Cryptographic Engineering
Digital design tutorial #1 : ECC project

Exercise from Elliptic Curve Cryptography hardware tutorial in Croatia Summer School given by

Associate Professor Dr. Nele Mentens from KU Leuven.

Léo Weissbart

March 18, 2020

TUTORIAL INSTRUCTIONS

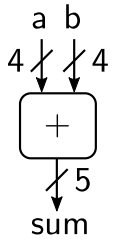
In this tutorial, we will design and simulate a hardware architecture for elliptic curve cryptography. First, we will gradually build the datapath. Next, we will add a memory unit and control logic to implement an elliptic curve point scalar multiplication. We simulate the behavior of each new design by following Step 3 of the pre-tutorial instructions. Note that we only do behavioral simulation in this tutorial; for actual hardware implementation, we would need tools for synthesis and physical implementation (as explained in the introductory tutorial presentation). Download the VHDL design files from Brightspace.
For the project submission, you should:

- Submit all VHDL code that you wrote and received.
- Hand in a report consisting of the following:
  - Introduction to the entire project containing a description of the whole process
  - At least a paragraph on how you solved each exercise that you had to write code for:(4,7,8,9,11,12,13,14,15).
  - If the exercise asks for a drawing, there should be at least one extra paragraph to explain the drawing.
  - A final conclusion paragraph of the project with reflections, comments and/or suggestions.
  - The report can have more explanations and descriptions, than what is described here. Your grade is proportional to the report contents and structure. (but you should not be redundant.)
  - The report should have clear identification of everyone involved (name and number).
  - Finally, it should be submit as a PDF document.
- The project should be done by two people or alone.
- Exercise 15 now includes an option to also synthesize your circuit.
- Important deadlines:
  - The final delivery date of the VHDL code and report is June 26 2020 23:59 Amsterdam Time.
  - You should complete all exercises up to 11 (included) by April 24 2020 23:59. You should bring your solution to the tutorial lecture on that day to be discussed.
  - The deliveries are done through Brighspace in both cases.
  - The resit deadline for this assignment is on July 10th 2020.

EXERCICES

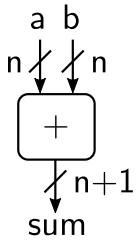
1 Simulate the 4-bit adder architecture

The first design we will simulate, is a 4-bit adder, with a and b as inputs and sum as the output. Open the file and try to understand the VHDL code of both the design module ("add4.vhd") and the testbench ("tb_add4.vhd"). Next, run the testbench and check if the output waveforms correspond to the expected behavior.



2 Simulate the n-bit adder architecture

Try to understand the VHDL code of the n-bit adder ("addn.vhd") and testbench ("tb_addn.vhd"). Notice that n gets the value 8 in the testbench. Run the testbench and check the outputs.

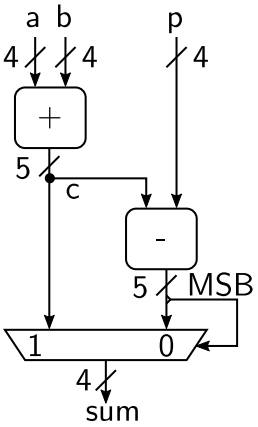


3 Simulate the 4-bit modular adder architecture

The 4-bit modular adder computes the addition of the inputs (a and b), and subtracts the modulus (p) from the intermediate result (c), with d as a result. The final result of the modular addition is either c or d, depending on the sign of d. The sign is determined by the MSB, i.e. the most-significant bit. The multiplexer drives the output in the following way:

- If the MSB of d is 0, $d \geq 0$ and sum = d;
- If the MSB of d is 1, $d < 0$ and sum = c.

Use the design module ("modadd4.vhd") and the testbench ("tb_modadd4.vhd") to understand and simulate the design.

