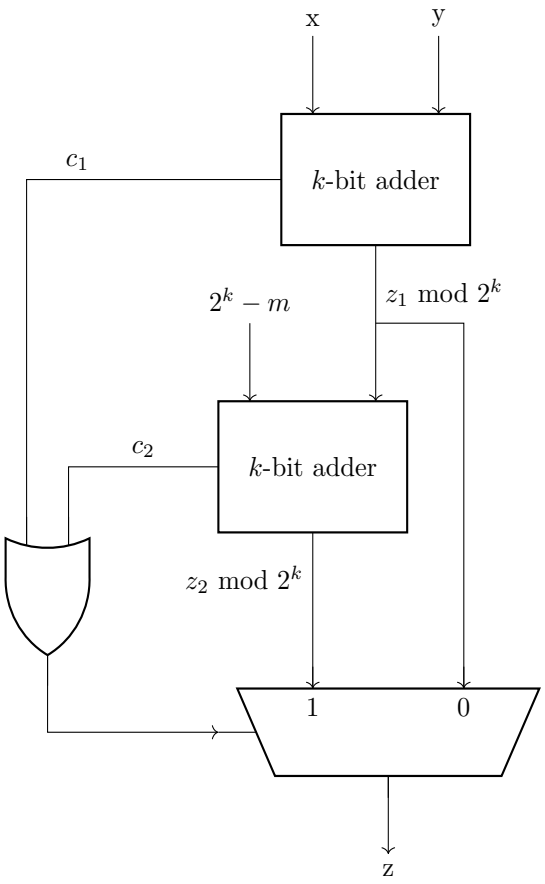


Figure 1. Datapath corresponding to the binary modular addition

1.2. Implement this datapath by completing the given VHDL RTL description in the file *modm_adder.vhd*

1.3. Reuse the initial behavioral description of the mod m addition to verify this new RTL implementation of the mod m addition with a testbench comparing the evaluations of the design against those of the behavorial component from modem_addition.vhd (reuse and complete tb_modm_adder.vhd).

2. Modulo m multiplier

We will now leverage the modulo m adder to create a modulo m multiplier.

There are many ways to create a modulo m multiplier from a modulo m adder. We will use the following computation algorithm.

MSB modulo m multiplication:

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

MSB means “most significant bit”, it means that we examine one of the operands (x in our case) starting from its most significant bit. Notice how the computations depend on the values of x(i) for decreasing values of i.

This algorithm may be implemented in different ways. We choose to use only one adder for both addition steps in our implementation. Here is a schematic of the multiplier datapath.

