IPECC

An open-source VHDL IP for generic elliptic curve cryptography over $\mathbb{F}_p$ with emphasis on side-channel resistance

Agence Nationale de la Sécurité des Systèmes d'Information

ANSSI

Agence Nationale de la Sécurité des Systèmes d'Information

Contents

Chapter 1

Abstract

IPECC is a hardware IP block performing the computation of scalar multiplication $[k]\mathcal{P}$ over any elliptic curve defined on a finite field of charasteristic $p > 3$. It is intended as a hardware accelerator to be embedded onto a silicon die to ensure highly reliable still efficient elliptic-curve based signature verification and Diffie-Hellman secure key exchange. IPECC has been developped with the least possible assumptions on the underlying technology and could be indifferently targeting any SRAM-based FPGA vendor/device or any ASIC flow. Typical applications include:

- secure SoC enclave and root-of-trust
- smartcard microcontroller peripheral
- academic/lab side-channel and fault test-vehicle
- SoC security analysis based on FPGA prototpying.

The size of prime p, denoted `nn` in the code and in the remainder of this document, defines the level of cryptographic security. By definition all large numbers used for cryptographic computations are `nn`-bit long. Parameter `nn` is statically defined by the designer at synthesis time and can be chosen to be any integer value. All you need for that is to edit the value of a unique HDL constant. The limitation only comes from the logical and memory ressources of the target part, as the amount of logic and memory consumed by the IP will obviously increase with the value of `nn`. An optional feature, named *dynamic prime size* feature, allows to dynamically set the size of large numbers, provided they stay below the value statically set to parameter `nn` (which in this case simply becomes the maximal size allowed).

IPECC is intended for production purpose as well as academic research:

- While targeting an end product, it will exhibit very good performance as it relies on Co-Z arithmetic to reduce the area of logic by a factor of almost two without impacting throughput nor latency. Countermeasures against side-channel attacks have been designed with a *defense-in-depth* state of mind, meaning that security does not rely on one unique countermeasure but in the contrary should be based on the application of several layers of defense. Thus every countermeasure can be hardware-locked at synthesis time to enforce a hardened, secure implementation, or left software-disengageable by designer instead. Available side-channel countermeasures are: *double-and-add-always*, *address-masking of $\mathcal{R}_0/\mathcal{R}_1$ points* (aka anti address bit DPA or ADPA), *blinding* (randomization of the private exponent), *Z-masking* (randomized projective coordinates) and *memory shuffling*. The computation time is perfectly constant and will depend solely on the value of `nn` as long as no arithmetic exception is met in the course of $[k]\mathcal{P}$ computation (like small-order base point causing a null intermediate point). The sequence of arithmetic operations implementing curve formulæ for each bit of the scalar at the underlying field level is unique, constant, and involves not a single conditional jump.

- For academic research purpose, IPECC provides a powerful and versatile testbed to analyze side-channel and fault attacks's practicability and efficiency. When synthesized in debug mode all countermeasures become optional and can be enabled or disabled dynamically through software programming. Conversely, it is straightforward to force usage of part or all of the countermeasures to allow for leak measure and detection, signal-to-noise ratio measures, or active stress testing/perturbation. Debug features allow breakpoint setting and step-by-step execution. Along with trigger-out signal generation to remote instruments (e.g oscilloscopes/EM injection probes) they allow for clear isolation of specific instruction, operation or computation, pattern detection and step-by-step leak detection.

IPECC architecture is very simple and allows for partial reprogrammation, as it is partially based on microcode execution of embedded software routines (hence the possibilty to set breakpoints as stated previously) which can be easily edited if one wants to implement a new countermeasure or test a new attack scenario.

IPECC is written in fully synthesizable VHDL. A high-level description approach has been adopted, with no explicit instanciation of any vendor dependent hardware macro, with the sole exception of multiplier-accumulators which use black-box instanciations of so-called FPGA "DSP blocks".

IPECC comes wrapped up as an AXI-lite interface (both 32-bit/64-bit compatible) allowing easy plug-&-play integration inside any ARM AMBA ecosystem, e.g in SoC-FPGA designs or for ASIC prototyping. It is easily programmed through a small set of control & status registers and optional IRQ generation. Pre-computations involved in Montgomery representation are automatically performed by hardware upon transmission of a new value of the field prime p, and do not involve further operation whatsoever from software.

The support of IPECC for software is twofold.

- A driver is provided which supports both the standalone mode or the application mode on top of Linux. In the latter case the driver is totally resident in userland and uses the generic UIO userspace driver API of Linux to access the bank of IP registers (less orthodox `/dev/mem` direct access interface is also supported).

- A software patch is also provided to hook-up IPECC with the *libecc* software library project also available from ANSSI's GitHub (`github.com/ANSSI-FR/libecc`).

Together with *libecc* project, for which it provides a hardware accelerator off-loading processor from all point operations, IPECC provides a comprehensive and highly secured solution to implement ECC protocols like EC*DSA for SoC prototyping and design, as well as ECDH secured key exchanges between embedded/IoT devices.

IPECC is licensed under the Gnu Public Licence version 2 (GPLv2) and available for download from ANSSI's GitHub at `https://github.com/ANSSI-FR/IPECC`.

Chapter 2

In a nutshell

If you're already familiar with elliptic curve cryptography and hardware design principles, the following should give you the technical bottom line.

Cryptographic aspects – The operation performed by IPECC is the $[k]\mathcal{P}$ computation on any elliptic curve over a prime field given in its Weirstraß from, even curves of composite order. Binary curves are not supported. There is no limit to the size `nn` of big numbers $\mathbb{F}_p$-arithmetics are based on. As long as hardware ressources are available in your circuit, you should be able to increase the level of security. You can choose to either set parameter `nn` as a constant statically for your design or to allow software driver to change it dynamically, in which case the value you statically set for `nn` becomes the runtime maximum allowed value. Aside $[k]\mathcal{P}$ computation, software API of the IP also offers the following point operations: addition ($\mathcal{P}+\mathcal{Q}$), doubling ($[2]\mathcal{P}$) and negation ($-\mathcal{P}$). Three logical tests are also provided: given the coordinates of one point, the IP tests if the point belongs to a curve; given the coordinates of two points, the IP tests if they are equal, or if they're opposite. Side channel countermeasure only concerns $[k]\mathcal{P}$ computation and aim at protecting the scalar k.

Hardware architecture – The IP is divided into a few components stacked functionnaly on top of one another (see fig-2.1 in p. 8). Top level component `ecc_axi` handles the AXI interface, holding the register bank and servicing the requests issued by software. Below is `ecc_scalar` that handles the main hardware state machine. Below is `ecc_curve` which is a tiny CPU operating from a dedicated microcode memory named `ecc_curve_iram`. The content of this memory is read-only and set at synthesis time (though debug mode allows to patch it at runtime) through the static link of a bunch of assembly source files[1]. The CPU is very simple, holds a 3-stage pipeline, no stack nor stack pointer, no general purpose registers nor register window, no cache, no memory hierarchy, no I/O, no call convention, no exception, no interrupt, no privilege level. Below component `ecc_curve` is component `ecc_fp` which can be considered as the Arithmetic Logic Unit (ALU) of `ecc_curve`. Instructions are fetched from the microcode memory by `ecc_curve`, decoded, and then transmitted to `ecc_fp` to be carried out from its internal memory (called the «IP internal memory of large numbers», this is component `ecc_fp_dram`) which by default can hold 32 large numbers (each of size `nn` bits). There is no memory management unit hence all addresses are physical, both for data and instructions. Instructions supported by `ecc_curve` fall into two categories: arithmetic operations and branch intructions. Main arithmetic instructions are: addition (mnemonic `NNADD`) subtraction (`NNSUB`), Montgomery multiplication (`FPREDC`) left and right bit shift (`NNSLL` and `NNSRL`), bitwise XOR (`NNXOR`). An instruction named `NNRND` also exists that is used to initialize with ramdomness any large number. Arithmetic is signed and uses two's complement representation. Instructions `NNADD` and `NNSUB` should be considered operating

[1] Assembly language is the only source file entry format. No higher-level language such as C or C++ is supported, nor is foreseen to be. The syntax of the assembly is very simple and uses AT&T order (e.g `NNADD s0 s1 dest`).

on integers only while `FPREDC` operate on field elements. In other words neither `NNADD` nor `NNSUB` directly perform reduction, while the Montgomery multiplication carried out by instruction `FPREDC` by definition maintains the reduced form of its inputs on its output. Whenever the result of an addition or subtraction requires reduction, it is carried out with a supplementary conditional subtraction and/or addition instruction, but always in constant time. This is made possible by a hardware mechanism named *on-the-fly patching* implemented inside `ecc_curve` (see next paragraph). All instructions are carried out synchronously (i.e the destination operand of instruction i is updated in memory before decoding instruction $i+1$) with the exception of Montgomery multiplications. Instruction `FPREDC` is handled asynchronously, meaning the REDC operation is posted by `ecc_fp` to a dedicated hardware Montgomery multiplier, this is component `mm_ndsp`. The support is for 1 to 4 `mm_ndsp` instances of `mm_ndsp`, however Co-Z arithmetic allows to reduce this to 2 without creating any wait-state that could be caused by dependency between intermediate terms. The number of `mm_ndsp` instances can still be reduced to one if you want to reduce design area, obviously at the cost of a reduced performance. The number of multiplier-accumulators (usually named «DSP-blocks» in the FPGA ecosystem) inside each component `mm_ndsp` is also configurable at synthesis time so as to allow you to set your own cursor in the area-speed trade-off. If software requires synchronization between instructions, it can use the special `BARRIER` instruction to guarantee strict ordering.

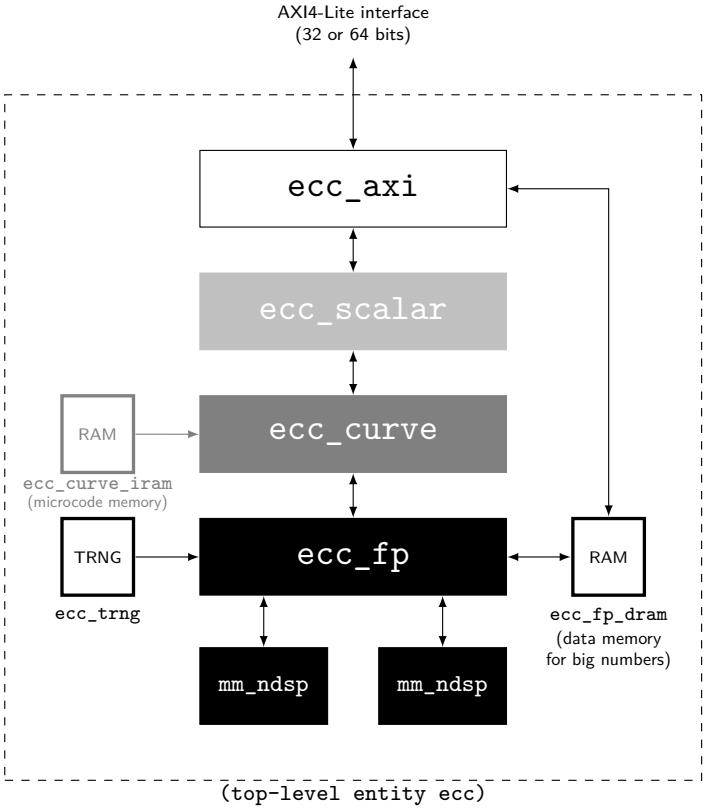


Figure 2.1: IPECC hardware architecture

Debug mode versus production mode – When synthesizing the IP you must choose (see below paragraph *Customization*) between two static exclusive modes: debug mode or production mode. Debug mode is synonym of *totally unsecure*. Production mode is synonym of *highly secure*. When in debug mode software driver can tamper in the IP internal operations in any way that the debug

registers allows it to. It can set breakpoints, read raw random data and modify them, modify the content of the internal memory of large numbers, program generation of an output trigger signal precise to the clock-period, modify content of the microcode memory, configure some of the physical parameters of the TRNG, disable or enable any of the side-channel countermeasures, some among the built-on ones. The debug mode is thus intended for pre-production analysis, hardware prototyping, academic studies on faults and side-channels. In production mode however all the countermeasures, both the built-in ones and the optional ones are enforced according to what was statically set at synthesis time. The data paths used by some of the debug features described previously are simply pruned (e.g read of the TRNG FIFO). When in production mode, if a countermeasure is not set statically, like blinding, shuffling or Z-coordinate randomization, it's still possible to runtime-modify the static configuration if it goes towards increasing the level of security. To emphasize the parallel between debug/production modes and the secure/unsecure property, you'll constantly see throuhgout this documentaion and throughout the code the reminding expression *«in debug (unsecure) mode versus in production (secure) mode»*.

On-the-fly patching – On-the-fly patching allows modifying the address of source and/or destination operands of an instruction opcode depending on different hardware low-level signals which are critical from the perspective of side-channel security. First, and as stated previously, this is how constant time reductions are performed. Second, and most importantly, the patches are also the mechanism responsible for balancing, on the side-channel level, the address in the memory of large numbers at which sensitive variables are read and written, that would otherwise leak information on the scalar (that mainly includes the X and Y coordinates of points $\mathcal{R}_0$ and $\mathcal{R}_1$ involved throughout the entire right-to-left scalar loop). The logic implementing on-the-fly patching is strictly hardwired and implies strong collaborative aspects between hardware and microcode software, making it sometimes difficult to deintricate from an outside look. You'll probably don't want to modify this logic at all, unless you really know what you're doing. Last, the patching mechanism is also used to manage the *exceptions* that may happen theoretically, though very rarely, in the course of point operations. An example of an exception is when the two points $\mathcal{R}_0$ and $\mathcal{R}_1$ turn out to be equal, or opposite. As the formulæ used by IPECC to compute point additions and doubling don't make use of the so-called unified formulæ [BJ02], meeting such an exception calls for a specific re-routing of the execution flow, thus leaking side-channel information both in time and address of the microcode. Instructions requiring dynamic patching have their mnemonic suffixed with `,p` followed by a patch ID number (e.g `NNADD,p17`) in the assembly source files. During the assembly of source files in order to build the microcode, the patch ID numbers are translated into a 6-bit binary field to fit in the instruction opcode, which makes a total of 64 patchs available (among which almost are already used).

Side-channel countermeasures – The algorithm implemented by IPECC to compute $[k]\mathcal{P}$ is given in p. 30 (algorithm 13). It is a double-and-add-always right-to-left loop [Joy07] that has been adapted in order to fit the ADPA (anti address-bit DPA) countermeasure. Algorithm starts by checking that base point actually falls on the curve and agrees to return the coordinates of the result point only if it does too. Curve arithmetics use the Co-Z refinement of the Jacobian projective coordinates [Mel07, VD10, GJM10, GJM$^+$11] meaning register points $\mathcal{R}_0$ and $\mathcal{R}_1$ share the same z-coordinate. The complexity of formulæ for point addition and doubling are almost halved as compared to non Co-Z solutions, which leads to a reduction, in our implementation, of almost 50% of the surface at equal speed. Countermeasures against side-channel attacks split in built-in ones and optional ones. Built-in countermeasures are constant-time execution, ADPA, double-and-add-always, and projective coordinate randomization, also known as z-coordinate masking or simply z-masking. Optional countermeasures are scalar blinding [Cor99], big number memory shuffling and isomorphic curve randomization [JT01]. The final inversion of the z-coordinate in order to switch back to affine coordinates is performed using a modular exponentiation to power $p-2$ (instead of Euclidian algorithm which is non constant time).

Random generator – IPECC embeds its own True Random Number Generator, based on ES–TRNG design by KU-Leuven ([YRG+18]). No post-processing function is provided with the IP, though the HDL structure was made to easy the integration of a custom one in a straightforward manner. A set of FIFOs is used to buffer the random data. Raw random bits are first buffered into a so-called raw random FIFO. Then they're pulled back and gathered to shape internal random numbers of different word sizes, then pushed back into one of 4 possible downtream FIFOs, according to a round-robin arbitration schedule. If you're familiar with the AIS31 standard terminology from german BSI, the first FIFO stores what is called *raw random bits* while the four downstram FIFOs store what is called *internal random numbers* (IRN). But IPECC provided as it is doesn't include any post-processing function so there's no real difference between raw random bits and internal random numbers except their bit width. Now if you choose to implement your own post-processing function (which is recommanded) be advised that it is a cryptographic hardware design of its own, usually implying the design of a hash function. You'll then have to instanciate your logic inbetween the raw random FIFO and the internal random number ones. Each of the 4 FIFOs serves its random numbers to one of component in the IP (acting like a client) that relies on randomness to implement one of the side-channel countermeasures. These components are: `ecc_axi` (for on-the-fly masking of the scalar), `ecc_curve` (for internal shuffling of the 4 coordinates X and Y of $\mathcal{R}_0$ and $\mathcal{R}_1$, a feature closely related to the patching mechanism described above), `ecc_fp_dram_sh` (this is the component that replaces `ecc_fp_dram` when shuffle countermeasure is present in the IP to implement complete shuffling of the memory of large numbers) and `ecc_fp` (for implementation of the `NNRND` instruction). You can choose at synthesis time the number of instances of ES–TRNG that you want to fit in your own design, with no restriction, depending on your assessment of the random genration throughput your own IP configuration will need (the ES–TRNG by itself is *very small* as it only consumes a few LUTs and a few DFFs in an FPGA). The RTL code will then automotically instanciate a binary routing tree including all the instances of the raw TRNG in order to gather their different binary outputs and multiplex their entrance in the raw random FIFO. In order to allow entropy assessment of a particular floorplan, the IP allows software to read the content of the raw random FIFO (only when configured in debug mode), a feature supposedly interesting only for FPGA targets in our opinion.

Hardware integration – IPECC is wrapped-up as an AXI4-lite compatible IP, which in practice means that you'll be able to directly plug it in any AXI-based interconnect system-on-a-chip such as an ARM or RISC-V application processor. IPECC then becomes a cryptographic peripheral memory-mapped in the overall physical address space of the system and, used in conjunction with a software driver running on the CPU, can be programmed to perform accelerated $[k]\mathcal{P}$ computations on behalf of the processor without requiring further CPU ressource. Given the rather long computation time required to perform scalar multiplication and given the small amount of data required to transfer both input and output data to and from the IP, no DMA interface is required and therefore IPECC only presents itself as an AXI slave/subordinate interface (whatever you prefer to call this).

Driving the IP from software – As the IP is simply an AXI subordinate interface, the piece of software driver to control its operations can stay extremely simple. Pseucode examples illustrating how to perform the main curve and point operations are given in Appendix E, but above all an official driver is provided (C sources in folder `driver/` of the repo) supporting both standalone/bare-metal mode and applications running over the GNU/Linux operating system, based on the UIO user-mode driver. Programming a $[k]\mathcal{P}$ computation simply requires software to transfer number p, curve parameters a and b, base-point coordinates $x_{\mathcal{R}_1}$ and $y_{\mathcal{R}_1}$ and scalar k (plus the order q of the curve if blinding countermeasure is desired) using a combined procedure of read/write from/to the main control and status registers (`W_CTRL`/`R_STATUS`) and write and read data registers (`W_WRITE_DATA`/`R_READ_DATA`). Registers are fully described in Appendix §C.