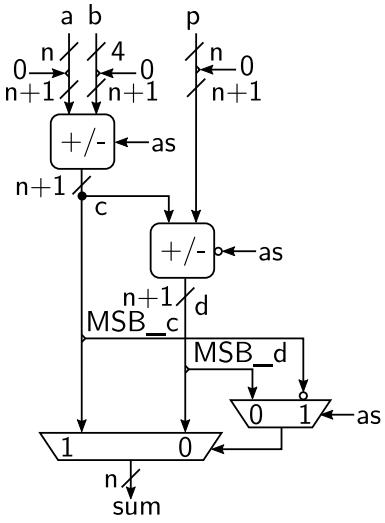


4 EXERCISE: Design and simulate an n-bit modular adder architecture

Design an n-bit modular adder architecture starting from the design module template ("modaddn.vhd"). Use the testbench ("tb_modaddn.vhd") to simulate the design with n = 8. Change the testbench such that it can handle 128-bit input and output values. Use hexadecimal representation for the input values. Example: a <= x"AB"; is the same as a <= "10101011";

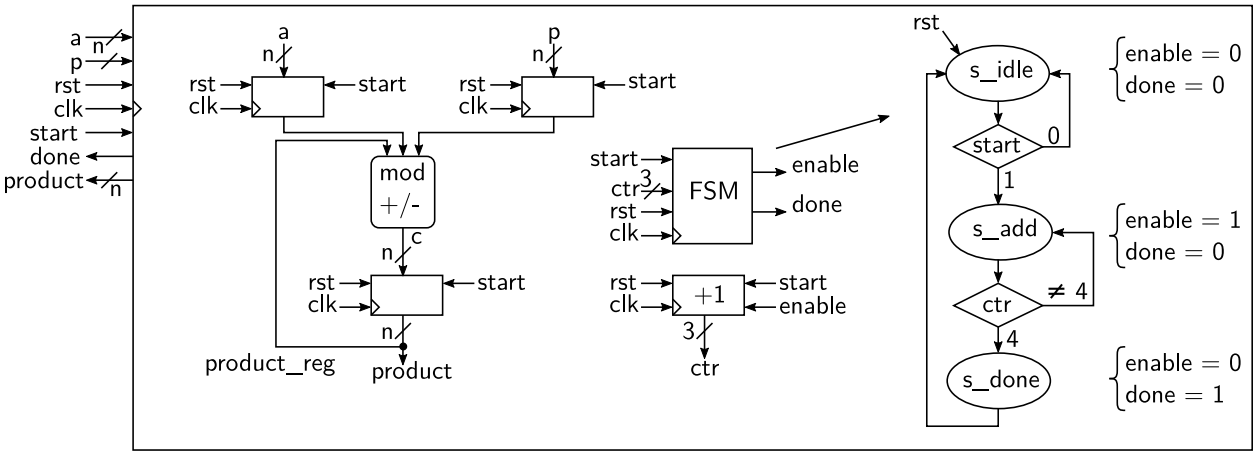
5 Simulate the n-bit modular adder/subtractor architecture

The modular adder/subtractor performs either a modular addition (when the add/subtract signal, as, is 0) or a modular subtraction (when as is 1) on the inputs a and b with modulus p. Try to understand the architecture and the design file ("modaddsubn.vhd"). You also need "addsubn.vhd" as a submodule. Simulate the behavior of the module using the testbench ("tb_modaddsubn.vhd"). We will use this module in the design of the elliptic curve point doubling.



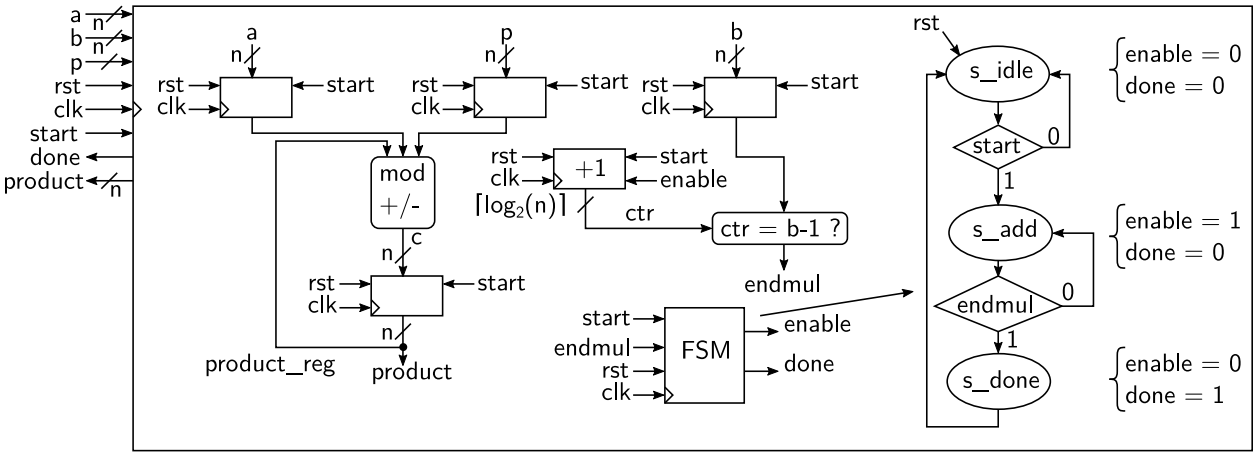
6 Simulate the n-bit constant multiplier (multiplication by 5)