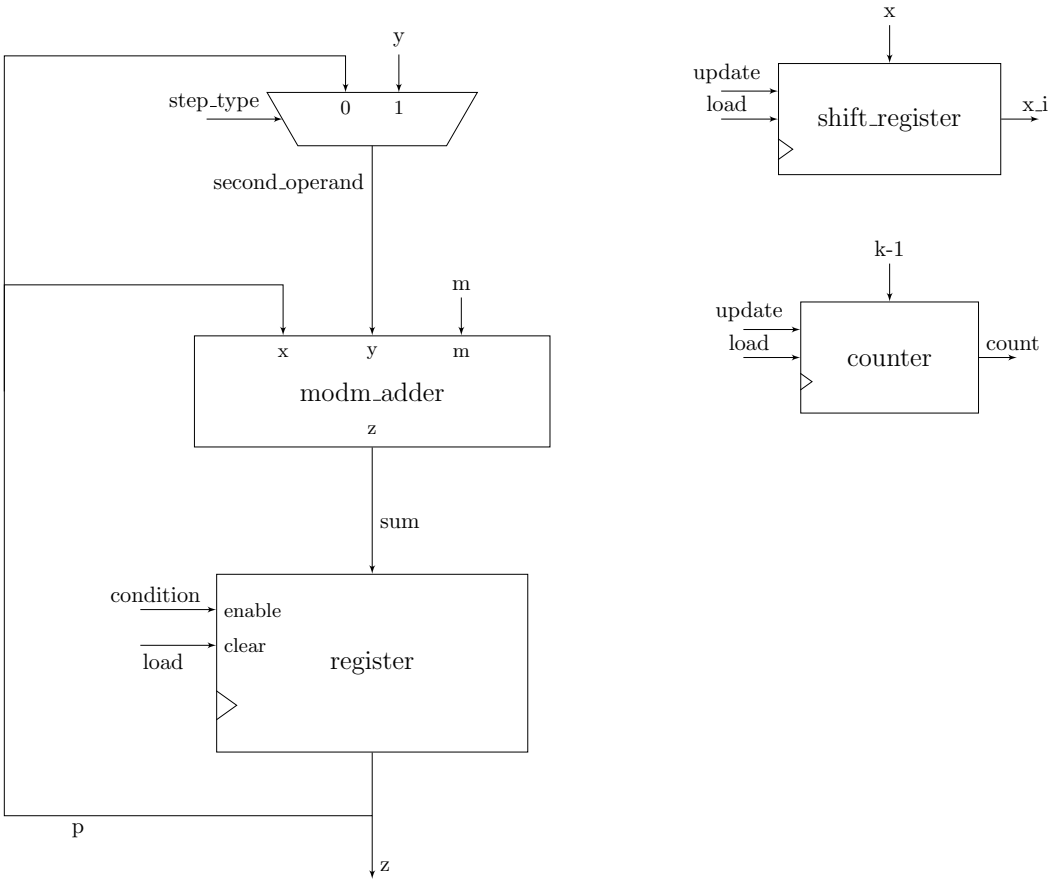


Figure 2. Datapath corresponding to MSB modular multiplication

The full implementation of the adder is provided in the file *modm_multiplier.vhd*.

2.1. Draw the Finite State Machine transition diagram corresponding to the control part of the circuit, as extracted from the design provided in modm_multiplier.vhd. Do not forget to indicate the output function!

2.2. Write a testbench and verify that the RTL description given in modm_multiplier.vhd is actually computing the mod m multiplication.

2.3. Analyze the computation time (counted in clock cycles) as a function of k, the size of x and y. Verify in your testbench that your count is correct.

3. Modulo m exponentiation

Exponentiation modulo m is the only computation needed to encrypt and decrypt messages with RSA, once the private/public key pair is set up.

In this part, you will complete a partial RTL description of an implementation of the MSB exponentiation algorithm.

The MSB exponentiation algorithm was already presented in the *sagemath* lab. Here is another, equivalent, presentation:

MSB modulo m exponentiation:

```
e := 1;
if x(k-1) = 1
  then e := y;
end if;
for i in k-2 .. 0 loop
  e := (e*e) mod m ;
  if x(k-i) = 1 then
    e := (e*y) mod m;
  end if;
end loop;
z := e;
```

Figure 3 presents the datapath of the implementation of the modulo m exponentiation which is partially coded in the file *modm_exponentiation.vhd*.