Curve compatibility and limitations – IPECC supports any elliptic curve over prime fields (of prime or composite order) as long as it's defined by its Weierstrass equation. There exists a one-to-one correspondance between all elliptic curves and their description with the Weierstrass equation, however it's not always a trivial work to translate the curves that are used in some standard elliptic curve protocols without being explicitely defined by the Weierstrass equation into the (a, b) representation required by IPECC. For such curves you should consider using IPECC together with the *libecc* highly versatile software library (and for other curves too) as it will automatically and transparently perform the mathematical transposition into the Weierstrass form. Furthermore the highest level of elliptic curve operations that IPECC can do is the $[k]\mathcal{P}$ computation. It does not implement any elliptic curve signature protocol nor elliptic curve key-exchange protocol. If you need to either produce or verify elliptic curve signatures based on any of the EC*DSA-like signature schemes, you'll need to consider using a hashing function on top of $[k]\mathcal{P}$ computation, either a software one of your own or by using a hash-function hardware accelerated IP. Actually this is all the more reason why you should consider using *libecc* as the main software component on top of the IP.

Customizing the IP – All static parameters used statically configure the IP are gathered in a unique VHDL package described in source file ⟨`ecc_customize.vhd`⟩. A comprehensive description of the different parameters can be found at the bottom of the file, which is also featured in Appendix A. Customization allows you to:

- choose the maximum level `nn` of cryptographic security
- choose whether the cryptographic level of security `nn` should be fixed or dynamically modifiable (that's the *dynamic prime size* feature)
- select between debug (unsecure) or production (secure) mode
- select the technology (FPGA or ASIC)
- set the number of instances of the hardware Montgomery multiplier (1 or 2) and, inside each one of these, the number of multiplier-accumulators you want to instanciate.
- select the read latency of SRAM blocks in the design (1 or 2 cycles)
- select to have the Montgomery multipliers share the same clock domain as the main logic or to place them instead in their own specific clock-domain (for specific floorplan optimization)
- select the optional side-channel coutermeasures that you want to hardware-lock in your design, or left disengageable
- select the number of true physical random genrators to instanciate, so as to fit your own random genration thoughput
- set the main physical parameters of the TRNG
- set the AXI interface to be 32-bit or 64-bit (it's advised to keep 32)
- change the default size of the IP internal memory of large numbers and the default size of the microcode memory (two things you should consider with caution).

Quick start – A tutorial is given in appendix F to bootstrap a small design using the IP on a low cost Xilinx SoC-FPGA board that can serve you as a base point to later meet your application specifics. The tutorial is given in a shoft version and a long, verbose one, the second one being rather intended for students and/or beginners in the the FPGA ecosystem.

Chapter 3

Elliptic curve cryptography

This short chapter briefly introduces elliptic curve cryptography or ECC (§3.1) and quickly exposes the mathematical aspects needed to understand the operations performed by IPECC (§3.2 and §3.3). The reader who wants to get more detail should refer to the many excellent textbooks on the subject (e.g., [BSS+99], [HM11]).

3.1 Definition of elliptic curves

An elliptic curve over a field $\mathbb{K}$ is formed by the set of points $\mathcal{P} = (x, y) \in \mathbb{K} \times \mathbb{K}$ satisfying the *short-Weierstraß* equation:

$$E_{/\mathbb{K}} : y^2 = x^3 + ax + b \tag{3.1}$$

to which one must also add the point at infinity, denoted $\mathcal{O}$. The numbers x and y are the *affine* coordinates of point $\mathcal{P}$. In the context of IPECC, the field $\mathbb{K}$ denotes a prime field of characteristic $p > 3$, so $\mathbb{K} = \mathbb{F}_p$. A necessary and sufficient condition of existence of $E_{/\mathbb{K}}$ is that quantity $\Delta \neq 0$, where $\Delta = -16(4a^3 + 27b^2)$.

An elliptic curve presents the mathematical structure of an additive group. What makes elliptic curves particularly suited for cryptographic applications ([Mil86]) is that the discrete logarithm problem in elliptic curve groups is harder than in groups previously considered. As a result, with shorter key lengths, comparable levels of security can be achieved.

The basic operation in elliptic curve cryptography is the *scalar multiplication*, that is, given a point $\mathcal{P} \in E_{/\mathbb{F}_p}$ called *base point*, one has to compute:

$$\mathcal{Q} = [k]\mathcal{P} \stackrel{\Delta}{=} \mathcal{P} + \mathcal{P} + \cdots + \mathcal{P} \tag{3.2}$$

(k times). The discrete logarithm problem consists in finding the value of k from values of $\mathcal{P}$ and $\mathcal{Q} = [k]\mathcal{P}$. All cryptographic protocols defined on elliptic curves use equation (3.2) as their core operation. Naturally, from the point of view of implementation, this equation also represents the operation which is the most consuming in time and logical ressource. It is also naturally the target of side-channel attacks, aiming at recovering the values of the bits of the scalar k from the physical observation of $[k]\mathcal{P}$ computation.

Definition (3.2) relies on the underlying additive group operation which, given two points $\mathcal{P} = (x_1, y_1)$ and $\mathcal{Q} = (x_2, y_2)$ on curve $E_{/\mathbb{K}}$, associates the new point $(\mathcal{P} + \mathcal{Q}) \in E_{/\mathbb{K}}$. The coordinates (x_3, y_3) of $\mathcal{P} + \mathcal{Q}$ are given by the *chord-and-tangent* law:

$$\begin{cases} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_1 - x_3) - y_1 \end{cases} \quad \text{with } \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \text{ if } \mathcal{P} \neq \mathcal{Q}, \\ \frac{3x_1^2 + a}{2y_1} \text{ if } \mathcal{P} = \mathcal{Q}. \end{cases} \tag{3.3}$$

As one can see, the special case where $\mathcal{P} = \mathcal{Q}$, called *point doubling*, calls for a specific computation case.

3.2 Jacobian projective coordinates

A workaround to the expensive inversions in (3.3) is to use *Jacobian projective coordinates*. In a Jacobian system of coordinates, a point $\mathcal{P} = (x, y)$ is represented by a triplet $(X : Y : Z)$ such that $(X/Z^2, Y/Z^3) = (x, y)$. This representation is based on the equivalence relation $\sim$ defined by:

$$(X_1, Y_1, Z_1) \sim (X_2, Y_2, Z_2) \quad \text{if and only if} \quad \exists \lambda \in \mathbb{F}_q^* \ \text{ such that } \begin{cases} X_1 = \lambda^2 X_2 \\ Y_1 = \lambda^3 Y_2 \\ Z_1 = \lambda Z_2 \end{cases},$$

The equivalence class containing $(X, Y, Z) \in \mathbb{F}_q^*$ is

$$(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z), \lambda \in \mathbb{F}_q^*\}.$$

$(X : Y : Z)$ is called a *projective point*, and (X, Y, Z) is called a *representative* of $(X : Y : Z)$. There is a one-to-one correspondance between the set of projective points

$$\{(X : Y : Z) : X, Y, Z \in \mathbb{F}_q, Z \neq 0\}$$

and the set of *affine points*

$$\{(x, y) : x, y \in \mathbb{F}_q\}.$$

The projective form of the Weierstraß equation (3.1) defined over $\mathbb{F}_q$ is now:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{3.4}$$

The point at infinity $\mathcal{O}$ corresponds to $(1 : 1 : 0)$, while the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

Using Jacobian projective coordinates, point addition can be performed as follows. Assuming $\mathcal{P} = (X_1 : Y_1 : Z_1)$ and $\mathcal{Q} = (X_2 : Y_2 : Z_2)$, then we have $\mathcal{P} + \mathcal{Q} = (X_3 : Y_3 : Z_3)$ with:

$$\begin{cases} X_3 = F^2 - E^3 - 2BE^2 \\ Y_3 = F(BE^2 - X_3) - DE^3 \\ Z_3 = Z_1 Z_2 E \end{cases} \quad \text{where} \quad \begin{cases} A = X_1 Z_2^2, B = X_2 Z_1^2 \\ C = Y_1 Z_2^3, D = Y_2 Z_1^3 \\ E = A - B \\ F = C - D \end{cases} \tag{3.5}$$

Implementing the formulæ (3.5) costs 12M + 4S + 7A and uses 7 field registers.

3.3 Co-Z arithmetic

Co-Z arithmetic, as the name suggests, is a form of curve point arithmetic where the two points which are being combined together share the same z-coordinate. It has been proposed first in [Mel07], as an important optimization in computation of formulæ (3.5). Simplifications are indeed made possible in this set of formulæ when points $\mathcal{P}$ and $\mathcal{Q}$ share the same z-coordinate. With notations $\mathcal{P} = (X_1 : Y_1 : Z)$ and $\mathcal{Q} = (X_2 : Y_2 : Z)$ (notice the identity of the z-coordinate of $\mathcal{P}$ and $\mathcal{Q}$) we have now:

$$\begin{cases} X_3 = D - B - C \\ Y_3 = (Y_2 - Y_1)(B - X_3) - E \\ Z_3 = Z(X_2 - X_1) \end{cases} \quad \text{where} \quad \begin{cases} A = (X_2 - X_1)^2 \\ B = X_1 A, C = X_2 A \\ D = (Y_2 - Y_1)^2 \\ E = Y_1(C - B) \end{cases} \tag{3.6}$$

with $(X_3 : Y_3 : Z_3)$ again denoting the coordinates of point $\mathcal{P} + \mathcal{Q}$.

We can now compute one point addition in 5M + 2S + 7A using 6 field registers. Compared to (3.5) the speed-up is substantial as the number of multiplications has been more than halved. Furthermore, computations derive an «update» version of point $\mathcal{P}$, that is a point representation of $\mathcal{P}$ having the same z-coordinate as the $\mathcal{P} + \mathcal{Q}$ sum, thus allowing Co-Z arithmetic to be used iteratively. Indeed, with the same notations as in (3.5) one can easily verifies that $\mathcal{P} = (B : E : Z_3)$. This allows to compute the subsequent addition $\mathcal{P} + (\mathcal{P} + \mathcal{Q})$ without incurring any overhead, which turns out to be practical when performing double-and-add iterations for $[k]\mathcal{P}$ computation (see chapter §4).

Another improvement is presented in [VD10, GJM10], in which is established the co-Z computation of point $\mathcal{P} - \mathcal{Q}$, along with $\mathcal{P} + \mathcal{Q}$. With notation $\mathcal{P} - \mathcal{Q} = (X_3' : Y_3' : Z_3)$ (notice the z-coordinate Z_3 which is the same as the one obtained in (3.6) for $\mathcal{P} + \mathcal{Q}$) we have:

$$\begin{cases} X_3' = F - B - C \\ Y_3' = (Y_1 + Y_2)(X_3' - B) - E \end{cases} \quad \text{with} \quad \begin{cases} F = (Y_1 + Y_2)^2 \\ A, B, C, D, E \text{ defined as in (3.6)} \end{cases} \tag{3.7}$$

The foregoing allows us to define two elementary operations on which co-Z arithmetic will be based. We call these operations ZADDU and ZADDC after [GJM+11]. ZADDU($\mathcal{P}$, $\mathcal{Q}$) is defined as the operation which, given two points $\mathcal{P}$ and $\mathcal{Q}$ sharing the same z-coordinate, yields the points $\mathcal{P} + \mathcal{Q}$ and $\mathcal{P}$ sharing the same (new) z-coordinate:

$$\begin{pmatrix} \mathcal{P} \\ \mathcal{Q} \end{pmatrix}_z \xrightarrow{\text{ZADDU}} \begin{pmatrix} \mathcal{P} + \mathcal{Q} \\ \mathcal{P} \end{pmatrix}_{z' \neq z}$$

ZADDC($\mathcal{P}$, $\mathcal{Q}$) is defined as the operation which, given two points $\mathcal{P}$ and $\mathcal{Q}$ sharing the same z-coordinate, yields the points $\mathcal{P} + \mathcal{Q}$ and $\mathcal{P} - \mathcal{Q}$ sharing the same (new) z-coordinate:

$$\begin{pmatrix} \mathcal{P} \\ \mathcal{Q} \end{pmatrix}_z \xrightarrow{\text{ZADDC}} \begin{pmatrix} \mathcal{P} + \mathcal{Q} \\ \mathcal{P} - \mathcal{Q} \end{pmatrix}_{z' \neq z}$$

ZADDU is also called *co-Z Addition with update* and is described in detail in algorithm 1 below. ZADDC is also called *co-Z conjugate addition* and is described in detail in algorithm 2.

Algorithm 1 ZADDU (aka co-Z addition with update)

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\mathcal{P} = (X_1 : Y_1 : Z)$ and $\mathcal{Q} = (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_p, \mathcal{P}, \mathcal{Q} \in E_{/\mathbb{F}_p}$
Output: (X_3, Y_3) and (X_1', Y_1') s.t. $\mathcal{P} = (X_1' : Y_1' : Z_3)$ and $\mathcal{P} + \mathcal{Q} = (X_3 : Y_3 : Z_3)$ for some $Z_3 \in \mathbb{F}_p$

1: $A \leftarrow (X_2 - X_1)^2$
2: $B \leftarrow X_1 A$
3: $C \leftarrow X_2 A$
4: $D \leftarrow (Y_2 - Y_1)^2$
5: $E \leftarrow Y_1(C - B)$
6: $X_3 \leftarrow D - (B + C)$
7: $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$
8: $X_1' \leftarrow B$
9: $Y_1' \leftarrow E$
10: **return** $((X_3, Y_3), (X_1', Y_1'))$

Algorithm 2 ZADDC (aka co-Z conjugate addition)

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\mathcal{P} = (X_1 : Y_1 : Z)$ and $\mathcal{Q} = (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_p, \mathcal{P}, \mathcal{Q} \in E_{/\mathbb{F}_p}$
Output: (X_3, Y_3) and (X_3', Y_3') s.t. $\mathcal{P} + \mathcal{Q} = (X_3 : Y_3 : Z_3)$ and $\mathcal{P} - \mathcal{Q} = (X_3' : Y_3' : Z_3)$ for some $Z_3 \in \mathbb{F}_p$

1: $A \leftarrow (X_2 - X_1)^2$
2: $B \leftarrow X_1 A$
3: $C \leftarrow X_2 A$
4: $D \leftarrow (Y_2 - Y_1)^2; F \leftarrow (Y_1 + Y_2)^2$
5: $E \leftarrow Y_1(C - B)$
6: $X_3 \leftarrow D - (B + C)$
7: $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$
8: $X_3' \leftarrow F - (B + C)$
9: $G \leftarrow Y_1 + Y_2; Y_3' \leftarrow G(X_3' - B) - E$
10: **return** $((X_3, Y_3), (X_3', Y_3'))$

Important remark 1 – Algorithms 1 and 2 do not actually involve the common z-coordinate of input points. The important thing is that input points share the same z-coordinate. Besides this, computations do not actually make use of the input z-coordinate, nor do they make determination of the output one.

Important remark 2 – Algorithms 1 and 2 are relevant only if both $\mathcal{P}$ and $\mathcal{Q}$ differ from $\mathcal{O}$. However the case where one of the point is null can be easily handled in an implementation by using a metadata boolean variable associated to each point, let's note this variable η, encoding the fact that it may equal $\mathcal{O}$: by definition $\eta(\mathcal{P}) = 1$ if $\mathcal{P} = \mathcal{O}$, and $\eta(\mathcal{P}) = 0$ if $\mathcal{P} \neq \mathcal{O}$. The implementation of ZADDU would then just have to check for the η metadata variable of each point before proceeding to apply co-Z formulæ. If for instance it happens that $\mathcal{P} \neq \mathcal{O}$ and $\mathcal{Q} = \mathcal{O}$, then computing ZADDU$(\mathcal{P}, \mathcal{Q})$ will consist simply in returning the couple of points $(\mathcal{P}, \mathcal{P})$ with variable η set to 0 for both points. Likewise, if $\mathcal{P} = \mathcal{O}$ and $\mathcal{Q} \neq \mathcal{O}$, then implementation of ZADDC$(\mathcal{P}, \mathcal{Q})$ will have to return the couple of points $(\mathcal{Q}, -\mathcal{Q})$ with η set to 0 for both points. As explained later (see §4) there are actually only two sensitive points involved in the IP to implement $[k]\mathcal{P}$ computation, which are named $\mathcal{R}_0$ and $\mathcal{R}_1$. For point $\mathcal{R}_0$, η is implemented as register `r.ctrl.r0z` in ⟨`ecc_scalar.vhd`⟩. Likewise, for point $\mathcal{R}_1$, η is implemented as register `r.ctrl.r1z`.

Important remark 3 – Algorithm 1 is relevant only if $\mathcal{P} \neq \mathcal{Q}$, and algorithm 2 is relevant only if $\mathcal{P} \neq \mathcal{Q}$ and $\mathcal{P} \neq -\mathcal{Q}$. But co-Z formulæ allow to easily detect and handle these situations:

- in the case of ZADDU, $\mathcal{P} = \mathcal{Q}$ occurs if and only if $X_1 = X_2$ and $Y_1 = Y_2$, that is if and only if $A = D = 0$. This has the consequence that all variable members inside ZADDU formulæ turn to 0, and thus also the results (X_3, Y_3) and (X_1', Y_1').
- in the case of ZADDC, $\mathcal{P} = -\mathcal{Q}$ occurs if and only if $X_1 = X_2$ and $Y_1 = -Y_2$, that is if and only if $A = G = 0$. This also has the consequence that all variable members inside ZADDC formulæ turn to 0, and thus also the results (X_3, Y_3) and (X_3', Y_3').

The IP detects the aforementioned situations, that we call **exceptions**, and handles them by calling a specific operation named ZDBL ([GJM+11]) [1]. The consequence however is the lost of two

[1] The operation we call ZDBL is actually called DBLU in [GJM+11].

important side-channel countermeasures for $[k]\mathcal{P}$ computation whenever such an exception is met: the lost of constant time feature and the lost of independence of the microcode execution flow as regards to the bits of the scalar. Please refer to §4.1.2 for more information on exceptions.

3.4 Montgomery representation

IPECC uses Montgomery representation [Mon85] to accelerate modular multiplication of numbers in $\mathbb{F}_p$.

3.4.1 Definition

Given a prime number p, let's call n the number of bits in the binary representation of p:

$$2^{n-1} \leq p < 2^n$$

We define:

$$R = 2^n$$

Since p is prime, R is invertible modulo p. We note $R^{-1} \mod p$.

The Montgomery transformation, noted ϕ, is the map from $\mathbb{F}_p$ to itself defined by:

$$\phi(x) = x \cdot R \mod p$$

This map (multiplication by R modulo p) establishes a 1-to-1 correspondence in $\mathbb{F}_p$, with the inverse map of ϕ obviously defined by:

$$\phi^{-1}(x) = x \cdot R^{-1} \mod p$$

$\phi(x)$ is also called the *Montgomery representation* of x. Theorem 1 below gives the Montgomery representation of a product of two numbers as a function of the Montgomery representation of each of them.