This is a module that we will not use in the design of the elliptic curve point doubling, but it should be a good starting point for the following two exercises on n-bit modular multipliers. Try to understand the architecture and the design file ("modaddn_mult5.vhd") and run the testbench ("tb_modaddn_mult5.vhd"). The architecture of the design module contains input registers to store a and p. New values are loaded when start = 1. The start signal also initiates a finite state machine (FSM) that interacts with a counter to make sure the modular addition is performed 5 times.



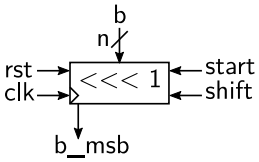
7 EXERCISE: Design and simulate an n-bit multiplier (by consecutive modular additions) and simulate the behavior

Start from the design module template ("modaddn_mult.vhd") and use the testbench ("tb_modaddn_mult.vhd"). The goal is to build a modular multiplier by consecutive modular additions of the multiplicand. Note that this is not an efficient way of implementing a modular multiplier, because it requires an impractical number of modular additions to be executed.



8 EXERCISE: Design and simulate an n-bit modular multiplier (through a left-to-right modular double-and-add algorithm)

Start from the design module template ("modmultn.vhd") and use the testbench ("tb_modmultn.vhd"). The architecture of the design module contains input registers to store a, b and p. The register to store b is also shiftable. When the shift input is active, the register shifts its content one position to the left (and shifts in a 0 on the right side). The registers are already present in the template design file.



The FSM uses the left bit of b to determine the next step in the algorithm. First, draw the FSM and the hardware architecture. Then, design the architecture in VHDL. Finally, run the simulation to check if the product is as expected.

Algorithm 1 Left to right addition chain

Inputs: $a = (a_{n-1}, ..., a_0)$, $b = (b_{n-1}, ..., b_0)$, $p = (p_{n-1}, ..., p_0)$
Output: $b \cdot a \bmod p$

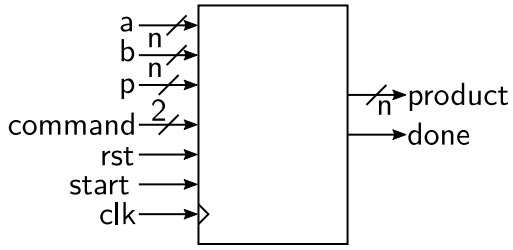
```
s ← 0
for i ← n − 1 to 0 by -1 do
    s ← s + s
    if b_i = 1 then
        s ← s + a
    end if
end for
```

9 EXERCISE: Draw, design and simulate a n-bit modular multiplier/addition/subtraction

In this exercise you have to take the previous unit that was able to perform multiplication and expanded it to also perform modular addition/subtraction. To choose the operation in the arithmetic unit an extra input port called "command" of two bits is added. This extra input will works as follows:

command	
00	Don't do nothing
01	Modular multiplication ($a \cdot b \bmod p$)
10	Modular addition ($a + b \bmod p$)
11	Modular subtraction ($a - b \bmod p$)

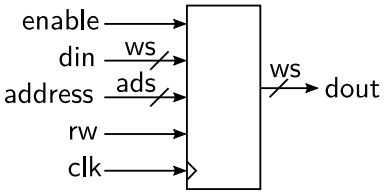
The core will work the same as in the multiplication operation before, it will begin to compute when "start" is equal to 1, and it will drive done to 1 when it has finished. And the outside black box will become:



To understand what should be changed in the code, you should first change the drawing from the previous exercise to support the extra operations. Therefore you should add necessary gates such as and, or, xor or multiplexers, and also change the state machine to perform the addition/subtraction operation. After you finish the new drawing, you fill the code in "modarithn.vhd" and test it with the testbench "tb_modarithn.vhd".