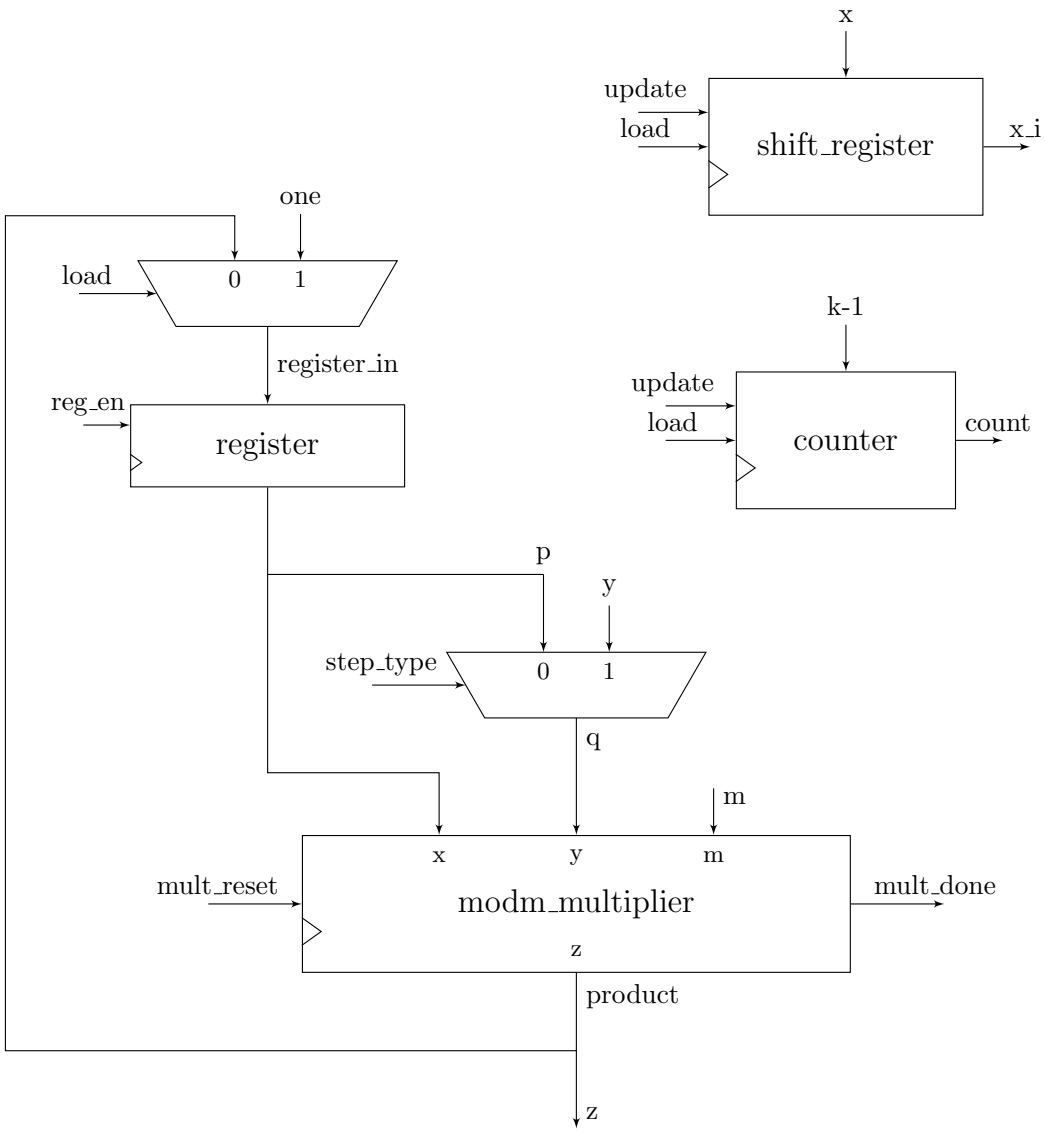


Figure 3. Datapath of a circuit that computes $z=y^x \bmod m$

3.1. Extract from the code provided in the file *modm_exponentiation.vhd* the states and transition of the FSM for the modulo m exponentiation.

3.2. By analyzing the algorithm, the datapath and the FSM, decide the output values of the FSM in each state for each of the output signals: shift, load, save, step_type, mult_reset, done.

3.3. Complete the design code with your output function; create a testbench, and test your implementation.

3.4. Try a final validation with the following data:

```
m = (2^192-1)-2^16; (this is a prime number!)
x = m;
y = 0x7FFF.....FF.
```

According to Fermat's little theorem, $y^m = y \pmod{m}$ whenever m is a prime number. You can deduce Fermat's theorem from Euler's theorem.

3.5. Use a value of 0xFFF...FF for x, in order to investigate the worst-case execution time. Compute the worst-case execution time of the modulo m exponentiation design, as a function of k; and validate that you measure the same value with the help of your testbench.

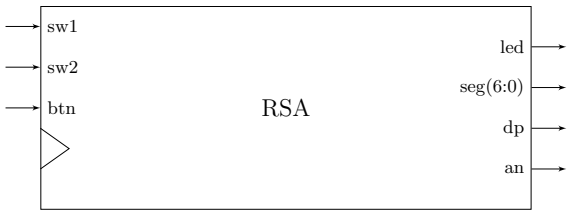
4. RSA in the Basys-3

The goal of this section is to use the modulo m exponentiation block to perform some RSA encryption and decryption. The implementation will be done in a Basys-3 FPGA board.