Theorem 1. *Let $z = x \cdot y$. Then:*

$$\phi(z) = \phi(x) \cdot \phi(y) \cdot R^{-1} \mod p$$

Proof.

$$\begin{aligned} \phi(x) \cdot \phi(y) \cdot R^{-1} \mod p &= x \cdot R \cdot y \cdot R \cdot R^{-1} \mod p \\ &= x \cdot y \cdot R \mod p \\ &= z \cdot R \mod p \\ &= \phi(z). \end{aligned}$$

□

Figure 3.1 provides an illustration of theorem 1.

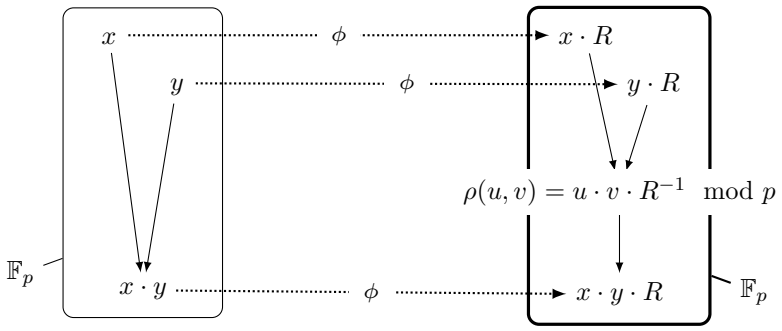


Figure 3.1: Montgomery transformation is a 1-to-1 correspondence inside $\mathbb{F}_p$

The advantage of using Montgomery representation is that computing

$$\rho(u,v) \;=\; u \;\cdot\; v \;\cdot\; R^{-1} \mod p$$

with $u = x \cdot R$ and $v = y \cdot R$ turns out to be a lot easier than multiplying x and y and then reducing the resulting product modulo p. Reducing modulo p requires to divide by p. On the other hand, applying ρ function on the Montgomery representatives $x \cdot R$ and $y \cdot R$ involves solely two multiplications followed by a division by $R = 2^n$. The latter is a particularly efficient operation as it simply requires a bit shift. This is explained in the two following sections §3.4.2 and §3.4.3.

3.4.2 Montgomery reduction

Montgomery reduction is an algorithm to compute $s/R \mod p = s \cdot R^{-1} \mod p$ for any number $s \in \mathbb{F}_p$ and $R = 2^n$. It is described in algorithm 3 below.

Algorithm 3 Montgomery reduction

Input: $s \in \mathbb{F}_p$
Output: $s \cdot R^{-1} \mod p$
1: $k \leftarrow s \times (-p^{-1} \mod R)$
2: $v \leftarrow s + k \cdot p$
3: $w \leftarrow v/R$
4: **if** $w \geq p$ **then**
5: $\quad w \leftarrow w - p$
6: **end if**
7: **return** w

The intuitive notion behind Montgomery multiplication is that thanks to p being a prime number, it is possible to add to any integer s an exact quantity of the number p so that the result, which is still obviously a modulo-p representation of s, is also an exact multiple of R, thus allowing the subsequent division by R to be computed very fast. Note that this implies leaving momentarily the set $\mathbb{F}_p$ and manipulating relative integers (numbers in $\mathbb{Z}$) instead. The quantity k is obtained by resolving the following equation, which also gives demonstation of algorithm 3's correctness:

$$\begin{aligned} & s + k \cdot p &&= 0 && \mod R \\ \Leftrightarrow \quad & k \cdot p &&= -s && \mod R \\ \Leftrightarrow \quad & k &&= s \times (-p^{-1}) && \mod R \end{aligned} \tag{3.8}$$

Relations (3.8) obviously require that p has an inverse modulo R, a property being guaranteed by the primality of p (Bezout identity dictates that if R has an inverse modulo p, then p also has an inverse modulo R).

A prerequisite to algorithm 3 is the computation of $-p^{-1} \mod R$.

☞ $-p^{-1} \mod R$ is called the *first Montgomery constant*.

Computation of $-p^{-1} \mod R$ usually involves a binary extended euclidian inversion, which is an expensive operation. But this computation has to be done only once and for all, and can be usually done offline, as p is a public fixed number.

3.4.3 Montgomery multiplication (REDC)

Montgomery multiplication simply consists in applying algorithm 3 to the product $s = u \cdot v$. This operation is described in algorithm 4 below. It is also sometimes called *REDC* operation.

Algorithm 4 Montgomery multiplication (aka REDC operation)

Input: $u, v \in \mathbb{F}_p$
Output: $u \cdot v \cdot R^{-1} \mod p$
1: $s \leftarrow u \cdot v$
2: $k \leftarrow s \times (-p^{-1} \mod R)$
3: $v \leftarrow s + k \cdot p$
4: $w \leftarrow v/R$
5: **if** $w \geq p$ **then**
6: $\quad w \leftarrow w - p$
7: **end if**
8: **return** w

Therefore Montgomery multiplication simply implements the function ρ that we've seen in §3.4.1.

3.4.4 Summary: modular multiplication based on Montgomery multiplication

Given the 1-to-1 correspondence defined by function ϕ, we can now consider performing modular multiplication between two numbers x and y in $\mathbb{F}_p$ by:

1. switching these numbers to their Montgomery representatives $x \cdot R$ and $y \cdot R$, using ϕ transformation function
2. use Montgomery multiplication algorithm so that to compute $\rho(xR, yR) = \phi(x \cdot y)$, as illustrated on figure 3.1
3. use ϕ^{-1} function to finally get $x \cdot y \mod p$.

Of course, one modular multiplication cannot afford the expensive transformations ϕ and ϕ^{-1}. But when it comes to perform several modular multiplications in a row, the cost of ϕ and ϕ^{-1} becomes relatively insignificant. That explains the powerful acceleration of Montgomery multiplication in the context of ECC. As we've seen in §3.2 and §3.3, point arithmetic on elliptic curves involves computation of several multiplications on coordinates in $\mathbb{F}_p$, along with additions and subtractions. To carry on modular multiplications efficiently, we simply need to switch the coordinates of basis-point $\mathcal{P}$ into their Montgomery representation, then use algorithm 4 as many times as necessary

(that is, as many times as we need to perform field multiplications) and then go back to normal representation of numbers using the inverse transformation ϕ^{-1}. How to efficiently implement ϕ and ϕ^{-1} transformations is explained in section 3.4.5 hereafter.

Note: additions and subtractions pose no particular challenges, since we obviously have:

$$\begin{aligned} & \phi(x+y) = (x+y)\cdot R \mod p = x\cdot R + y\cdot R \mod p = \phi(x)+\phi(y) \\ \text{and} \quad & \phi(x-y) = (x-y)\cdot R \mod p = x\cdot R - y\cdot R \mod p = \phi(x)-\phi(y). \end{aligned}$$

3.4.5 Switching in and out of Montgomery representation

Algorithm 4 also provides a very efficient means to compute the Montgomery representative of a number as well as its inverse.

To switch any number x to its Montgomery representative, we simply need to compute ρ on x and $R^2 \mod p$.

Proof.

$$\rho(x, R^2) = x\cdot R^2\cdot R^{-1} \mod p = x\cdot R \mod p = \phi(x).$$

□

☞ $R^2 \mod p$ is called the *second Montgomery constant* .

To retrieve any number x back from its Montgomery representative $x\cdot R$, we simply need to compute ρ on $x\cdot R$ and 1.

Proof.

$$\rho(x\cdot R, 1) = x\cdot R\cdot R^{-1} \mod p = x \mod p = \phi^{-1}(x\cdot R).$$

□

3.4.6 One last trick ($R > 4p$)

The reader may have noticed that a final subtraction $w-p$ can be present at the end of algorithm 4 (see §3.4.3) when $w \geq p$. The reason for this conditional subtraction is that computations performed in lines 1-4 yield a number w such that $0 \leq w \leq 2p$ (this is demonstrated in [Mon85]). Obtaining the true p-residue value of ouput $s\cdot R^{-1} \mod p$ therefore requires to test for condition $w \geq p$ and, if that condition is fulfilled, to subtract p.

This conditional substraction has the negative effect that it leaks information on the variables on which REDC operation is conducted. It also reduces performances, although this effect is quite small compared to the computational cost of multiplications.

A trick was introduced in [SV93] that allows not having to perform the conditional subtraction in algorithm 4 in the case where the three conditions $R > 4p$, $0 \leq u < 2p$ and $0 \leq v < 2p$ are met. It is then easy to verify that $0 \leq w < 2p$. In other words, the property for numbers of being in the range $[0, 2p[$ becomes an invariant of the algorithm. Therefore when $R > 4p$, each time we need to perform several consecutive REDC operations, and as long as all input numbers are smaller than $2p$ we can remove the conditional subtraction. This removes the side-channel leak as well as the computational cost, with the trade-off of requiring four extra bits in order to encode big numbers (more details on this are given in §D.1).

Chapter 4

Countermeasures against side-channel attacks

Table 4.1 below shows the side-channel countermeasures offered in IPECC.

Countermeasure	Biblio	Description	Refer to
Joye right-to-left *double-and-add-always* loop algorithm (this is a built-in countermeasure)	[Joy07]	Thwarts timing attacks (allows constant-time scalar loop iteration)	§4.1.1
Projective coordinate randomization (aka Z-masking, built-in)	[Cor99]	Thwarts differential attacks based on the hypothesis and monitoring of *values* of cryptographic variables implied in the computation of $[k]\mathcal{P}$	§4.2
Scalar blinding (this is an optional countermeasure)	[Cor99]	Thwarts attacks based on hypothesis and monitoring of *values* of cryptographic variables implied in the computation of $[k]\mathcal{P}$	§4.3
Anti address-bit DPA (aka ADPA, built-in)	[IIT03]	Thwarts attacks based on hypothesis and monitoring of *addresses* of cryptographic variables implied in the computation of $[k]\mathcal{P}$	§4.1.3
Complete memory shuffling (optional)		Thwarts attacks based on hypothesis and monitoring of *addresses* of cryptographic variables implied in the computation of $[k]\mathcal{P}$	§4.4
Verification that initial and final points belong to the curve (built-in)		Thwarts attacks using illegal points injection to obtain information on the scalar	§4.5
Complete independence of microcode execution flow from the scalar (built in)		The same microcode routine is executed for each bit of the scalar, with no dependency of the fetch address towards the value of the bit. The execution of this routine is purely sequential and involves no conditional jump whatsoever	§4.6

Table 4.1: Side-channel countermeasures provided with IPECC

Each of these countermeasures is described in detail in a proper section below (see right-most column of table 4.1. Joye scalar loop and ADPA are presented together because of their tight relation.

The reader who is already aware of the state-of-the-art in the matter of side-channel countermeasures can refer directly to §4.7 where the complete algorithm for $[k]\mathcal{P}$ computation is detailed with precision (this is algorithm 13) including main side-channel countermeasures. The aim of the present chapter anyway is to gradually introduce to algorithm 13 starting from the naive double-and-add versions of the scalar loop to the complete secured version implemented by the IP, going through the steps of each countermeasure and/or computation optimization one after the other, from §4.1

to §4.7.

4.1 Joye right-to-left *double-and-add-always* loop with ADPA

4.1.1 Double-and-add

Algorithms computing $[k]\mathcal{P}$ using point addition and doubling formulæ (see chapter 3) look much alike algorithms computing the exponentiation of a number using multiplication and squares. In the context of elliptic curves, we call *binary scalar multiplication*, or also *double-an-add loop*, algorithms which parse the binary representation of the scalar k so that to compute the scalar multiplication of a point on an elliptic curve. All of these algorithms fall into two categories, depending on the direction, *left-to-right* or *right-to-left*, in which they run over the bits of the binary representation of scalar k. Algorithm 5 (resp. 6) below gives the so-called *naive* version of the left-to-right (resp. right-to-left) binary scalar multiplication.

Algorithm 5 Left-to-right binary algorithm

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: $\mathcal{R}_0 \leftarrow \mathcal{P}, \mathcal{R}_1 \leftarrow \mathcal{P}$
2: **for** $i = n-2$ **downto** 0 **do**
3: $\quad \mathcal{R}_0 \leftarrow [2]\mathcal{R}_0$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad \mathcal{R}_0 \leftarrow \mathcal{R}_0 + \mathcal{R}_1$
6: $\quad$ **end if**
7: **end for**
8: **return** $\mathcal{R}_0$

Algorithm 6 Right-to-left binary algorithm

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: $\mathcal{R}_0 \leftarrow \mathcal{O}, \mathcal{R}_1 \leftarrow \mathcal{P}$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad \mathcal{R}_0 \leftarrow \mathcal{R}_0 + \mathcal{R}_1$
5: $\quad$ **end if**
6: $\quad \mathcal{R}_1 \leftarrow [2]\mathcal{R}_1$
7: **end for**
8: **return** $\mathcal{R}_0$

Algorithms 5 and 6 are called naive because the *if-then* branch inside each iteration of the loop allows an attacker to identify, using physical observation of the overall loop execution, which bits of the scalar are 0 and which are 1. The distinction can become very easy even on a unique observation trace, as the execution of the conditional addition $\mathcal{R}_0 + \mathcal{R}_1$ inside the loop iteration will yield a considerably different shape of the trace, both in time and amplitude.

In order to produce identical physical observation to a side-channel attacker regardless of the scalar bit values, regular algorithms are used instead, which are called *double-and-add-always* algorithms. Algorithm 7 below gives the regular (or double-and-add-always) version of the left-to-right loop. It is called *Montgomery ladder* and was first introduced in [Mon87]. Algorithm 8 gives the regular (or double-and-add-always) version of the right-to-left loop. It is called *Joye double-and-add* algorithm and was introduced in [Joy07].

Algorithm 7 Montgomery ladder

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: $\mathcal{R}_0 \leftarrow \mathcal{O}, \mathcal{R}_1 \leftarrow \mathcal{P}$
2: **for** $i = n-1$ **downto** 0 **do**
3: $\quad b \leftarrow k_i$; $\mathcal{R}_{1-b} \leftarrow \mathcal{R}_{1-b} + \mathcal{R}_b$
4: $\quad \mathcal{R}_b \leftarrow [2]\mathcal{R}_b$
5: **end for**
6: **return** $\mathcal{R}_0$

Algorithm 8 Joye double-and-add

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: $\mathcal{R}_0 \leftarrow \mathcal{O}, \mathcal{R}_1 \leftarrow \mathcal{P}$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad \mathcal{R}_{1-b} \leftarrow [2]\mathcal{R}_{1-b} + \mathcal{R}_b$
5: **end for**
6: **return** $\mathcal{R}_0$

Algorithm 7 maintains the invariant $\mathcal{R}_1 - \mathcal{R}_0 = \mathcal{P}$. Algorithm 8 maintains the invariant $\mathcal{R}_0 + \mathcal{R}_1 = [2^{i+1}]\mathcal{P}$.

4.1.2 Joye algorithm with co-Z arithmetic

We have seen in section 3.3 that ZADDU$(\mathcal{P}, \mathcal{Q})$ and ZADDC$(\mathcal{P}, \mathcal{Q})$ operations (as described by algorithm 1 and algorithm 2) cannot handle the situation where $\mathcal{P} = \mathcal{Q}$. We also mentioned that this situation could be handled by using a third operation named ZDBL. Now considering the Joye loop (algorithm 8) the situation $\mathcal{R}_0 = \mathcal{R}_1$ is subject to happen in two cases:

1. We are to meet $\mathcal{R}_0 = \mathcal{R}_1$ **at the begining** of algorithm 8 with a probability of 1, that is with certainty. Indeed, as the algorithm maintains the equality $\mathcal{R}_0 + \mathcal{R}_1 = [2^i]\mathcal{P}$ after each iteration of the scalar loop, the equality $\mathcal{R}_0 = \mathcal{R}_1$ is guaranteed to occur after the first iteration i_0 of the loop where $k_{i_0} = 1$. Since $\mathcal{R}_0$ is initialized to $\mathcal{O}$ (line 1), running the scalar loop from right to left will ensure that $\mathcal{R}_1 \leftarrow [2^i]\mathcal{P}$ after each iteration of the loop as long as only null bits are met in the scalar, while $\mathcal{R}_0$ will remain the $\mathcal{O}$ point. Then, upon hitting the first non-null bit of the scalar, line 4 of algorithm 8 will compute $\mathcal{R}_0 \leftarrow [2]\mathcal{O} + \mathcal{R}_1$, after what we are to reach the situation $\mathcal{R}_0 = \mathcal{R}_1 (= [2^i]P)$. This is illustrated on figure 4.1 below representing the set of all possible values of the pair of points $(\mathcal{R}_0, \mathcal{R}_1)$ for the first four iterations of algorithm 8 (some values have not been displayed on the figure for sake of clarity). The problematic cases where $\mathcal{R}_0 = \mathcal{R}_1$ have been framed inside the scalar decisional tree.

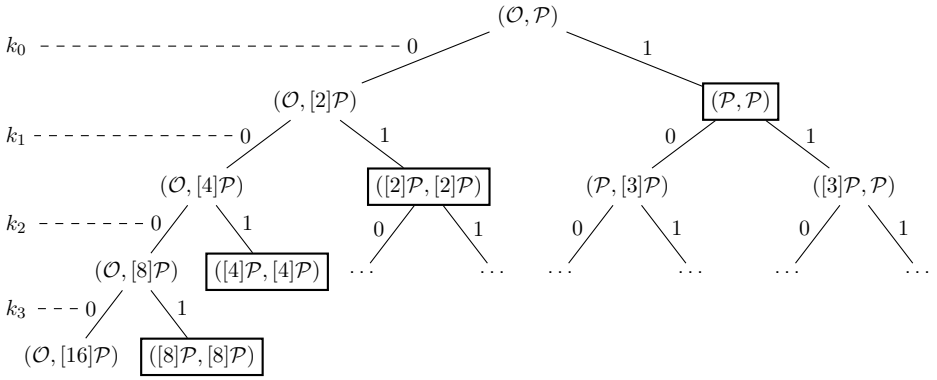


Figure 4.1: Due to initial conditions $(\mathcal{R}_0, \mathcal{R}_1) = (\mathcal{O}, \mathcal{P})$ (root of the tree), algorithm 8 is bound to meet condition $\mathcal{R}_0 = \mathcal{R}_1$ after the first non-null bit of the scalar starting from the right. For instance if the three least significant bits of the scalar are 100_2 (meaning $k_0 = 0$, $k_1 = 0$ and $k_2 = 1$) then we'll hit $\mathcal{R}_0 = \mathcal{R}_1 = [4]\mathcal{P}$ with certainty after processing of k_2.

2. From a purely theoretical point of vew, it is also possible to meet condition $\mathcal{R}_0 = \mathcal{R}_1$ **in the course** of algorithm 8 (i.e after any iteration of the loop) but given the size of cryptographic numbers, this can happen with so small a probability that we can consider this to be 0 (i.e impossible) since that event might never be encountered in the life-time of any implementation. Note that even if this event was to happen, the IP as it has been designed will handle the situation properly. We simply made up our mind to accept that whenever such an exception occurs, the side-channel resistance of $[k]\mathcal{P}$ will be lowered, as two important side-channel countermeasures will then be lost: the constant time execution, and the independence towards the scalar of the microcode execution flow..