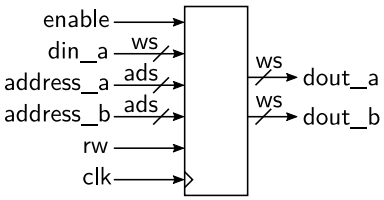
10 Simulate one port memory

In order to make our elliptic curve core we need a memory to store constants, intermediate values, input point and output point. We could use registers and flip-flops to make those memories, but since we don't need all those values at the same time it is better to have them sharing the same interface. Also, these memories are more optimized in terms of resources and need less to store the same amount of bits.
Simulate the memory unit "ram_simple.vhd" with the testbench "tb_ram_simple.vhd" and understand its inner workings. Also, pay attention on the "ram_simple.vhd" on how to construct an array of std_logic_vectors.



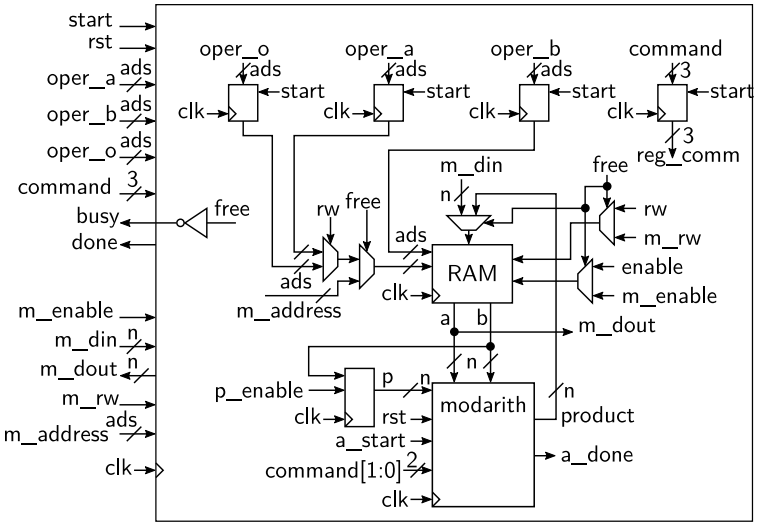
11 EXERCISE: Design and simulate a two ports memories system.

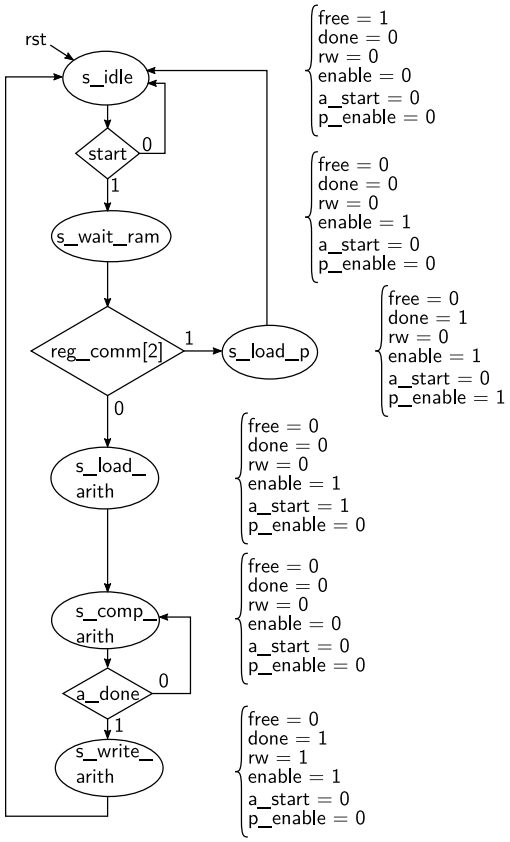
Given the "ram_simple.vhd" before as a component, construct a similar unit called "ram_double" which can perform two reads in one cycle instead of one. This will be necessary for our ECC unit, since the operations are described by two inputs and one output. After describing your unit, test it with the testbench "tb_ram_double.vhd" or use the testbench to understand how the signals work.