The following 16-bit values will be used to perform the encryption/decryption computations.

```
P (cleartext):  32768
C (cryptotext): 61967
k (private key): 54463
K (public key): 127
n (modulo): 63383
```

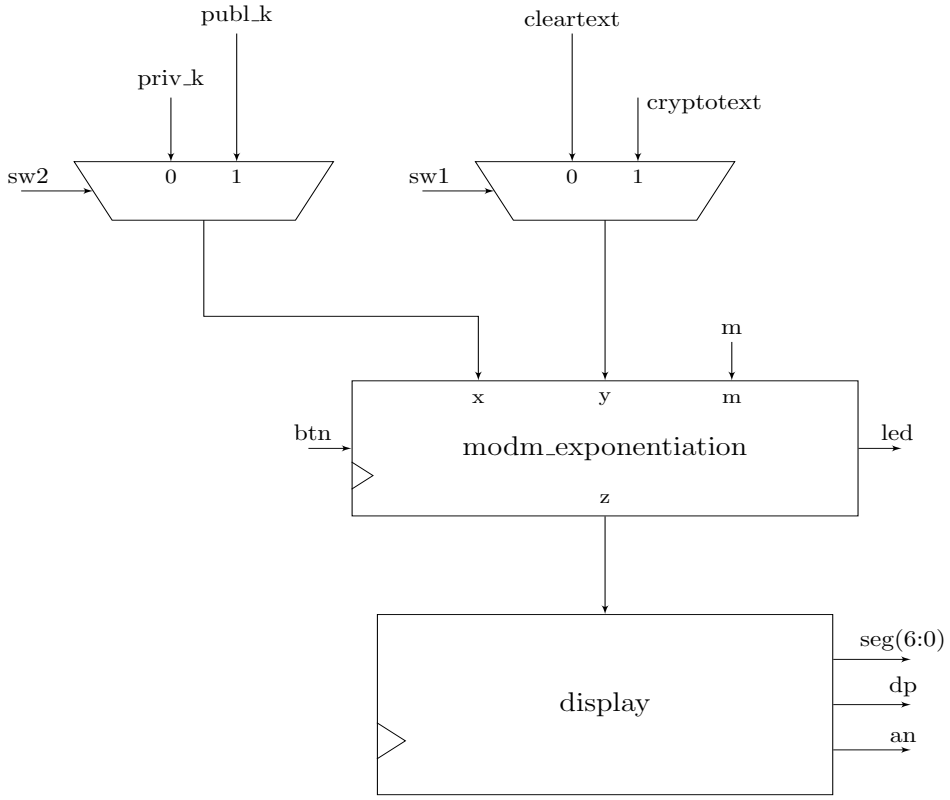
4.1. Create a block RSA that has the following input and output signals:



It uses 2 switches and one button from the basys3. Its output goes to one led, and the 3 other outgoing signals are meant to drive the 4 character led segment display of the basys3.

This block's function is to compute either an RSA encryption or decryption, according to the 2nd switch; and it is acting either on the internal cleartext, or the internal cryptotext, according to the 1st switch. The button acts as a reset button: when depressed, the circuit is restarted. The led lights when the computation is finished, and the led display shows the result with 4 hexadecimal characters.

You may use and take inspiration from the 7-segment display code found in Chamilo. Here is the datapath for this block:



Verify your design with a testbench. When you are confident that it is acting correctly, synthesize it, flash it into the basys3, and verify that the given public key encrypts the cleartext to the ciphertext, and that the given private key decrypts the ciphertext to the cleartext.

4.2. (Optional) In this final part, you will design, first on paper, and then in VHDL, a block that carries out a brute-force attack on RSA. Of course, the key we're looking for is a private key. So the design should use a 16-bit cleartext message, the matching 16-bit ciphertext, and of course the value of the modulus m. All these values will be declared as constants in your code. It is going to look for the encryption key by trying all possible encryption exponents, and stopping at those that match. Of course, the exponent that is found is to be presented on the 7-segment display.

Here are the numeric data:

- m = 64507
- Plaintext = 1000
- Ciphertext = 12486

You may do a first version that stops at the first match. Then you may try your hand at a version that stops at each match, but lets you relaunch the attack to gather all the possible keys. For this last version, it is recommended that you first create a circuit that counts the presses on one of the buttons of the Basys3. You will probably find that debouncing the button signal is necessary to have a usable circuit.

How much time does it take to carry out a brute-force attack on 16-bit RSA using a Basys3? How many private keys encode the Plaintext to the Ciphertext?