The situation $\mathcal{R}_0 = \mathcal{R}_1$ certainly could be handled by using a special trick, for instance by putting aside co-Z arithmetic temporarily, using instead explicit formulæ for point-doubling, and then return

to co-Z arithmetic once the doubling is done. The drawback is that it would create an assymetry inside the scalar loop, leaking the number of least-significant bits of the scalar equaling 0. Assuming for instance a value of $k = \ldots 111000_2$, an attacker could easily identify on only one side-channel observation trace that $k_2 = k_1 = k_0 = 0$. Such a leak does not seem acceptable.

We elected to overcome that pitfall by always forcing to 1 the least-significant bit of the scalar, and then afterwards, once the entire scalar loop is completed, by conditionally subtracting the basis point $\mathcal{P}$ to the result in case the scalar was an even number to begin with. It is easy to ensure that the hardware produces the same side-channel signature whether or not that final subtraction actually happens (or at least to produce a less-than-observable side-channel signal difference between the two situations). Especially can the execution time be kept *exactly* the same.

If we now modify algorithm 8 so that it takes into account the «always $k_0 = 1$» trick that we just introduced, we obtain algorithm 9. Note that initial conditions are now $\mathcal{R}_0 = \mathcal{R}_1 = \mathcal{P}$ (line 1). Also note that the loop now starts at $i = 1$ instead of $i = 0$.

IPECC uses algorithm 9 modified so that to account for the risk of side-channel address-bit DPA attacks, which is explained in next section (§4.1.3).

Algorithm 9 Joye double-and-add loop ensuring that condition $\mathcal{R}_0 = \mathcal{R}_1$ is never met

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \ldots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$
1: $\mathcal{R}_0 \leftarrow \mathcal{P}, \mathcal{R}_1 \leftarrow \mathcal{P}$
2: **for** $i = 1$ **to** $n - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad \mathcal{R}_{1-b} \leftarrow [2]\mathcal{R}_{1-b} + \mathcal{R}_b$
5: **end for**
6: **if** $k_0 = 0$ **then**
7: $\quad \mathcal{R}_0 \leftarrow \mathcal{R}_0 - \mathcal{P}$
8: **end if**
9: **return** $\mathcal{R}_0$

4.1.3 Anti address-bit DPA (ADPA)

Algorithms 7 to 9 only provide constant time execution of the double-and-add loop. They still exhibit physical leakage in that they are vulnerable to physical observation and exploitation of the address of the registers $\mathcal{R}_0$ and $\mathcal{R}_1$ which are manipulated inside each iteration of the loop. In the case of Joye loop, depending on the value of k_i bit, the hardware is bound to compute either $\mathcal{R}_1 \leftarrow 2\mathcal{R}_1 + \mathcal{R}_0$ if $k_i = 0$, or $\mathcal{R}_0 \leftarrow 2\mathcal{R}_0 + \mathcal{R}_1$ if $k_i = 1$. If the attacker is able to distinguish the two operations, for instance by differentiating which register among $\mathcal{R}_0$ and $\mathcal{R}_1$ is being written for a particular iteration i of the loop, then she can directly infer the value of bit k_i. This kind of side-channel attack is named ADPA, which stands for *address-bit DPA*.

> The concept of an address of a variable inside a circuit may not be as clear as the one that is used in an algorithm or a piece of software. A variable inside a circuit is less local than it is in software, as it may be represented by both combinational signals (the ones connecting logic gates together) and/or sequential cells (flip-flops and memories). In the context of present discussion, we consider an address of a variable inside a circuit to be any physical signal that leaks the fact that a specific value of a digital number, at some time, is transferred into a sequential cell[1] rather than into any other element of the set of all possible cells where it could have been transferred into. Of course, what is interesting to the attacker are the addresses of variables that depend on the value of the scalar. For example, if during

[1] This could be a set of D-flip-flops or a set of SRAM cells inside an embedded on-chip memory array

one particular clock cycle a digital value – let's say a number made of n bits –, has been latched into a set (a) of D-flip-flops $\{d^a_{n-1}, \ldots, d^a_1, d^a_0\}$ rather than into another set (b) $\{d^b_{n-1}, \ldots, d^b_1, d^b_0\}$ which was *a priori* possible to the knowledge of the attacker, and if this event is linked to the scalar (e.g to one bit k_i) then any physical signal leaking that information may be the target of an ADPA attack. One can think of the clock-enable signal triggering the write operation into the set of target flip-flops as an example of an «address». Perhaps a more obvious example is the physical signal driving the address port of an SRAM block inside the logic fabric of an FPGA. Eventually what is really an address signal for us actually depends on the way the hardware design was created during the logic synthesis step of the design process. Be that as it may, it is always a bias in the physical floorplan of the circuit that creates a bias among the paths followed by some signal values known to the attacker and which may be linked to the scalar. This has to do with the space location of logic cells inside the circuit and the physical paths followed by the nets connecting them.

ADPA hence exploits the fact that not only the data signals inside hardware may leak bits of the scalar over physical channels (power consumption or EM field) but also their location at the surface of the chip. Electromagnetic waves induced by switching logic gates inside the circuit may influence the overall signal collected by an attacker at an EM probe with a huge dependance on their relative location to the probe. At any given point in space and time, the resulting EM field is obviously the linear addition of all the waves that are propagating in that point at that time, a direct consequence of the linearity of Maxwell Laws. Furthermore, depending on the geometry of the electrical conductors carrying electrical charges in the circuit, the intensity of the $\vec{B}$ field will vary in $1/r^d$, with $d > 1$, r denoting the distance to the surface of the chip. A rule of thumb in near-field analysis is that the signal decreases in $1/r^3$ in the range of 0 to approximately twice the wavelength. Such a changing slope creates a dramatic fall of the field with distance and thus a strong locality effect. If the attacker places her probe close enough to the circuit, she might therefore be able to discriminate between very specific events, for instance a write inside an FPGA's BlockRAM occuring at some address rather than another (even if she can't even so determine the exact complete address, as the exact complete address is not required for her in order to get a critically exploitable information[2]). As a conclusion, algorithm 8, although providing resistance against timing analysis, turns out to be vulnerable to power analysis, even possibly in a one trace scenario. We're therefore going to present how one can amend the scalar loop to thwart ADPA. Although we're mostly interested in the right-to-left version for IPECC, for sake of generality we'll introduce the ADPA countermeasure for both cases. We will thus obtain algorithm 10 as the refined anti-ADPA version of algorithm 7 and algorithm 11 as the refined anti-ADPA version of algorithm 8. We'll start discussing the left-to-right case.

Left-to-right case: an anti-ADPA countermeasure is presented in [IIT03] for the left-to-right versions of the double-and-add-always loop, in particular for the Montgomery ladder (algorithm 7). It uses a third point-register $\mathcal{S}$ and a random masking of the scalar, and modifies the computation inside each iteration of the scalar loop so that to balance the address at which both registers $\mathcal{R}_0$ and $\mathcal{R}_1$ are read and/or written. The Montgomery ladder modified for anti-ADPA is described in algorithm 10 below.

Algorithm 10 can be derived from algorithm 7 by maintaining the same invariant ($\mathcal{R}_1 - \mathcal{R}_0 = \mathcal{P}$) and ensuring that after each iteration i of the loop, the current «valid»[3] result is randomly stored in either $\mathcal{R}_0$ or $\mathcal{R}_1$ (it is the meaning of ϕ_i that it holds the address, 0 or 1, of the current valid result). The invariant becomes $\mathcal{R}_1 - \mathcal{R}_0 = [(-1)^{\phi_i}]\mathcal{P}$.

[2]For instance, during execution of algorithm 8, all the attacker needs to find is a place of her probe where the EM influence of writing the coordinates of point $\mathcal{R}_0$ will result in a significantly stronger signal than the EM influence of writing the coordinates of point $\mathcal{R}_1$ (or vice-versa).

[3]By «valid» we mean the intermediate point that is obtained, in a right-to-left implementation, after the last time that a non-null bit was met in the scalar. For instance if the ten least significant bits of the scalar are 0010000101_2, then the «valid» result will be $[5]\mathcal{P}$ from bits k_2 to k_6, and it'll become $[133]\mathcal{P}$ after processing of bit k_7.

Algorithm 10 Montgomery ladder with anti-ADPA countermeasure [IIT03]

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1} = 1, k_{n-2}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: **Compute** κ **and** κ', **two masked versions of** k:
2: $\phi \leftarrow (\phi_{n-1}, \phi_{n-2}, \dots, \phi_0)_2$ random number $\in \mathbb{N}$
3: $\phi' \leftarrow (0, \phi_{n-1}, \dots, \phi_2, \phi_1)_2 = \phi/2 = \phi$ right-shifted by one bit position
4: $\kappa \leftarrow k \oplus \phi$
5: $\kappa' \leftarrow k \oplus \phi'$
6: **Perform the scalar parsing-loop:**
7: $\mathcal{R}_{\phi_{n-1}} \leftarrow \mathcal{P}$
8: $\mathcal{R}_{1-\phi_{n-1}} \leftarrow [2]\mathcal{R}_{\phi_{n-1}}$
9: **for** $i = n-2$ **downto** 0 **do**
10: $\mathcal{S} \leftarrow [2]\mathcal{R}_{\kappa'_i}$
11: $\mathcal{R}_{1-\kappa_i} \leftarrow \mathcal{R}_0 + \mathcal{R}_1$
12: $\mathcal{R}_{\kappa_i} \leftarrow \mathcal{S}$
13: **end for**
14: **return** $\mathcal{R}_{\phi_0}$

Right-to-left case: now for IPECC, we've adapted the anti-ADPA countermeasure of algorithm 10 so that it fits the Joye right-to-left version of the scalar loop (algorithm 8) and obtained algorithm 11 below. This algorithm maintains the same invariant as the Joye loop ($\mathcal{R}_0 + \mathcal{R}_1 = [2^i]\mathcal{P}$) except that after each iteration i, the «valid» result is now randomly stored in $\mathcal{R}_{\phi_i}$, same as in algorithm 10.

Algorithm 11 Joye double-and-add loop with anti-ADPA countermeasure

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

1: **Compute** κ **and** κ', **two masked versions of** k:
2: $\phi \leftarrow (\phi_{n-1}, \phi_{n-2}, \dots, \phi_0)_2$ random number $\in \mathbb{N}$
3: $\phi' \leftarrow (\phi_{n-2}, \dots, \phi_1, \phi_0, 0)_2 = 2 \times (\phi \mod 2^{n-1}) = \phi$ left-shifted by one bit position
4: $\kappa \leftarrow k \oplus \phi$
5: $\kappa' \leftarrow k \oplus \phi'$
6: **Perform the scalar parsing-loop:**
7: $\mathcal{R}_0 \leftarrow \mathcal{O}$, $\mathcal{R}_1 \leftarrow \mathcal{P}$
8: **for** $i = 0$ **to** $n-1$ **do**
9: $\mathcal{S} \leftarrow \mathcal{R}_{\kappa'_i}$
10: $\mathcal{R}_{1-\kappa_i} \leftarrow [2]\mathcal{R}_{1-\kappa'_i} + \mathcal{R}_{\kappa'_i}$
11: $\mathcal{R}_{\kappa_i} \leftarrow \mathcal{S}$
12: **end for**
13: **return** $\mathcal{R}_{\phi_{n-1}}$

Table 4.2 below shows how ADPA countermeasure in algorithm 11 allows balancing addresses at which values of point registers $\mathcal{R}_0$ and $\mathcal{R}_1$ are actually read and written. Line 4 in algorithm 9 (or, for that matters, line 4 in algorithm 8) is now replaced with the three lines 9-11 of algorithm 11, where:

- line 9 may be equally duplicated into one of the two assertions: $\mathcal{S} \leftarrow \mathcal{R}_0$ or $\mathcal{S} \leftarrow \mathcal{R}_1$, depending on the random value of ϕ_i

- line 10 may be equally duplicated into one of the four assertions: $\mathcal{R}_1 \leftarrow [2]\mathcal{R}_1 + \mathcal{R}_0$, $\mathcal{R}_1 \leftarrow [2]\mathcal{R}_0 + \mathcal{R}_1$, $\mathcal{R}_0 \leftarrow [2]\mathcal{R}_1 + \mathcal{R}_0$ or $\mathcal{R}_0 \leftarrow [2]\mathcal{R}_0 + \mathcal{R}_1$, depending on the random values of ϕ_i and ϕ'_i
- line 11 may be equally duplicated into one of the two assertions: $\mathcal{R}_0 \leftarrow \mathcal{S}$ or $\mathcal{R}_1 \leftarrow \mathcal{S}$, depending on the random value of ϕ'_i.

κ_i \ κ'_i	0	1
0	$\mathcal{S} \leftarrow \mathcal{R}_0$ $\mathcal{R}_1 \leftarrow [2]\mathcal{R}_1 + \mathcal{R}_0$ $\mathcal{R}_0 \leftarrow \mathcal{S}$	$\mathcal{S} \leftarrow \mathcal{R}_1$ $\mathcal{R}_1 \leftarrow [2]\mathcal{R}_0 + \mathcal{R}_1$ $\mathcal{R}_0 \leftarrow \mathcal{S}$
1	$\mathcal{S} \leftarrow \mathcal{R}_0$ $\mathcal{R}_0 \leftarrow [2]\mathcal{R}_1 + \mathcal{R}_0$ $\mathcal{R}_1 \leftarrow \mathcal{S}$	$\mathcal{S} \leftarrow \mathcal{R}_1$ $\mathcal{R}_0 \leftarrow [2]\mathcal{R}_0 + \mathcal{R}_1$ $\mathcal{R}_1 \leftarrow \mathcal{S}$

Table 4.2: ADPA removes biases on the address at which point registers $\mathcal{R}_0$ and $\mathcal{R}_1$ are read and written.

To account for the necessity: (1) not to ever hit condition $\mathcal{R}_0 = \mathcal{R}_1$ (as previously explained in §4.1.2, see algorithm 9) together with: (2) the ADPA countermeasure, we finaly obtain algorithm 12 below.

Algorithm 12 Joye double-and-add loop with anti-ADPA and ensuring $\mathcal{R}_0 \neq \mathcal{R}_1$ always

Input: $\mathcal{P} \in E_{/\mathbb{F}_p}, k = (k_{n-1}, \dots k_1, k_0)_2 \in \mathbb{N}$
Output: $\mathcal{Q} = [k]\mathcal{P}$

Compute κ and κ', two masked versions of k:
  $\phi \leftarrow (\phi_{n-1}, \phi_{n-2}, \dots, \phi_0)_2$ random number $\in \mathbb{N}$
  $\phi' \leftarrow (\phi_{n-2}, \dots, \phi_1, \phi_0, 0)_2 = 2 \times (\phi \mod 2^{n-1}) = \phi$ left-shifted by one bit position
  $\kappa \leftarrow k \oplus \phi$
  $\kappa' \leftarrow k \oplus \phi'$
Perform the scalar parsing-loop:
  $\mathcal{R}_{\kappa_1} \leftarrow \mathcal{P}$
  $\mathcal{R}_{1-\kappa_1} \leftarrow [3]\mathcal{P}$
  for $i = 2$ **to** $n-1$ **do**
    $\mathcal{S} \leftarrow \mathcal{R}_{\kappa'_i}$
    $\mathcal{R}_{1-\kappa_i} \leftarrow [2]\mathcal{R}_{1-\kappa'_i} + \mathcal{R}_{\kappa'_i}$
    $\mathcal{R}_{\kappa_i} \leftarrow \mathcal{S}$
  end for
Conditionaly subtract $\mathcal{P}$:
  if $k_0 = 0$ **then**
    $\mathcal{R}_{\phi_{n-1}} \leftarrow \mathcal{R}_{\phi_{n-1}} - \mathcal{P}$
  end if
return $\mathcal{R}_{\phi_{n-1}}$

Note that ϕ_0 random bit is of no actual use in algorithm 12 and can be ignored. This reflects the fact that the «always $k_0 = 1$» trick, as seen in §4.1.2, leads to bypass the scalar loop iteration corresponding to $i = 0$. Furthermore, forcing an odd value for the scalar k, as it implies that we would have $\mathcal{R}_0 = \mathcal{R}_1 = \mathcal{P}$ after iteration $i = 0$, also implies that after iteration $i = 1$, the execution

would be in either of two situations: $\mathcal{R}_0 = \mathcal{P}$ (in which case $\mathcal{R}_1 = [3]\mathcal{P}$) or $\mathcal{R}_0 = [3]\mathcal{P}$ (in which case $\mathcal{R}_1 = \mathcal{P}$). This can be easily seen on figure 4.1 (see right part of the tree). The random bit ϕ_1 is used to set up the initial condition of the loop by randomly selecting between the two possibilities (this appears through the use of κ_1 on lines 7 and 8 of algorithm 12) and explains why the scalar loop now starts at $i = 2$, as compared to algorithm 9 which had this loop starting at $i = 1$.

4.2 Projective coordinate randomization

This countermeasure is built-in inside IPECC. It was introduced in [Cor99]. As we've seen in §3.2 the projective coordinates of a point are not unique because:

$$(X : Y : Z) = \{(\lambda^2 X, \lambda^3 Y, \lambda Z), \lambda \in \mathbb{F}_q^*\}.$$

for any $\lambda \neq 0$ in the field $\mathbb{F}_p$.

This is the mathematical property on which is based the side-channel countermeasure called *randomization of projective coordinates*. A random number $\lambda \in \mathbb{F}_p$ is drawn at the beginning of each new scalar multiplication and is used as the z-coordinate of the basis point $\mathcal{P}$. Coordinates x and y are then recomputed using $x \mapsto \lambda^2 x$ and $y \mapsto \lambda^3 y$ respectively. Side-channel attacks using the known values of coordinates of $\mathcal{P}$ therefore become inefficient.

4.3 Scalar blinding

Scalar blinding countermeasure was also introduced in [Cor99]. It is based on the property that an elliptic curve defined over a finite field has the structure of a cyclic group. Therefore if we note q the number of points on a curve $E_{/\mathbb{F}_p}$ (called the *order* of the curve) then $[q]\mathcal{P} = \mathcal{O}$ for any point $\mathcal{P} \in E_{/\mathbb{F}_p}$. From this we can derive the following countermeasure:

1. Before any new scalar multiplication $[k]\mathcal{P}$, we draw a random number $\alpha \in \mathbb{N}$
2. We compute $k' = k + \alpha q$
3. We then proceed to compute $[k']\mathcal{P}$.

We have: $[k']\mathcal{P} = [k]\mathcal{P} + [\alpha q]\mathcal{P} = [k]\mathcal{P} + [\alpha]([q]\mathcal{P}) = [k]\mathcal{P}$ since $[q]\mathcal{P} = \mathcal{O}$. Therefore using the blinded scalar k' in place of k yields the expected result.

This countermeasure is optional in IPECC. It can be engaged or disengaged on a $[k]\mathcal{P}$-computation per $[k]\mathcal{P}$-computation basis. The user (software driver) simply needs to inform the IP of the size of the blinding scalar α, which is application dependent.

4.4 Complete memory shuffling

This countermeasure is also optional. Complete memory shuffling is used to randomize, before each iteration of the scalar loop, the address at which temporary variables involved in the curve and point arithmetics are stored in the IP internal memory of large numbers. Before each new loop iteration, a random permutation of the space address of the memory array is drawn, and applied to its content. This countermeasure thwarts the attacks aimed at guessing which intermediate variables are manipulated inside curve formulæ, by using their address. As memory shuffling suppresses the relation between addresses and values of the aforementioned variables, these attacks becomes infeasible.