12 EXERCISE: Design and simulate the arithmetic unit with the double memory, and instructions

The next step for our architecture is to join the double ram with the arithmetic unit. By joining them, we can load values from the ram into the arithmetic unit, compute the desired operation, and finally write the output into the ram. This will be the latest step, since we can iterate through all instructions necessary to compute the elliptic curve point addition and doubling.
The architecture is described below:





In this architecture we can divide the external connections as the regular ones to send the commands, and the ones to interface with the memory ram. The external memory ram interface has been added, thus it is possible to load and store the elliptic curve point or some other necessary constants. This interface is only accessible when the unit is free and not executing any command.
The architecture supports all the same commands as the previous one, but it has one extra command : load value into prime register "p".

command	
000	Don't do nothing
001	Modular multiplication $(a \cdot b \bmod p)$
010	Modular addition $(a + b \bmod p)$
011	Modular subtraction $(a - b \bmod p)$
100	Load p $(p \leftarrow b)$
101,110,111	Don't do nothing

The purpose of this command is to just load a value from the memory ram to be the prime used by the arithmetic unit. It is a separate command and must be distinguished in the state machine.
Write this circuit in the file "ecc_base.vhd" and simulate with the provided testbench "tb_ecc_base.vhd".

13 EXERCISE: Design and simulate a point addition/doubling architecture

In this exercise you should extend the previous architecture in order to be able to perform point addition or point doubling for Weierstrass elliptic curves. In our elliptic curve core we will be using the complete formulas of Renes, Batina and Costello [2]. Those formulas work with all elliptic curves with prime order and in Weierstrass format, but in this exercise we will try to focus on the NIST P-256 curve [1]. The formulas below are also only for Weierstrass curves with constant $a = -3$.

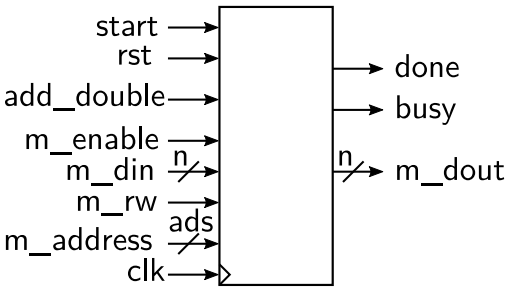
Addition formula.
(X3,Y3,Z3) = (X1,Y1,Z1) + (X2,Y2,Z2)

```
t0 = X1*X2;    t6 = X2+Z2;    t6 = t6-t0;
t1 = Y1*Y2;    t5 = t5*t6;    t1 = t6+t6;
t2 = Z1*Z2;    t6 = t0+t2;    t6 = t1+t6;
t3 = X1+Y1;    t6 = t5-t6;    t1 = t0+t0;
t4 = X2+Y2;    t7 = b*t2;     t0 = t1+t0;
t3 = t3*t4;    t5 = t6-t7;    t0 = t0-t2;
t4 = t0+t1;    t7 = t5+t5;    t1 = t4*t6;
t3 = t3-t4;    t5 = t5+t7;    t2 = t0*t6;
t4 = Y1+Z1;    t7 = t1-t5;    t6 = t5*t7;
t5 = Y2+Z2;    t5 = t1+t5;    Y3 = t6+t2;
t4 = t4*t5;    t6 = b*t6;     t5 = t3*t5;
t5 = t1+t2;    t1 = t2+t2;    X3 = t5-t1;
t4 = t4-t5;    t2 = t1+t2;    t7 = t4*t7;
t5 = X1+Z1;    t6 = t6-t2;    t1 = t3*t0;
                              Z3 = t7+t1;
```

Doubling formula.
(X3,Y3,Z3) = (X1,Y1,Z1) + (X1,Y1,Z1)

```
t0 = X1*X1;    t4 = t1-t5;    t3 = t0+t0;
t1 = Y1*Y1;    t5 = t1+t5;    t0 = t3+t0;
t2 = Z1*Z1;    t5 = t4*t5;    t0 = t0-t2;
t3 = X1*Y1;    t4 = t4*t3;    t0 = t0*t6;
t3 = t3+t3;    t3 = t2+t2;    t2 = Y1*Z1;
t6 = X1*Z1;    t2 = t2+t3;    t2 = t2+t2;
t6 = t6+t6;    t6 = b*t6;     Y3 = t5+t0;
t5 = b*t2;     t6 = t6-t2;    t6 = t2*t6;
t5 = t5-t6;    t6 = t6-t0;    X3 = t4-t6;
t4 = t5+t5;    t3 = t6+t6;    t6 = t2*t1;
t5 = t4+t5;    t6 = t6+t3;    t6 = t6+t6;
                              Z3 = t6+t6;
```

In this exercise you will design the main unit "ecc_add_double.vhd", since the the testbenches "tb_ecc_add_double_small.vhd" and "tb_ecc_add_double_nist.vhd" are provided. The first testbench is for a small elliptic curve with prime = 127 and constants $a = -3$ and $b = 5$, while the second is for the NIST P-256. Because the testbenches are given, the main unit "ecc_add_double.vhd" external interface should follow the drawing below, and the memory arrangement:



Memory address	value
00000	prime
00001	a
00010	b
00011	X1
00100	Y1
00101	Z1
00110	X2
00111	Y2
01000	Z2
01001	X3
01010	Y3
01011	Z3
01100	t0
01101	t1
01110	t2
01111	t3
10000	t4
10001	t5
10010	t6
10011	t7
11111,...,10011	Don't care

The circuit should work with a "start = 1" to initialize the operation, where point addition is done when "add_double = 0", and doubling should occur when "add_double = 1". The "busy" signal should be 1 when the unit is performing computations, and one cycle before it finishes the signal "done" should go from 0 to 1, where in the it should go back to 0 in the next cycle. When is not doing computations the signal "busy" should be 0.
By having the communication and external interface fixed, it is possible for the testbench to work with any circuit, as long as this interface is correct.
In order to construct this circuit, you just need to add in the previous exercise "ecc_base.vhd" another core that will give the necessary instructions. This extra core can work as a CPU fetcher where it would load instructions from memory until it reaches the final instruction. Otherwise, this extra core can be a state machine with all instructions embedded into it, which are around 77 arithmetic instructions. A special fetcher is more easy to scale, but is more complicate, and the state machine approach is more simple and easy to understand, but it will be a lot of states to declare.

14 EXERCISE: Design and simulate a scalar multiplication through Montgomery Power ladder.

In the final exercise you should draw, design and simulate an architecture which performs the scalar multiplication. Also, you should create a testbench for the NIST P-256 with a at least the test below. You are free to add more tests to help you debug, for example one with 5 or 8 bits scalar. But in the end the test

below should be added:

- scalar(s) = 0xC03A898C5E674B2CE564F25F96BB8AE944985061ACCE54CAA5554BB508542151
- Gx = 0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296
- Gy = 0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5
- Gz = 0x0000000000000000000000000000000000000000000000000000000000000001
- sGx = 0xB586EFD756C25F6BC6469AA162BAC531C877C99DF5CBD8F95EEF31CE74226860
- sGy = 0x8E5C8AAD3B74642DBF40A6851090A6DB6210C97AFE36CCF65300CC2F6514DE66
- sGz = 0x5590FD84B53850234D3CEF495D7DB307470C449A1CA8431F184D4DDDB70B3714

The algorithm for scalar multiplication should be the Montgomery power ladder:

Algorithm 2 Montgomery power ladder

Inputs: $k = (k_{n-1}, ..., k_0)$, $G = (G_x, G_y, G_z)$
Output: $R_0 \leftarrow k \cdot G$

```
R0 ← (0, 1, 0)
R1 ← G
for i ← n − 1 to 0 by -1 do
    if b_i = 1 then
        R0 ← R0 + R1
        R1 ← 2R1
    else
        R1 ← R0 + R1
        R0 ← 2R0
    end if
end for
```

Two ideas to solve this problem: through conditional swaps of the values or by memory addresses. In the conditional swaps, you would change your previous design to perform doubling in another memory position, and then after performing addition and doubling you would have to copy/swap the contents from one memory place to another. Another option is to make addition and doubling operations work on any input/output position, or at least in two possible positions. Therefore when executing, the position would be given by the scalar bit.

15 EXERCISE: Design and simulate an advanced unit

This exercise you should expand the previous design by any of the ideas below:

- Optimization:
  - Faster elliptic curve scalar multiplication algorithm, like a window based method.
  - Two hardware units, one computing point addition while the other computes point doubling.
- Side-channel:
  - Scalar blinding.
  - Randomized projective coordinates.

- Synthesis results:
  - Download and install Xilinx Vivado HL WebPACK Edition.
  - Synthesize your circuit from exercise 14 for the Xilinx Kintex 7 - xc7k160tfbg676, fix all warnings and add the results in the report.

Just like the previous exercise you should make a drawing, design and create a testbench for this exercise.

Questions

You can contact me by email: L.Weissbart@cs.ru.nl

References

[1] Mehmet Adalier et al. Efficient and secure elliptic curve cryptography implementation of curve p-256. In *Workshop on Elliptic Curve Cryptography Standards*, volume 66, 2015.

[2] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. Cryptology ePrint Archive, Report 2015/1060, 2015. `https://eprint.iacr.org/2015/1060`.