4.5 Point verification

This is a built-in countermeasure. Before each new scalar multiplication, the curve equation

$$y^2 = x^3 + ax + b$$

is checked against the coordinates of the basis point $\mathcal{P}$ so that to verify that $\mathcal{P}$ actually belongs on the curve. Scalar multiplication is started only if this test is sucessful. Similarly, once the scalar multiplication is complete, the same equation is checked against coordinates of point $[k]\mathcal{P}$ to check that no perturbation or tampering was tried with the hardware during computation, that might lead to an invalid output point. If the test is not successful, IPECC won't yield any output to software.

4.6 Independent microcode execution flow

(Redaction in progress, sorry)

4.7 Complete algorithm for scalar multiplication

The overall algorithm used in IPECC for $[k]\mathcal{P}$ computation is given below (algorithm 13).

☞ It is strongly advised to acquire a thorough understanding of algorithm 13 as it precisely describes (although at a high-level of description) the computation of the scalar multiplication performed by the IP. Algorithm 13 will be constantly referred to in the remainder of this document.

Algorithm 13 $[k]\mathcal{P}$ computation in IPECC

Input:
- prime number p $(2^{n-1} \leq p < 2^n), (a, b) \in {\mathbb{F}_p}^2$ s.t $4a^3 + 27b^2 \neq 0 \mod p$
- $\mathcal{P} = (x_P, y_P) \in E_{(a,b)/\mathbb{F}_p}, k = (k_{n-1}, \ldots k_1, k_0)_2 \in \mathbb{N}$
- optionally for blinding: n_α=bitwidth of blinding scalar α, q=order of $E_{(a,b)/\mathbb{F}_p}$

Output: $\mathcal{Q} = [k]\mathcal{P}$

1: **if** ${y_P}^2 \neq {x_P}^3 + ax_P + b$ **then return error**
2: **If blinding is active:**
3: $\alpha \leftarrow$ **random number** $\in \mathbb{N}$ **of size** n_α **bits**
4: $k \leftarrow k + \alpha q$
5: $n' \leftarrow n + n_\alpha$
6: **else:** $n' \leftarrow n$
7: **Compute** κ **and** κ'**, two masked versions of** k**:**
8: $\phi \leftarrow (\phi_{n'-1}, \phi_{n'-2}, \ldots, \phi_0)_2$ **random number** $\in \mathbb{N}$
9: $\phi' \leftarrow (\phi_{n'-2}, \ldots, \phi_1, \phi_0, 0)_2 = 2 \times (\phi \mod 2^{n'-1})$
10: $\kappa \leftarrow k \oplus \phi$
11: $\kappa' \leftarrow k \oplus \phi'$
12: **Switch to projective coordinates and enter Montgomery representation:**
13: $x_P \leftarrow x_P \cdot R$
14: $y_P \leftarrow y_P \cdot R$
15: $z_P \leftarrow R$
16: **Randomize projective coordinates:**
17: $\lambda \leftarrow$ **random number** $\in \mathbb{F}_p$ **of size** n **bits**
18: $z_P \leftarrow z_P \cdot \lambda$
19: $x_P \leftarrow x_P \cdot \lambda^2$
20: $y_P \leftarrow y_P \cdot \lambda^3$
21: **Perform the scalar parsing-loop:**
22: $\mathcal{R}_{\kappa_1} \leftarrow \mathcal{P}$
23: $\mathcal{R}_{1-\kappa_1} \leftarrow [3]\mathcal{P}$
24: **for** $i = 2$ **to** $n' - 1$ **do**
25: **if big-number memory shuffling is active then do shuffle**
26: $\mathcal{S} \leftarrow \mathcal{R}_{\kappa'_i}$
27: $\mathcal{R}_{1-\kappa_i} \leftarrow [2]\mathcal{R}_{1-\kappa'_i} + \mathcal{R}_{\kappa'_i}$
28: $\mathcal{R}_{\kappa_i} \leftarrow \mathcal{S}$
29: **end for**
30: $\mathcal{Q} \leftarrow \mathcal{R}_{\phi_{n-1}}$
31: **Conditionaly subtract** $\mathcal{P}$ **(in constant time):**
32: **if** $k_0 = 0$ **then**
33: $\mathcal{Q} \leftarrow \mathcal{Q} - \mathcal{P}$
34: **end if**
35: **Switch from projective coordinates back to affine:**
36: $Z \leftarrow {Z_\mathcal{Q}}^{-1}$
37: $x_\mathcal{Q} \leftarrow X_\mathcal{Q} \cdot Z^2$
38: $y_\mathcal{Q} \leftarrow Y_\mathcal{Q} \cdot Z^3$
39: **Leave Montgomery representation:**
40: $x_Q \leftarrow x_Q \cdot R^{-1}$
41: $y_Q \leftarrow y_Q \cdot R^{-1}$
42: **if** ${y_Q}^2 \neq {x_Q}^3 + ax_Q + b$ **then return error else return** $\mathcal{Q} = (x_Q, y_Q)$

Chapter 5

Software programming

5.1 Debug mode vs production mode

IPECC can be used in two different modes which are exclusive of one another: either the IP is configured in **production mode** or it is configured in **debug mode.** This configuration is static and can't be modified at runtime. The choice between the two options is made in the source file ⟨`ecc_customize.vhd`⟩, by setting parameter `debug` either to `TRUE` (IPECC is then in debug mode) or to `FALSE` (IPECC is then in production mode).

5.1.1 Debug mode

Debug mode means that interacting with the IP through software is made very permissive, so as to allow pre-production analysis of the IP. Thus software can tamper freely with almost any security feature implemented by the IP.

In debug mode, software can:

- read or write the value of any large number memory, at any time[1]
- insert breakpoints into microcode to interrupt scalar multiplication and perform step by step execution (e.g to allow time isolation of a specific part of $[k]\mathcal{P}$ computation for clean, reduced-noise side-channel measurement)
- specify a precise time in the course of $[k]\mathcal{P}$ computation where to activate or deactivate the trigger signal driven outside of the IP. Use cases for the external out trigger feature include oscilloscope input-trigger activation and EM injection probe setting-off. This feature is clock-cycle accurate
- bypass the TRNG and force use of deterministic numbers instead of random ones
- access internal memory of TRNG in order to read raw random numbers generated by the physical source of entropy as well as the internal (post-processed) numbers, and this way perform entropy quality assessment
- modify the content of the microcode memory to implement and evaluate different curve formulæ or different countermeasures.

[1] Note however that when a computation is pending in the IP (either computation of Montgomery constants, $[k]\mathcal{P}$ computation or any other point operation) access to large number memory requires the IP to be first halted, e.g on a breakpoint hit or by using register `W_DBG_HALT`.

5.1.2 Production mode

As the name suggests, production mode is aimed at targeting the final version of the IP in the case you intend to design a product. In production mode, all debug features described thereabove in §5.1.1 are disengaged. More precisely, in production mode:

- the only large numbers software is allowed to write in memory are the first eight ones (large number address 0 to 7). These are: prime number p, curve parameters a, b and q, scalar k, and affine coordinates of points $\mathcal{R}_0$ and $\mathcal{R}_1$ (address 4 is overloaded as it's used both for scalar k and for the x-coordinate of $\mathcal{R}_0$). Note that for scalar multiplication, point $\mathcal{R}_1$ is the one that software must use as the base point $\mathcal{P}$
- the only large numbers software is allowed to read from memory are $x_{\mathcal{R}_1}$ and $y_{\mathcal{R}_1}$ where are stored the affine coordinates of the result of all point operations, including $[k]\mathcal{P}$
- breakpoints do not exist (control logic is pruned at synthesis time)
- outside trigger does not exit (control logic is pruned at synthesis time)
- TRNG cannot be bypassed, nor random values be forced, nor random values be read by software (control logic and read data paths are pruned at synthesis time)
- microcode memory cannot be modified (content is fixed at synthesis time and write-data path to memory is pruned)
- interactions between software and IP are strongly restricted. Whenever software issues a command that requires computation from the IP, it can no longer issues any command before that computation is carried out and completed.

  These computation phases are fourfold:

  1. **Computation of Montgomery constants** which automatically takes place each time software writes a new value of large number p

     Bit `STATUS_MTY` in `STATUS` register, when asserted along with `STATUS_BUSY` bit, indicates that computation of Montgomery constants is currently pending and software shouldn't perform any AXI operation on the IP but to poll `STATUS` register until it shows a `STATUS_BUSY` bit deasserted.

  2. **Computation of internal hardware signals related to the size `nn` of prime number** p, which takes place each time a new value of `nn` has been set dynamically (i.e at runtime) by software. This is only possible when the feature has been activated statically (i.e at synthesis time) by setting the constant `nn_dynamic` to TRUE in ⟨`ecc_customize.vhd`⟩.

     Bit `STATUS_NNDYN_ACT` in `STATUS` register, when asserted along with `BUSY` bit, indicates that computation of internal hardware signals related to `nn` is currently pending and software shouldn't perform any AXI operation on the IP but to poll `STATUS` register until it shows a `STATUS_BUSY` bit deasserted.

  3. **Scalar multiplication**

     Bit `STATUS_KP` in `STATUS` register, when asserted along with `STATUS_BUSY` bit, indicates that scalar multiplication is currently pending and software shouldn't perform any AXI operation on the IP but to poll `STATUS` register until it shows a `STATUS_BUSY` bit deasserted.

  4. **Point operation (other that $[k]\mathcal{P}$) or test**

     Bit `STATUS_POP` in `STATUS` register, when asserted along with `STATUS_BUSY` bit, indicates that a point operation/test is currently pending and software shouldn't perform any AXI operation on the IP but to poll `STATUS` register until it shows a `STATUS_BUSY` bit deasserted.

Whenever one of the four previously mentioned computations is pending, software can no longer issues any command on the AXI bus but polling the `STATUS_BUSY` bit from the `STATUS` register (or wait for the raise of IRQ by the IP if this was requested by software) until corresponding operation is over and IP becomes available again. Any AXI transaction issued by software during these computation phases will be simply disgarded by the IP, except for read of the `STATUS` register. This means:

- IP will simply acknowledge write transactions, however discard data transmitted on the `WDATA` bus of the AXI write data channel
- IP will acknowledge read transactions by driving the `RDATA` bus of the AXI read data channel with all bits asserted high.

Note: Four bits `STATUS_MTY`, `STATUS_NNDYN_ACT`, `STATUS_KP` and `STATUS_POP` of `STATUS` register simply give a secondary information on the type of computation currently taking place inside the IP when `STATUS_BUSY` bit is also asserted. Software should rely only on `STATUS_BUSY` bit to determine if pending operation by IP is still in progress or if it's over.

5.2 Instanciating the IP in your design

Instanciating the IP in your hardware design requires that you first statically configure it. This is achieved very simply by editing a few parameters defined in the source file ⟨`ecc_customize.vhd`⟩.

```
-- from hdl/ecc_customize.vhd:
 20 package ecc_customize is
 21   -- ***********************************
 22   -- Start of: user-editable parameters
 23   -- ***********************************
 24   -- Please refer to the in-file documentation of parameters below (after the
 25   -- package specification), where parameters are described in the same order
 26   -- as they appear hereafter.
 27   constant nn : positive := 528;
 28   constant nn_dynamic : boolean := TRUE;
 29   type techno_type is (spartan6, virtex6, series7, ultrascale, ialtera, asic);
 30   constant techno : techno_type := series7; -- set a 'techno_type' value
 31   -- ------------------------------
 32   -- Performance related parameters
 33   -- ------------------------------
 34   -- multwidth is only used if 'techno' = 'asic'
 35   -- (otherwise its value has no meaning and can be ignored)
 36   constant multwidth : positive := 32; -- 32 seems fair for an ASIC default
 37   constant nbmult : positive range 1 to 2 := 2;
 38   -- parameter nbdsp below is range-constrained because it must be >=2
 39   constant nbdsp : positive range 2 to positive'high := 6;
 40   constant sramlat : positive range 1 to 2 := 1;
 41   constant async : boolean := TRUE;
 42   -- ------------------------------------------------
 43   -- Side-channel countermeasures related parameters
 44   -- ------------------------------------------------
 45   constant debug : boolean := FALSE; -- FALSE = highly secure, TRUE = highly not
 46   constant blinding : integer := 96; -- 96 seems fair for size of blinding rnd
 47   constant shuffle : boolean := TRUE; -- memory shuffling
 48   type shuftype is (none, linear, permute_lgnb, permute_limbs);
 49   constant shuffle_type : shuftype := permute_lgnb; -- set a 'shuftype' value
 50   constant zremask : integer := 4; -- quite arbitrary
 51   -- ------------------------
 52   -- TRNG related parameters
 53   -- ------------------------
 54   constant notrng : boolean := FALSE; -- set to TRUE for simu, to FALSE for syn
 55   constant nbtrng : positive := 4;
 56   constant trngta : natural range 1 to 4095 := 32;
 57   constant trng_ramsz_raw : positive := 4; -- in kB
 58   constant trng_ramsz_axi : positive := 4; -- in kB
 59   constant trng_ramsz_fpr : positive := 4; -- in kB
 60   constant trng_ramsz_crv : positive := 4; -- in kB
```

```
61   constant trng_ramsz_shf : positive := 16; -- in kB
62   -- -------------
63   -- Miscellaneous
64   -- -------------
65   constant axi32or64 : natural := 32; -- 32 or 64 only allowed values
66   constant nblargenb : positive := 32;  -- Change these two parameters only if
67   constant nbopcodes : positive := 512; -- you really know what you're doing.
68   -- --------------------------
69   -- Simulation-only parameters
70   -- --------------------------
71   constant simvecfile : string := "/tmp/ecc_vec_in.txt";
72   constant simkb : natural range 0 to natural'high := 0; -- if 0 then ignored
73   constant simlogfile : string := "/tmp/ecc.log";
74   constant simtrngfile : string := "/tmp/random.txt";
75   -- ********************************
76   -- End of: user-editable parameters
77   -- ********************************
78 end package;
...
```

Please refer to Appendix A for complete description of the different parameters.

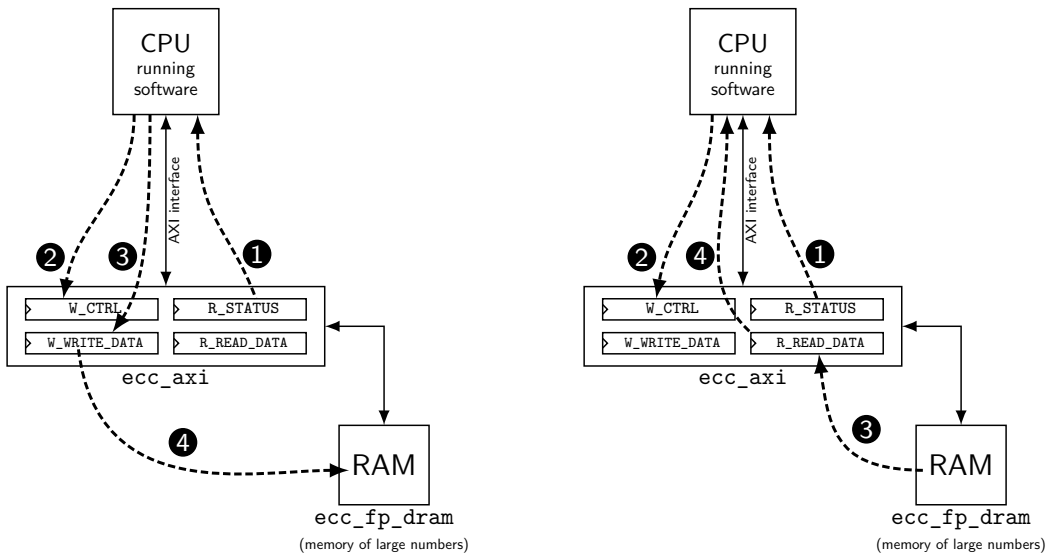
We also refer the reader to the tutorial in Appendix F for a description of how the IP can be integrated into the AXI interconnect of a system-on-chip.

5.3 Software runtime interaction with the IP

5.3.1 Overview

Interactions between software and IP are illustrated on figure 5.1 (fig. 5.1a illustrates write operations whereas fig. 5.1b illustrates read operations). As large numbers are usually much larger than 32 bit (or 64 bit) in size, several accesses are needed to read or write a large number from/to `ecc_fp_dram`, the IP internal memory of large numbers.

We strongly recommmend to refer to Appendix E, which provides pseudocode programming sequences, to complete the reading of the present chapter.



(a) write (b) read

Figure 5.1: Sequence of operations for software to write (a) or read (b) large numbers from/to `ecc_fp_dram` memory

Writing one large number into `ecc_fp_dram` (refer to figure 5.1a) requires software to first poll the status of the IP through its `R_STATUS` register until it shows an idle state (❶) then to access the `W_CTRL` register (❷) by indicating the desire to write a whole large number, along with target address in `ecc_fp_dram` of said large number. Refer to §C.1.1.1 for detail on the bit-fields of `W_CTRL` register that need be set for this. Then a contiguous sequence of AXI write transfers is expected to be produced by software (❸) in a quantity exactly equal to the number of 32-bit words (or 64-bit words if IP is configured in 64-bit mode) that are needed to encode a large number of bitwidth `nn` – that is in quantity $\lceil \frac{nn}{32} \rceil$ if IP is configured in 32-bit mode, and $\lceil \frac{nn}{64} \rceil$ if IP is configured in 64-bit mode. As data words are written by software into `W_WRITE_DATA` register they are transferred from `W_WRITE_DATA` register into `ecc_fp_dram` memory (❹) with a size conversion from 32-bit (or 64-bit) chunks into `ww`-bit ones[2].

Example with curve parameter:

```
a = 0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
```

with hardware settings: `nn` = 256 and assuming a 32-bit AXI interface. Register `R_STATUS` was read and `STATUS_BUSY` bit found deasserted.

1. Software writes `W_CTRL` register with value `0x00011000` (as address of curve parameter a is `0x01`)
2. $\lceil \frac{nn}{32} \rceil = 8$, therefore software performs 8 write cycles in a row into `W_WRITE_DATA` register (address `0x08`) transferring the following 32-bit values one after the other: `0xd86e9c00`, `0xe8fcf353`, `0xabc8ca6d`, `0x3961adbc`, `0xce42435b`, `0x10126de8`, `0x0b3ad58f` and `0xf1fd178c`.

☞ Each time software wants to write a large number into `ecc_fp_dram` it needs to write the whole large number. Accessing only a subset of memory words inside a particular large number is not possible. Contravening to this rule (for instance by not issuing the expected number of writes into `W_WRITE_DATA` register) will lead to unexpected behaviour.

Reading one large number from `ecc_fp_dram` (refer to figure 5.1b) requires software to first poll the status of the IP through its `R_STATUS` register until it shows idle state (❶) then to access the `W_CTRL` register (❷) by indicating the wish to read a whole large number, along with target address in `ecc_fp_dram` of said large number. Refer to §C.1.1.1 for detail on the bit-fields of `W_CTRL` register that need be set for this. IP then automatically starts fetching first `ww`-bit chunks of target large number from `ecc_fp_dram` into `R_READ_DATA` register (❸) which can be read back by software (❹). Therefore a contiguous sequence of AXI read transfers is expected to be produced by software in a quantity exactly equal to the number of 32-bit words (or 64-bit words if IP is configured in 64-bit mode) that are needed to encode a large number of bitwidth `nn` – that is in quantity $\lceil \frac{nn}{32} \rceil$ if IP is configured in 32-bit mode, and $\lceil \frac{nn}{64} \rceil$ if IP is configured in 64-bit mode.

☞ Each time software wants to read a large number into `ecc_fp_dram` it needs to read the whole large number. Accessing only a subset of memory words inside a particular large number is not

[2]For the meaning of parameter `ww` see Appendix D.

possible. Contravening to this rule (for instance by not issuing the expected number of reads from `R_READ_DATA` register) will lead to unexpected behaviour.

As parameter `nn` may not necessarily be a multiple of `ww`, the last 32-bit word (or the last 64-bit word if IP is configured in 64-bit mode) transferred by software in order to write a large number into `ecc_fp_dram` might contain supernumerary (useless) bits. The value of these bits is not important (software is not required to set them to 0) as the shifting logic in the IP will ignore these bits and keep only the exact total of `nn` bits. Furthermore the IP will enforce a length of `nn` + 4 bits for all large numbers[3] transferred by software into in `ecc_fp_dram`, by adding 4 extra 0 valued bits to the most significant part of each. On the other hand, when reading a large number from `ecc_fp_dram`, the last 32-bit word (or the last 64-bit word if IP is configured in 64-bit mode) of data transmitted from `ecc_fp_dram` will be appropriately padded with 0-valued bits by the IP in its upper part, in order to guarantee that software gets a large number with all bits beyond the `nn`'th one set to 0.

The set of IP registers accessible through the AXI interface is listed in table C.1 p. 63 and described in detail in appendix C. Refer to §E.2 (resp. §E.3) for more detail on the precise sequence of operations required for software to write (resp. read) a large number.

[3] The reason for these 4 extra bits is explained in Appendix D.

Appendix A

Customizing the IP

A.1 Parameters related to cryptography and technology

A.1.1 Parameter `nn`

Definition. Main security parameter.

Type/value. Integer. No limit except that of the hardware ressources available in your target/die-area.

Description. Defines the size in bit of large numbers implied in cryptographic computations: the prime number p of course which defines the field on which the elliptic curve is based on, the curve parameters a, b, q of the curve, the point coordinates $x_{\mathcal{P}}$ and $y_{\mathcal{P}}$ of the base point, and all intermediate variables used during computations.

Note: the order q of the curve can be greater than p (by a quantity that, according to Hasse theorem, may be up to $2\sqrt{p}$) therefore the value of `nn` should be chosen as

$$\max(\text{size of } p, \text{size of } q).$$

Refer for instance to [HMV04] p. 82, theorem 3.7, or the Wikipedia article «Hasse's theorem on elliptic curves» `https://en.wikipedia.org/wiki/Hasse%27s_theorem_on_elliptic_curves`.

See also. `nn_dynamic`

A.1.2 Parameter `nn_dynamic`

Definition. Option to set the *dynamic prime size* feature.

Type/value. Boolean (true or false).

Description. When this option is set to `TRUE`, it becomes possible for software driver to dynamically set the security parameter, meaning the value of `nn` can be modified at runtime. This is done by writing register `W_PRIME_SIZE`. The value statically set to `nn` in the present file then becomes the maximum value allowed at runtime. Hardware enforces verification of this and, in case the condition that the value set by software is greater than `nn`, raises an error flag in main status register `R_STATUS` and freezes operations until a correct value is set again by software.

When the option it set to `FALSE`, software cannot modify value of `nn` at runtime, and only cares the value set of `nn` in the present file. The advantage of setting this option to `FALSE` when the feature is not considered useful for your own design is to save logic.

The advantage of setting this option to `TRUE`, when you need for your own application to be able modify the security parameter at runtime, is to increase performances (latency and throughput) whenever a smaller value of `nn` can be chosen.

See also. `nn`

A.1.3 Parameter `techno`

Definition. Defines the technology (ASIC vs FPGA) you wish to target and, in the FPGA case, the vendor/part.

Type/value. Enumerate. Choices are between:

- `series7`, `spartan6` (not supported yet but easy to implement), `ultrascale` for ARM-Xilinx FPGAs
- `ialtera` for Intel-Altera FPGAs (not supported yet, difficult given the strong restriction from Intel-Altera to instanciate «at-hand» the low-level primitives required for the TRNG).
- `asic` if you're designing an ASIC or a system-on-a-chip

Description. This parameter mainly impacts the instanciation of the pipelined chain of multipliers-accumulators (aka «MACC» in ASICs and «DSP blocks» in FPGAs) inside each Montgomery multiplier, as this is almost the only hardware feature that needs to be «black-box» instanciated in the design (as opposed to inferred by synthesizer from behavorial VHDL).

See also. `multwidth`

A.2 Parameters related to performance

A.2.1 Parameter `multwidth`

Definition. Only used if `techno = asic` and in this case designates the size of limbs in which large numbers are split and are accessible into/from the memory of large numbers.

Type/value. Integer. No limit a priori but you should obviously consider the possible loss of performance of a multiplier which inputs would become too large in your specific technological node. Default of 32 seemed fair for a multiplier hardwired in an ASIC.

Description. This is the size (bitwidth) of the input operands to the multipliers in the design. For sake of architectural simplicity, it is hence also the bitwidth of the limbs in which large cryptographic numbers are split, buffered and processed in arithmetic operations inside the IP. If your design targets an FPGA, `multwidth` is ignored and the size of the limbs is given by parameter `ww` instead (not customizable and automatically set by the RTL).

- In an ASIC, there is obviously no predefined size for the multiplier you wish to use in your design, unless your founder/standard-cell library imposes you one. Therefore the designer can tweak this parameter to set the area and performance of multipliers that will be inferred in the hardware.
- In FPGA circuits, the multiplier-accumulators, which are called «DSP blocks», already exist («hard-coded») and cannot accept any size of operands on their inputs. For instance the 7-series family of FPGA from ARM-Xilinx offers the DSP48E1 primitive which is a 25x18 signed multiplier driving a(n also signed) result on 48 bits. The 5 bit difference (48 - (25 + 18)) means that 5 extra bits are present in the accumulator part of the DSP block to allow 32 output terms of the DSP block to be added together (or added to the output of neighbouring DSP blocks) without an overflow incurring. The DSP blocks are performing to their best when they are instanciated as a chain (aka in a pipeline) because there exist dedicated fast physical connexions on both inputs and outputs of neighbouring pairs of DSP blocks that bypass the general-purpose routing fabric of the circuit, thus possibly incurring very high pipeline frequencies. This is the architectural choice made within the IP for implementing multiplications of large numbers that are required to carry on Montgomery multiplications (c.f source file ⟨`mm_ndsp.vhd`⟩). Similarly, in the Spartan-6 family, also from ARM-Xilinx, the DSP48A1 primitive allows signed 18x18 multiplications and an accumulated output of 48 bits. In Intel-Altera FPGAs such as Arrias and Stratix the DSP blocks can be configured in 3 different ways: 9x9, 18x18 and 27x27, with an accumulation on up to 64 bits.

The size of the limbs of large numbers manipulated in the IP (hence also the size of the inputs to the multipliers) is denoted `ww` (for word's width) and is set statically (meaning at compilation/synthesis time) by function `set_ww` of package `ecc_utils` (file ⟨`ecc_utils.vhd`⟩):

- if you have set `techno = asic`, value of `ww` is set to the same value as `multwidth`

- if target technology is an FPGA one, `ww` is set according to the `techno` parameter:
    - for ARM-Xilinx FPGAs, `ww` is set to 16. Note: the reason for this number is that Xilinx DSP blocks being either SIGNED 25 x 18 multipliers (in 7-series and ultrascale) or SIGNED 18x18 multipliers (in Spartan-6) multipliers, only 17 bits are actually available when doing unsigned arithmetics (which is what we need when splitting large multiplications into smaller ones performed on limbs). The MSBit is necessarily tied to a logic 0. For sake of number readability when simulating the IP, the value of 16 was chosen instead of 17, which should not incur a significant difference in the final area nor the performance of the design. However if you really want to scrimp and save to the maximum, you can tweak `set_ww` function to return 17 instead of 16.
    - for Intel-Altera FPGAs, `ww` is set to 27. This is a peremptory choice that you can also modify to a default 9 or 18 by editing the code.

Note: in any of the cases described above, the VHDL code in ⟨`mm_ndsp.vhd`⟩ performs a simple static test in order to enforce that the number of chained/pipelined multiplier-accumulators cannot create an overflow on the output result of the chain. The dynamic on the ouput value of the chain is simply given by $(2\times \texttt{ww}+\log_2(\texttt{ndsp}))$ where `ndsp` is the number of multiplier-accumulator blocks in the chain (and `ww` as defined previously) so the test merely consists in ensuring that this quantity does not exceed the max accumulation available in accumulators of the target technology. (For instance in 7-series parts, DSP blocks have a 48 bit accumulator, hence the maximum number of such blocks when tiled in a pipeline chain is given by $2\times 16+\log_2(\texttt{ndsp}) = 48$, which yields a huge number (65536) that cannot be reached physically on one single die. On the other hand, if you're targeting an ASIC and your multipliers are 32x32 with an accumulator of let say 68 bit, you may only thread 16 MACC per Montgomery multiplier, which will usually do the job but won't fit the needs of an application targeting a 521 bit security (521 bit is the maximum security you may find in standardized crypto protocols based on elliptic curves in 2023). The numerical examples we've seen just above show you that in most cases you won't have to worry about `ww` and `multwidth` parameters.

See also. `nbdsp`

A.2.2 Parameter `nbmult`

Definition. Number of Montgomery multipliers instanciated in the IP.

Type/value. Integer which should be kept small (default being 2)

Description. The number of Montgomery multipliers in the IP should be dictated by the maximum Montgomery multiplications (aka REDC operation) that can be simultaneously carried out when performing point operations on a curve, that is considering one possible set of formulæ for point addition, point subtraction, and point doubling, among all the different ones which are available from state-of-the-art.

In IPECC we use the so-called CoZ formulæ which are available in the jacobian projective representation of points. This set of formulæ is made possible when the points that need be added do share the same Z coordinate. CoZ formulæ were introduced in [Mel07]. A comprehensive survey can be found in [Riv11].

From the purely algorithmic/software point of view, CoZ formulæ reduce rouhgly by half the number of REDC operations needed to complete one point addition, as compared to formulæ that were existing so far (e.g [CMO98]). Obviously in hardware this is a matter of area/speed trade-off, however the gain can be estimated by stating that the same computation speed can be obtained as compared to a non-CoZ implementation with only half the number of Montgomery multipliers instanciated in the hardware. In IPECC by default only 2 REDC operators are instanciated in the design (this is block `mm_ndsp`, c.f source file ⟨`mm_ndsp.vhd`⟩). This is the maximum number of Montgomery multiplications that it is possible to carry out in parallel due to the dependency that exists between intermediate variables in the CoZ formulæ. This means that setting `nbmult` to a value more than 2 for your design would simply increase – quite significantly – its surface, without improving the speed of curve computations at all. Normally you don't want to do that. On the other hand, if for any particular reason you're considering choosing another set of formulæ for your specific design, then you may also consider tweaking parameter `nbmult`.

Note that the set of formulæ implemented in IPECC is implemnented in software – what we call the microcode –, using a custom assembly language. It can be easily edited/modified, however the hardware logic implementing the patch mechanism for side-channels countermeasures in component `ecc_curve` would have

to be completely redesigned and recoded accordingly. Visit the page *Explicit-Formulas Database* of website `hyperelliptic.org` for a comprehensive description of elliptic curve formulæ, with a comparison of their performances.

A.2.3 Parameter `nbdsp`

Definition. Number of MACC/DSP blocks per Montgomery multipliers

Type/value. Integer greater or equal to 2

Description. This parameter directly influences the performance of the IP by allowing you to control the area/speed trade-off in the Montgomery multipliers. Obviously the more there are MACC/DSP blocks in each Montgomery multiplier, the less it will take for them to carry out a complete REDC operation. Note however that the «speed vs nbdsp» relation is linear only in a specific range of the `nbdsp` parameter, a range which can sometimes turn to be quite small – meaning that past a specific threeshold (that is dependent mainly on the values of `nn` and `ww`) the computation speed obviously will plateau however hard you try and increase parameter `nbdsp` (actually it may even decrease due to the fact that the excess of MACC/DSP blocks will increase the time for data to go through the chain without increasing the number of usefull multiplications per clock cycle). The table below gives the duration in microseconds of one REDC operation for `nn = 256` for `techno = asic`, for different values of the parameter `ww` (which in the case of asic matches `multwidth`) and different values of the `nbdsp` parameter. The `mm_ndsp` block is assumed to run at 300 MHz.

ww \ nbdsp	2	3	4	5	6	8	10	12	13
8	12.8	9	7.8	6.7	6.1	5.5	5	4.5	4.5
16	4.3	3.3	3	2.7	2.4	2.4	2.2	2.2	
32	1.8	1.4	1.4						
64	.9	.8.	9						

Note that the static value of parameter `nn` determines the maximum value allowed for `nbdsp`, which matches the number of `ww`-bit limbs large numbers are made of. Function `set_ndsp` in package `ecc_pkg.vhd` enforces that this limit is not exceeded, it is used to compute the value of a parameter called `ndsp` (based on `nbdsp`) which is the constant actually used in the RTL code: if value of `nbdsp` set by user in present file is below the limit, then `ndsp` = `nbdsp`, and if it is greater, then ndsp is set to the limit. For instance in the example above and with `ww = 16` the limit to `nbdsp` is 17, therefore any value for `nbdsp` above 17 will be ignored and replaced by 17. Now the table above also tells us that 17 is already a suboptimal value for `nbdsp`, and that you'd probably better set `nbdsp = 10` instead.

A.2.4 Parameter `sramlat`

Definition. Read latency for all the blocks of SRAM memory used in the IP

Type/value. Integer, with only values 1 or 2 allowed.

Description. All SRAM blocks instanciated in the IP are purely synchronous and have the same latency when accessed in read mode, which can be either 1 or 2 clock cycles, as defined by this parameter. Choosing 2 instead of 1 simply means trying to increase the maximum frequency at the cost of an extra layer of flip-flops on the output data bus. This can be of interest in particular in FPGAs as the SRAM blocks are natively provided with this extra layer of registers inside the block itself, which is why synthesizer will most probably infer to use these internal registers instead of consuming flip-flops from the general logic fabric. Hence for an FPGA technology you probably want to set this parameter to 2.

A.2.5 Parameter `async`

Definition. If set to `TRUE`, the Montgomery multipliers in the design will reside in a specific clock-domain (independent of the rest of the IP).

Type/value. Boolean (true or false).

Description. This is an attempt at prooviding a *globally-asynchronous / locally-synchronous* feature in the IP. The Montgomery multiplication is notoriously the most costful operation in asymetric cryptography algorithms, so the purpose of this option is to allow the designer to isolate the Montgomery multipliers in their own specific clock-domain that can thus be optimized as regards to timing considerations.

As a hardware designer you may try to:

- increase the frequency of that specific clock domain without stressing the backend tools on the remaining parts of the circuit, as they are less impacting on performances
- perform timing optimization solely on this clock domain as it is the one that deserves the highest optimization effort. Obviously the two aspects are related and should be analyzed together.

Depending on the value of `async` paramter, the clock-domains in the IP are as follows:

- if `async = FALSE`, then there is only one clock-domain and the IP is totally synchronous to the same clock, which thus also happens to be the clock of the AXI-interconnect the IP is connected to.
- if `async = TRUE`, then there are two clock-domains in the IP. One clock-domain is the one in which data transfers are made on the AXI-interconnect and all arithmetic operationsa except REDC are also carried out. The second clock-domain is the one of the REDC operators (Montgomery multipliers). In this case the usual synchronization issues at the crossing of the domains are resolved in two differents ways:
  - for control signals a layer of two or three resynchroniozation flip-flops is used
  - for data signals, we use the fact that most FPGA families offer asynchronous dual-port memories to push data in one clock domain and pull them in another without having to worry about synchronization issues (possible metastability in the read clock-domain that would be due to asynchronicity with write clock will be gone by the time data are actually read).

  If you're targeting an ASIC, you should first check the availability of single dual port (asynchronous) memories in your technological library.

Note having solely one clock-domain may be penalizing in FPGAs if the SoC and interconnect (the IP is connected to) imposes you to run at a low frequency. Values such as 100 MHz or 150 MHz are typically found in SoC-FPGA ecosystems however this is not the highest frequency by far that you can hope the Montgomery multipliers to run at. Actually 250 MHz should be seen as the minimum target on the 28 nm node FPGAs such as the ARM-Xilinx 7-series family or the Intel-Altera Stratix V family. On Xilinx UltraScale+ technology (16 nm) a frequencly close to 400 MHz was obtained.

If `async` is set to `FALSE`, then the unique input clock to the IP is the `asyns_axi_aclk` input port on top-level `asynecc` entity. If `async` is set to `TRUE`, `s_axi_aclk` is still the primary clock input, and the dedicated clock to the REDC operators must be driven on `clkmm` input port of the IP top-level.

A.3 Parameters related to side-channel countermeasures

A.3.1 Parameter `debug`

Definition. To choose between *debug mode* versus *production mode* of the IP. In a nutshell, debug mode means any tampering with the IP is made easy to the software driver. On the contrary production mode means hardware security is at its max.

- Setting `debug` to `TRUE` is very dangerous and should not apply to production purposes, as this mode allows software driver to tamper in any possible way with the IP, making it possible for each security feature and countermeasure to be disengaged at runtime at the software initiative.
- Setting `debug` to `FALSE` is what you wish if you're targeting production mode and you aim at disposing of a true hardware secure element for your application.

Type/value. Boolean, true or false.

Description. The IP can be used in two different modes which are exclusive of one another: either the IP is configured in production mode or it is configured in debug mode. This configuration is static and can't be modified at runtime. Setting `debug` to `TRUE` naturally means setting the IP in the debug mode, and in production mode otherwise.
Debug mode means that interacting with the IP through software driver is made very permissive, so as to allow pre-production analysis of the IP and of its side-channel leakages. Software driver can then tamper freely with almost any security feature implemented in the IP.
In debug mode, software-driver can:

- read or write the value of any large number in memory, at any time
- insert breakpoints into microcode to interrupt scalar multiplication and perform step by step execution (e.g to allow time isolation of a specific part of $[k]\mathcal{P}$ computation for clean, reduced-noise noise side-channel measurement)
- specify a precise time in the course of $[k]\mathcal{P}$ computation where to activate or deactivate the trigger signal driven out of the IP (this is `dbgtrigger` output port of top-level entity `ecc`). Use cases for the external out trigger feature include oscilloscope input-trigger activation and EM injection probe setting-off. This feature is clock-cycle accurate
- bypass the TRNG and force use of deterministic numbers instead of random ones
- access internal memory array of the TRNG raw random bit FIFO, in order to perform entropy quality assessment
- modify the content of the microcode memory to implement and evaluate different curve formulæ or different countermeasures.
- enable or disable almost any of the countermeasures.

In production mode:

- the only large numbers software driver is allowed to WRITE in memory are the first eight ones (large number address 0 to 7). These are: prime number p, curve parameters a, b and q, scalar k and base point coordinates $x_{\mathcal{P}}$ and $y_{\mathcal{P}}$
- the only large numbers software driver is allowed to READ from memory are:
  - the two coordinates of the result point (these are $x_{[k]\mathcal{P}}$ and $y_{[k]\mathcal{P}}$ in affine representation) which are available for read at the same address as XP and YP are respectively available for write)
  - the one-shot random masking token that the IP uses to whiten the $[k]\mathcal{P}$ result coordinates. Software driver is required to read that number before ordering any $[k]\mathcal{P}$ computation, otherwise it won't be able to obtain the plain (unmasked) value of the $[k]\mathcal{P}$ result coordinates.
- breakpoints do not exist (control logic and data paths are pruned at synthesis time)
- outside trigger does not exit (control logic and data paths are pruned at synthesis time).
- TRNG cannot be bypassed, nor random values be forced, nor random values be read by software (control logic and read data paths are pruned at synthesis time)
- microcode memory cannot be modified (content is fixed at synthesis time and write data path to memory is pruned).
- Interactions between software driver and IP are strongly restricted. Whenever software issues a command that requires computation from the IP, it can no longer issues any command before that computation is totally carried out and completed.
- The choice made by the hardware designer at synthesis time on the following 3 options : `blinding`, `shuffle`, and `zremask` can only be modified by the software driver if it increases the level of security. For instance, if `blinding` is 0 in ⟨`ecc_customize.vhd`⟩, it means software driver will still be able to program $[k]\mathcal{P}$ computations with blinding. On the other hand, if `blinding` > 0, then software driver won't be able to suppress blinding at runtime. Same applies to `shuffle`, and same applies to `zremask`. The idea behind the three parameters named `blinding`, `zremask` and `shuffle` (along

with `shuffle_type`) is to allow the hardware designer, to lock the corresponding countermeasure as always applicable to each $[k]\mathcal{P}$ computation, and it is important to keep in mind that these locks are effective only if you also set at the same time parameter `debug` to `FALSE`. Setting `debug` parameter to `TRUE` instead would actually maintain the possibility to engage each of these countermeasures, but at the discretion of the software driver and on a $[k]\mathcal{P}$-computation per $[k]\mathcal{P}$-computation basis. Note that `blinding`, `shuffle` and `zremask` features are not the only side-channel countermeasures provided with the IP, but only those that you can choose to statically hardlock. This is because they are the most performance costly.

To sum-up:

- If you intend to use the IP in a security application such as a Secure Element, a Hardware Security Module or if you entend to integrate it as hardware accelerator inside a general purpose SoC, choose `FALSE`.
- If you intend to use the IP for an academic/research purpose, to evaluate the side-channel resistance of the IP on a specific target, or during the preproduction phase of your product, choose `TRUE` (in the latter case, you may consider to «logic-lock» all the portion of the design that will remain when you eventually turn the parameter to `FALSE` and send it to foundry).

See also. `blinding`, `shuffle`, `zremask`

A.3.2 Parameter `blinding`

Definition. This parameter is used to statically activate (and configure), or statically deactivate the blinding side-channel countermeasure.

Type/value. Integer. 0 means disable, otherwise any strictly positive number will set the bit size of the blinding number. Default is 96.

Description. When non null, `blinding` gives the bit size of the random number α used to randomize the original scalar k set by software, according to equation:

$$k' = k + (\alpha \times q)$$

where q designates the order of the elliptic curve.

The rule of thumb is to set a bitwidth of approx. 96 for `nn = 256`. Hardware enforces that the size be smaller than `nn` (either the static value when `nn_dynamic` = `FALSE`, or the runtime dynamic one when `nn_dynamic` = `TRUE`).

Setting 0 instead will keep the blinding countermeasure available but solely as an option that software can decide to set or not to set at each $[k]\mathcal{P}$ computation.

Important note Setting a non-0 value to `blinding` only makes sense if you also set `debug` to `FALSE`. If you set `debug` to `TRUE`, the value set for `blinding` won't make a difference as software driver wil then be considered legitimate in modifying the blinding settings at runtime, including the possibility to completely disable blinding.

See also. `debug`, `shuffle`, `zremask`

A.3.3 Parameter `shuffle`

Definition. This parameter is used to statically activate or statically deactivate the shuffling side-channel countermeasure.

Type/value. Boolean (true or false) Default is true.

Description. The purpose of the shuffling countermeasure is to break the relation between large numbers and the address in the physical memory they are read from during sensitive computations.

When set to `TRUE`, `shuffle` parameter activates the random shuffling of the complete memory storing large cryptographic numbers between the processing of each bit of the scalar. This countermeasure thwarts the

attacks aimed at guessing which intermediate variables are manipulated inside point addition formulæ by use of their address's physical leakage. As memory shuffling removes the relation between addresses and values of the aforementioned variables, these attacks become infeasible – or at least much difficult to be carried out.

Set to `TRUE` if you want the shuffling of memory of large numbers to be activated at each $[k]\mathcal{P}$ computation (memory is shuffled inbetween the processing of two consecutive bits of the scalar). The method used to to shuffle the memory is then defined as per parameter `shuffle_type`.

Setting `FALSE` instead to `shuffle` parameter will keep the countermeasure available (if parameter `shuffle_type` is different than `none`, see below) but solely as an option that software can decide to use or not to use at each $[k]\mathcal{P}$ computation.

In any case, the choice of the shuffling method is static as only one can be implemented at synthesis time, among `linear`, `permute_lgnb` and `permute_limbs` (see parameter `shuffle_type` below).

Setting `shuffle` to `TRUE` only makes sense if you also set `debug` to `FALSE`. If you set `debug` to `TRUE`, the value set for `shuffle` won't make a difference, as software driver will then be considered legitimate in modifying the shuffle settings at runtime, including the possibility to completely disable it.

See also. `shuffle_type`, `debug`, `blinding`, `zremask`

A.3.4 Parameter `shuffle_type`

Definition. Defines the way memory of large numbers is shuffled. This parameter is relevant even if `shuffle` = `FALSE`, because it defines what will be synthesized and hence instanciated in in the hardware.

Type/value. One of `linear`, `permute_lgnb`, `permute_limbs` or `none`. Default is `permute_lgnb`.

Description. The three available methods of shuffling correspond to a compromise between implementation complexity and perf. cost on one side, and efficiency as a side-channel countermeasure on the other.

The `linear` method is simple to implement and should incur almost no performance penalty at all (neither in surface nor on speed). The `permute_limbs` method should quite damage the speed performance, and is probably «overkill». Besides, as discussed below, as it requires an SRAM block memory to store the address indirection, it is possible that its gain might not be real in terms of side-channel resistance. It is also expected to consume large quantities of randomness. The `permute_lgnb` method offers an intermediate solution, both in terms of surface and speed, which makes it probably the best solution (which is why it was set as the default choice).

The description below first covers the two «extreme» methods (`linear` and `permute_limbs`), `permute_lgnb` being explained after.

- `linear` method:

  Shuffling will consist in drawing a random mask and applying it linearly (xor) to the read address of each data word to determine the target (shuffled) write address. Therefore when the countermeasure is activated the memory is physically duplicated inside the IP and a flip/flop mechanism is used to ensure consistency of the transfer of one version of the memory into its newly shuffled version. The address here designates the address of *limbs* inside the memory of large numbers. Hence limbs of the large numbers are what is shuffled here, meaning that each one of the `nblargenb` large numbers stored in memory will have its `ww`-bit limbs scattered all accross the memory array, however their addresses will keep a linear relation between them. The number of possible permutations in this case is not very important. For instace for `nn` = 256 and `ww` = 16 (and assuming the default value of 32 for `nblargenb`, the address of a `ww`-bit limb in memory will be of 10 bits, which makes it a total of 2^{10} (1024) different possible permutations (this should be compared to the 1024! ways of permutating a memory array of 1024-words, which is what is offered by the `permute_limbs` method).

- `permute_limbs`:

  This method consists in permutating the memory of large numbers using the Fisher-Yates algorithm. Generally speaking, the Fisher-Yates algorithm is used to randomly generate any permutation of an n-element set a among the total $n!$ possibilities of doing so. The algorithm is

very simple and consists in scanning the n items (for instance in the range i from $n-1$ downto 0), generating for each step i a random number j in $[0..i]$, and swapping the items of the set a in positions i and j (noted $a[i] \leftrightarrow a[j]$). The difference between the `permute_limbs` and `permute_lgnb` methods of shuffling is that in the former the Fisher-Yates algorithm is applied on limbs, while in the latter it is applied on whole large numbers. Hence in the `permute_limbs` method, many performance aspects of the implementation are expected to decrease: $[k]\mathcal{P}$ computation time mmight probably increase a lot, and an important throughout will probably required for randomness. Furthermore, and this is probably the worst, it is obvious that a second SRAM block memory is made necessary in the implementation in order to store the definition of the permutation at any time. Well, the presence of an SRAM memory in order to store the large numbers was already a drawback in itself (from the perspective of side-channels (*)) that the shuffling countermeasure was intended at mitigating in the first place. Using another SRAM block to store the permutation (i.e the address translation table) might therefore not be the more judicious to do so. That's the reason for the option `permute_lgnb` described hereafter. The advantage of the `linear` option is also that it won't be synthesized but in logic gates, not memory. (Finally the option `permute_limbs` was kept in the source code to allow experiments).

(*) For instance in FPGAs all physical memories have the same size (typically 32 kbit) therefore the physical leakage occuring while reading a large number from its RAM block is expected to be the same as the one that would occur when reading the randomized address from another RAM block. Power consumption as well as EM emission in digital circuits is proportional to the capacitive load of nets, which are quite important in memory physical layouts.

- `permute_lgnb`:

  Here `lgnb` stands for large numbers, meaning the Fisher-Yates algorithm is used to permutate whole large numbers (not only their limbs). Considering the default value of 32 for the number of large numbers stored in memory, it is then possible to synthesize the translation table as a tiny memory of 160 bits (32 words of 5 bits) whose side-channel signature should be a lot weaker than the large memory of large numbers, while still allowing the complete $32!$ permutations ($\sim 10^{35}$) to be equally feasible (the component named `virt_to_phys_ram_async` which implements this memory describes an asynchronous read memory in order to enforce this synthesis result in FPGAs (in Xilinx FPGA, the translation memory could thus be synthesized using only 4 LUT located in the same slice).

Note. As for `blinding` and `zremask`, it's possible for the software driver to enable shuffling even if option `shuffle` was statically set to `FALSE`, and provided that `shuffle_type` was set no another value than `none`. It's not possible to modify the shuffling method dynamically (meaning at runtime).

If `shuffle_type` differs from `none`, then only one shuffling method will be synthesized and present in the hardware. If moreover `debug` = `TRUE`, then software driver will be able to enable or disable the shuffling.

Now if `shuffle_type` = `none`, software driver won't be able to activate shuffling. This is because the hardware implementing the shuffling of memory won't be present in the circuit to begin with.

See also. `shuffle`, `nblargenb`, `debug`, `blinding`, `zremask`

A.3.5 Parameter `zremask`

Definition. Used to enable or disable statically the periodic repetition, all along the course of $[k]\mathcal{P}$ computation, of random re-generation of the coordinates of sensitive points, using each time a new fresh random value. This countermeasure is based on the so-called Jacobian projective representation which by definition uses 3 coordinates $(X : Y : Z)$ to characterize points on an elliptic curve, instead of two in the affine (x, y) system.

Type/value. Integer. A value of 0 disables the countermeasure, yet software driver will still be able to activate it at rutime. Default value is 16 but this is quite arbitrary. You should consider the performance penalty induced by the countermeasure when selecting the value for this parameter (and when doing so, keep in

mind that the smaller the value, the bigger the performance loss) – see last paragraph of section **Description.** below for performance considerations.

Description. For `zremask` parameter, setting a non-0 integer will define a periodicity, expressed in number of bits of the scalar, at which the coordinates of points $\mathcal{R}_0$ and $\mathcal{R}_1$ (used throughout the scalar loop to compute $[k]\mathcal{P}$) will be re-randomized using a fresh multiplicative random (aka «Z-masking» countermeasure). Note that such a masking is always applied at the beginning of the scalar loop, regardless of value set for `zremask`. What you can set with `zremask` is to force that masking to happen again and periodically throughout the entire scalar loop. If you set for instance `zremask` = 4, then coordinates will be randommized with a new fresh random every one in four bits of the scalar. There is a 1-1 correspondence between the set of affine points

$$(x, y) : x, y \in \mathbb{F}_p$$

and the set of projective points

$$(X : Y : Z) : X, Y, Z \in \mathbb{F}_p, Z! = 0$$

verifying $x = X/(Z^2), y = Y/(Z^3)$. Hence the Z coordinate can be used as some kind of «free» variable allowing us to choose between virtually an infinite number (actually $p - 1$) of ways to represent any point on the elliptic curve. For any valid representation $(X : Y : Z)$ that belongs to the equivalence class of an affine point (x, y) then for any number $L \in \mathbb{F}_p\ 0$, the triplet $((L^2).X : (L^3).Y : L.Z)$ belongs to the same equivalence class, hence represents the same point. This countermeasure has been called *Randomized projective coordinates* in the paper that originally introduced the idea [Cor99]. In the context of the IP we call that countermeasure *Z-remasking* as the idea is to allow the user of of the IP/designer of the hardware system to repeat the randomization of the point representation several times inside each $[k]\mathcal{P}$ representation.

Setting `zremask` = 0 will keep the Z-masking countermeasure available but solely as an option that software can decide to set or not to set at each $[k]\mathcal{P}$ computation.

Setting a non-0 value to `zremask` only makes sense if you also set `debug` to `FALSE` (if you set `debug` to `TRUE`, the value set for `zremask` won't make a difference as software driver will then be considered legitimate in modifying the `zremask` settings at runtime or simply disabling it).

Note that when option `nn_dynamic` is set to `TRUE`, the value set for `zremask` is independent of the runtime value `nn` can take. If for instance `zremask` $= 4$, re-randomization will happen every 4 bits of the scalar whether `nn` = 256 or `nn` = 384, simply in the former case it will happen 64 times, and in the latter 96 times.

The cost of the countermeasure is 8 REDC (8 Montgomery multiplications) at each remasking, so you'd better consider the performance cost of setting the countermeasure `zremask`, given that each bit of the scalar already costs 16 REDC per itself in Co-Z representation (which is the one used in the IP). Obviously the performance cost decreases as the value of `zremask` increases (choosing for instance `zremask` $= 1$ would mean randomly refreshing the point coordinates after each bit of the scalar, which would dramatically increase the computation cost of the $[k]\mathcal{P}$ operation).

See also. `debug`

A.4 Parameters related to the true random generator

A.4.1 Parameter `notrng`

Definition. Used to amputate the testbench from the TRNG HDL despcription which is not fit to simulation and replace it with a file containing «random» data.

Parameter `notrng`:

- must be set to `FALSE` in synthesis (otherwise synthesis will fail)
- must be set to `TRUE` in simulation (otherwise simulation will hang).

Type/value. Boolean (true or false). Default is `TRUE`, hence fitting simulation. Change to `FALSE` before synthesis!

Description. The HDL description of the TRNG in the IP contains a combinational loop which cannot be simulated (it would hang the simulator engine by creating an infinite number of simulation steps (deltas) inside each physical instant). This is why you must set this parameter depending on what you're doing with the IP, simulating it or synthesizing it:

- when you're simulating, parameter `notrng` must be set to `TRUE`. All the ES–TRNG instances are then removed from the HDL model, as well as the binary tree gathering their outputs (see above discussion of parameter `nbtrng`). Instead a simulation-only process is instanciated that will read «random» data from the file specified in parameter `simtrngfile` (see below this parameter).
- when you're synthesizing, parameter `notrng` must be set to `FALSE`. The HDL model then embeds the ES–TRNG component with the combinational loop describing the ring oscillator.

If you provided the IP with a random post-processor (again refer to discussion for parameter `nbtrng` above) it will still be part of the simulation model when `notrng` = `TRUE`.

WARNING Vivado tool from ARM-Xilinx seems not comfortable with this parameter being set to either one of `/dev/random` or `/dev/urandom` (it will halt simulation from the begining).

See also. `simtrngfile`

A.4.2 Parameter `nbtrng`

Definition. Number of TRNG primitives instanciated in parallel in the IP.

Type/value. Integer which should be set according to the entropy throughput required for a specific application. Default is 1.

Bibliography. [YRG+18] `https://tches.iacr.org/index.php/TCHES/article/view/7276`

Description. The physical true random number generator used in the IP is the ES–TRNG, designed at the COSIC research group of KU Leuven. It was first published at the CHES'18 conference. Its architecture makes it particularly suitable to FPGA designs. It has a very small footprint, while still exhibiting very good throughput results. Porting it to an ASIC technology should not be difficult as the only FPGA-specific features that it relies on are carry propagation primitives (usually used for adders and counters) which are actually used in ES–TRNG as pure delay propagation lines. ASIC buffers would normally fit this purpose without any problem.

We do not give here the architecture nor the design principles of ES–TRNG, rather we point the reader to its online presentation paper (c.f section Bibliography above).

The way ES–TRNG is used in the IP is that exactly `nbtrng` copies of the ES–TRNG primitive are physically instanciated in the IP. They operate completely independently one from the other and «periodically» generate a random bit along with its strobe signal (the quotation marks on word «periodically» are because each random bit itself is generated after a period of accumulation of jitter which is itself random, althought there is a minimum for that delay which is given by parameter `trngta` - see below). The outputs of each different instances of ES–TRNG primitive are gathered together using a binary routing tree that terminates with a unique root output. (Althought not required, it is probably best that you set a power of two for value of `nbtrng`). Due to the average number of clock cycles it will take for each ES–TRNG primitive to issue one random bit (this number is related to `trngta` parameter, see below) congestion of the tree is not to be expected unless you really set too large a value for `nbtrng`. Ideally this parameter should be set to a number such that at each clock cycle, and given the frequency at which the IP main clock is set, a new random bit presents itself at the root of the tree. Output raw random bits are pushed into a FIFO which feeds a post-processing unit.

⚠ The post-processing unit is not part of the IPECC design and should be provided by your own care. the role of post-processing function in random applications is important because it guarantees that the ouput of the random generator remains unpredictable to an attacker even if she takes control of the physical entropy source and that the output of the physical entropy source becomes deterministic to her (e.g through a «stuck-at-0» or a «stuck-at-1» invasive or semi-invasive kind of attack).

In present release of the IP, as no postprocessing unit is provided, the output port of the raw random bit FIFO is directly connected to the logic that would normally extract postprocessed bits. This «logic stub» allows the IP to be operational as is – that is, despite the absence of postprocessing on the raw random bits – while allowing to very simply plug any cryptographic logic component that you may design or reuse to implement the postprocessing, provided that its interfaces fit the ones we use inside the IP TRNG (which are very basic).

Past the postprocessing unit, random bits are pushed one at a time into one of the 4 possible target FIFOs, each serving as the entropy pool for one of the 4 features/countermeasures of the IP that require random data. These 4 features are:

1. the on-the-fly masking of the scalar in the AXI interface, at the time it is pushed by software driver in the memory of large numbers;
2. the NNRND instruction which IP microcode can use to generate a complete random large number in the memory of large numbers;
3. the 4-by-4 shuffling that the IP applies to the 4 coordinates `XR0`, `YR0`, `XR1` and `YR1` of points $\mathcal{R}_0$ and $\mathcal{R}_1$ inbetween the processing of each bit of the scalar;
4. the shuffling countermeasure of the complete memory of large numbers, when it is set and activated (see parameter `shuffle` above).

A round-robin scheduling is applied to the bits pulled from the postprocessing unit to ensure that each of the 4 target FIFOs fairly gets the same amount of random data as long as it is not already full. Needless to say, no bit pushed in any of the 4 FIFOs gets aslo pushed in any of the three others. The 4 FIFOs aare this filled with completely independent random bits.

The sizes of each of the 4 target FIFOs are defined in the package file ⟨`ecc_trng/ecc_trng_pkg.vhd`⟩. The default sizes are functions of the parameter `nn` (directly or indirectly, as some depends on parameter `n` which in turn depends on `nn` and `ww`). If you need to customize the amount of random data equired for each of the 4 features in your application, you may do it in this file.

In debug mode, diagnostic features allow you to read the content of the raw random bit memory through the AXI interface in order to perform statisticial tests and estimate the entropy of each TRNG instances (the entropy performance of each instance directly relies on its floorplan realization, therefore they won't be necessarily the same for all instances).

For FPGA applications, the design rationale you should apply here is that once the entropy quality s estimated good enough of one instance (this is assessed using statistical tests suits such as the one provided by U.S NIST or german BSI) you should «logic-lock» this instance using the software feature that CAD tools provide you with, preventing the instance from being re-placed or re-routed elsewhere in future place-and-route runs of your circuit.

Note that the IP does not include statistical self-tests.

Important note. The default setting of 1 for parameter `nbtrng` might not be sufficient to your application! You must perform a throughput analysis and evaluation of the randomness your application specifically needs, and set, along with a properly chosen `trngta` parameter, the number of ES–TRNG instances that is fit to achieve this throughput. The effect of not assessing the quantity/throughput of entropy your application needs is that the IP might often stall at runtime. This will happen each time the IP needs to perform an operation that requires a minimum amount of entropy to be properly carried out till the end, and that amount is not available yet in the corresponding FIFO.

See also. `trngta`, `notrng`

A.4.3 Parameter `trngta`

Definition. Main sizing parameter of the entropy quality generated by the ES–TRNG primitive.

Type/value. Integer. Default is 32 but this value is quite arbitrary.

Description. The greater the value of `trngta` is, the highest the entropy per random bit is, but also the smaller the random production througput. The smaller the value of `trngta` is, the greater the throughput is, but also the poorer the entropy per bit is.

The principle of ES–TRNG is the same as many other TRNG designs: a free running oscillator is formed using a ring of inverters in odd number, and let to run freely until a specific amount of time is passed so as to consider that enough entropy has been accumulated in the phase noise (or *jitter* in the time domain) of the oscillator edges. Value of parameter `trngta` exactly denotes the duration of that «free-running» phase of the oscillator, expressed in number of periods of the main system clock of the IP. (Remember from what was stated earlier – see `async` section above –, that is clock is the one connected to the input port `s_axi_aclk` of the top-level entity). The meaning of the value of `trngta` therefore depends on the frequency of that main clock and it carries no sense without it. When the free-running phase has passed, some very tiny logic (that can fit into a very small number of FPGA LUTs) is triggered to detect the first rising edge of the free oscillator. This detector samples the information as to whether the sampling clock «saw» the edge of the oscillator before or after its own edge, thus capturing in one bit the jitter of the free-oscillator.

Only tests and measurements made on real hardware can assess that number properly.

See also. `nbtrng`, `notrng`

A.4.4 Parameters `trng_ramsz_[raw|axi|fpr|crv|shf]`

Definition. These 5 parameters each set the size of one of the FIFO used to buffer random numbers inside the IP before they are serviced to their respective entropy clients.

Type/value. All 5 parameters are integer, expressed in kilo-bytes.

Description. In terminology of AIS31 standard, parameter `trng_ramsz_raw` is related to *raw random numbers*, which are random numbers directly taken at the output of the physical source, before any logical *post-processing*. Other 4 parameters `trng_ramsz_[axi|fpr|crv|shf]` are related to *internal random numbers* (AIS31 terminology again), which are random numbers taken at the output of the post-processing operations.

Each of the 5 parameters is translated (this is done in package `ecc_trng_pkg`) into an associated parameter expressing the size of the same FIFO, but this time in number of elements (or *words*) of the corresponding FIFO's memory array, and rounded up to the next or equal power of 2.

- `trng_ramsz_raw` is translated into parameter `raw_ram_size`.
  Now as the raw random FIFO stores bits, we obviously have:
  `raw_ram_size` = greater or equal power of 2 of quantity $(\texttt{trng\_ramsz\_raw} \times 1024 \times 8)$
- `trng_ramsz_axi` is translated into parameter `irn_fifo_size_axi`.
  The FIFO of internal random number served to `ecc_axi` storing `ww`-bit words, we have:
  `irn_fifo_size_axi` = greater or equal power of 2 of quantity $(\texttt{trng\_ramsz\_axi} \times 1024 \times 8)/\texttt{ww}$
- `trng_ramsz_fpr` is translated into parameter `irn_fifo_size_fp`.
  The FIFO of internal random number served to `ecc_fp` storing `ww`-bit words, we have:
  `irn_fifo_size_fp` = greater or equal power of 2 of quantity $(\texttt{trng\_ramsz\_fpr} \times 1024 \times 8)/\texttt{ww}$
- `trng_ramsz_crv` is translated in parameter `irn_fifo_size_curve`.
  The FIFO of internal random number served to `ecc_curve` storing 2-bit words, we have:
  `irn_fifo_size_curve` = greater or equal power of 2 of quantity $(\texttt{trng\_ramsz\_crv} \times 1024 \times 8)/2$

- `trng_ramsz_shf` is translated into parameter `irn_fifo_size_sh`. The FIFO of internal random number served to `ecc_fp_dram_sh_*` storing words whose bitwidth depends on the type of shuffling algorithm selected by parameter `shuffle_type` (see that parameter and its desccription), parameter `irn_fifo_size_sh` is deduced from `trng_ramsz_shf` through a relation similar as those above but depending on the parameter `shuffle_type` (see VHDL function `set_irn_width_sh` defined in package file ⟨`ecc_pkg.vhd`⟩ and used in ⟨`./ecc_trng/ecc_trng_pkg.vhd`⟩).

See also. `nbtrng`, `notrng`, `shuffle_type`

A.5 Parameters related to microcode and memory of large numbers

A.5.1 Parameter `nblargenb`

Definition. Number of cryptographic large numbers that inner memory of the IP can buffer.

Type/value. Integer. Default is 32. Obviously keep it a power of 2.

Description. Note that this is not a memory size in bytes or bits or whatever absolute unity of size, this is a relative number. For instance if each large number is 256 bit wide, then the size of the memory of large numbers will be given by 32 × 256 bit = 8 Kbit (assuming the default of 32 for parameter `nblargenb`).

The IP is basically an ALU for large numbers controlled by a hardware state machine that fetches and decodes arithmetic instructions operating on these large numbers. An opcode format exists for such instructions describing in particular the address of the numbers in the memory of large numbers which are to be read and/or written by each instruction. Most of instructions contain 3 operands named `opa`, `opb` and `opc`, with `opa` and `opb` being the input (read) operands and `opc` the output (written) operand. All these fields are, in the current release of the IP, 5 bit address fields pointing to a large number. This value of 5 bit obviously matches the size of the memory, which is made of 32 large numbers. The 32 default was chosen because it is the minimum that we were able to fit into the set of all intermediate variables involved in the computation of the scalar multiplication (the most complex operation performed by the IP). To that default setting of 32 large numgers corresponds 5-bit fields for addresses op[abc] in the instruction opcodes, which packed along other fields turned out to to form a quite practical 32-bit size for the opcode words.

You can change the value of parameter `nblargenb`, but you should do it with precaution. The HDL code of the IP was written all along with genericity in mind, notably regards the parameters `nblargenb` and `nbopcodes` (see below) however not many simulation runs nor tests were actually done with other values than the default ones (resp. 32 and 512). So you should change these parameters only if you really know that you're doing.

Mind in particular that increasing the value of `nblargenb` will have the size of fields `op[abc]` in instruction opcodes obviously also increase (e.g to 6 bit if you set `nblargenb` to 64). This in turn will make opcodes become larger than 32 bit.

A.5.2 Parameter `nbopcodes`

Definition. Size of the microcode memory of the IP, in number of instruction opcodes.

Type/value. Integer. Default is 512. Same as `nblargenb` above: obviously keep it a power of 2.

Description. Note that this is not a memory size in bytes or bits or whatever absolute unity of size, this is a relative number. For instance if size of each opcode is 32 bit wide (the default) then the physical size of the microcode memory will be given by 512 x 32 bit = 16 Kbit (assuming the default of 512 for parameter `nbopcodes`).

Please refer to the discussion above for parameter `nblargenb`, as it also widely applies to `nbopcodes`.

A.6 Parameters related to simulation

A.6.1 Parameter `simvecfile`

Definition. Only used in simulation. Name of the input file providing input test-vector files to the simulation testbench.

Type/value. Character string indicating a file path which should be accessible in read mode.

Description. Please refer to Appendix B, particularly section §B.2 where the format expected for the input test-vector file is described.

Note that this file shares the exact same format as the input test-vector file expected by the programs `ecc-test-linux-uio` and `ecc-test-linux-devmem` that aim at testing the real hardware (the sources of which are in `driver/` folder).

A.6.2 Parameter `simkb`

Definition. Only used in simulation, to artificially restrict the size of a large scalar.

Type/value. Integer. Must be greater than or equal to 3.

Description. When `simkb` is different from 0, then the main right-to-left loop which parses the scalar bits will span only bits 0 to `simkb` - 1. You can therefore use parameter `simkb` as if you were truncating the scalar set by software driver to its lowest `simkb` bits, without modifying your simulation testbench.

When `simkb` is 0, then the main loop will parse all the bits of the scalar.

Warning. It doesn't make sense to restrict the size of the scalar (using `simkb`) in a simulation where blinding countermeasure is also enabled, or at least simply keep in mind that the quantity of bits that will be taken into account by the IP are the ones of the *blinded* scalar, hence the result can bear no more logic relation to the scalar as it was set initially by the software driver.

A.6.3 Parameter `simlogfile`

Definition. Only used in simulation. Path file for the simulation trace log.

Type/value. Character string indicating a file path which should be accessible in write mode.

Default is ⟨`/tmp/ecc.log`⟩ which should be convenient at least for Unix-like systems.

Description. This is the path of the file where simulation will dump the information detailing all the instructions that were executed by ⟨`ecc_curve.vhd`⟩, with the address of thier operands, their result, timestamp, etc.

If you provide a relative path, the place where the file will actually be placed is dependent on your simulator (the default ⟨`/tmp/ecc.log`⟩ was made an absolute path to avoid this).

A.6.4 Parameter `simtrngfile`

Definition. Only used in simulation. Name of the input file providing «random» data to the TRNG simulation testbench.

Type/value. Character string indicating a file path which should be accessible in read mode.

Default is ⟨`/tmp/random.txt`⟩ is arbitrary.

Description. Format of this file is the following: each value should be an unsigned decimal integer ranging from 0 to 255, with one value per line. Simulation will stop when it hits the end of the file.

File ⟨`sim/HOWTO-random.txt`⟩ gives an example of a one-liner command you can use to quickly generate this file on a Unix/Linux system using the `od` program with `/dev/u*random`.

See also. `notrng`

A.7 Parameters related to the AXI interface

A.7.1 Parameter `axi32or64`

Definition. Defines the width of the data buses of the AXI interfaces of the IP. These are: the `WDATA` signal bus of the AXI-lite write-data channel, and the `RDATA` signal bus of the AXI-lite read-data channel.

Type/value. Integer, with only values 32 or 64 allowed.

Description. To simplify integration of the IP inside any AXI-interconnect/system-on-chip (meaning: independently of whether the architecture is 32 or 64 bit) registers accessible by the software driver:

- are all 32-bit long (as if data buses were that of a 32-bit system) *with the exception of the two registers* `W_WRITE_DATA` *and* `R_READ_DATA`;
- have address aligned on 8 bytes (as if address buses were that of a 64-bit system).

The size of both registers `W_WRITE_DATA` and `R_READ_DATA` depends on parameter `axi32or64`:

- when the IP is configured using `axi32or64 = 32`, then `W_WRITE_DATA` and `R_READ_DATA` are 32 bit long. Therefore, if the AXI-interconnect the IP is connected to is 64 bit then:
  - when reading register `R_READ_DATA`, the 32 upper bits of the signal bus RDATA (AXI read data channel) will be cleared (zeroed) by the IP
  - when writing register `W_WRITE_DATA`, the 32 upper bits of the signal bus WDATA (AXI write data channel) will be ignored by the IP.
- when the IP is configured using `axi32or64 = 64`, then `W_WRITE_DATA` and `R_READ_DATA` are 64 bit long. Therefore, if the AXI-interconnect the IP is connected to is 32 bit, then you might consider adding extra logic in between the IP and the AXI-interconnect so as to:
  - gather two consecutive read accesses to `R_READ_DATA` into one 64 bit read transaction to the IP and split the result into two 32-bit responses to transaction initiator (CPU, memory controller, bridge, etc).
  - gather two consecutive write accesses to `W_WRITE_DATA` into one 64 bit write transaction to the IP.

  In both cases (read & write) software should be aware of the mismatch between size of the IP AXI interface and the size of the AXI interconnect, so that to enforce that transactions should always be issued by pair (otherwise deadlock might occur).

Important note. Experiments made on real hardware, namely Xilinx® Zynq® SoC/FPGA platforms, show that it's probably better to always set parameter `axi32or64` to 32. These real-hardware tests indeed showed that even when the IP was connected to an AXI 64-bit interface (and despite the fact that Cortex-A53 used for these tests were 64-bit cores) bus transactions issued from the CPU were still 32-bit ones, thus incurring data errors with an IP configured in mode `axi32or64 = 64`.

In any case, set `axi32or64 = 64` *only if* you are absolutely sure and have characterized, in your own hardware application, the property that AXI transactions emitted by the CPU when writing (resp. reading) to register `W_WRITE_DATA` (resp. from register `R_READ_DATA`) are 64-bit transactions and that they will never be split in 32-bit transactions during their complete path through the interconnect from the CPU to the IP (resp. from the IP to the CPU).

The restriction descrived above probably narrows down the utility of parameter `axi32or64`, which hence might be suppressed in future releases of the IP.

Appendix B

Simulating the IP

B.1 Preparing sources for simulation

Follow these steps to simulate the IP:

1. **Generate the input test vector file.**

   The format and usage of this file are described below in §B.2.

   You can use the script file ⟨`generate-tests.sage`⟩ in folder `sage/` to quickly and easily generate an input test vector file that fits your needs. The script contains a start section that includes the initialization of all the parameters required for the generation of the tests (we encourage you to customize the parameters) including an online textual help for each of the parameters (which you're also encouraged to read). Once you've edited the script, simply run it as input to the *SageMath* program:

   From your Linux shell prompt

   ```
   $ sage generate-tests.sage >/tmp/ecc_vec_in.txt
   Generating curves for nn = ...
   ```

   The command-line above assumes that you want your file to be generated as ⟨`/tmp/ecc_vec_in.txt`⟩ so adapt this to what you want. Simply ensure that parameter `simvecfile` (see below) in configuration source file ⟨`ecc_customize.vhd`⟩ is set with the correct complete path and filename (here we'd have: `simvecfile : string := "/tmp/ecc_vec_in.txt"`).

2. **Generate the input random values file.**

   On a Unix-like system you can use this simple one-liner to generate the file (16 millions in this example):

   From your Linux shell prompt

   ```
   $ od -t u1 -w1 -v /dev/urandom | awk '{print $2}' | head -$((16*1024*1024)) > /tmp/random.txt
   (just takes a few seconds...)
   $ head -5 /tmp/random.txt   # Obviously your values will differ
   31
   201
   222
   0
   11
   ```

   As you can see on the example code above, the format of this file is very simple. It should contain one «random» value per line, each value being represented by a positive decimal integer in the range [0 . . . 255] (both limits included). Basicly this file replaces the physical TRNG in simulation.

   When performing behavioral simulation, the RTL structural hierarchy will be given a cut of scissors at the point where random bits normally generated by the physical TRNG are gathered by bunch of 8 to

form bytes and pushed to the post-processing function. Then every VHDL component/logic located before that cutting point will be removed from compilation and altogether replaced with a process reading random bytes from the random values file and pushing them to the post-processing function identically as would happen on the read hardware.

Remember that IPECC as is doesn't include any random post-processing logic (see discussion in §A.4.2) so by default the byte values read from the input random values file will be simply gathered as 32-bit words and distributed to the different random FIFOs from which internal random numbers (according to the AIS31 terminilogy) are then extracted to serve the different portions of the logic in the IP that implement side-channel countermeasures.

3. **Edit file ⟨`ecc_customize.vhd`⟩ to set the hardware config of the IP.**

   Open file ⟨`ecc_customize.vhd`⟩ and set the different parameters (they are all described in Appendinx A) according the hardware configuration you want to simulate for the IP. We'll simply recall here the parameters directly related to simulation:

   - Parameter `notrng` must be set to `TRUE`. This will tell the HDL structural hierarchy not to instanciate the real model of the physical TRNG but to run instead a behavorial process reading random values from the input random values file. The real TRNG can't be simulated as it contains a combinational loop.
   - Parameter `simvecfile` must be set to point to the input test-vector file (see step 1 above and §B.2 below).
   - Parameter `simtrngfile` must be set to point to the input random values file (see step 2 avove).
   - Parameter `simlogfile` designates the simulation log file, it must point to a filename where the simulation testbench will dump (obviously the path must have write permission) the information detailing all the instructions that were executed by `ecc_curve`, with the address of thier operands, their result, timestamp, etc. Remember that `ecc_curve` is the tiny internal CPU of the IP executing microcode, so consider the file you set here as the execution log trace of it. Please refer to §B.5 for the description of the simulation log file.

     The default value of `simlogfile` is ⟨`/tmp/ecc.log`⟩ and most of the time you won't need to change this.
   - Mind the value of parameter `nn`. Remember that the values of `nn` found in the input test-vector file shouldn't exceed the value of `nn` set in ⟨`ecc_customize.vhd`⟩.
   - Optionnaly, parameter `simkb` can be used to restrict the size of the scalar. Suppose you have a very long scalar (say 384 bits long) whose least significant bits are like this: $k = \texttt{0x...011011011}$ and suppose that for your simulation you simply need to get the intermediate value of $[k]\mathcal{P}$ after the parsing of the 6 least significant bits. Now setting `simkb = 6` will make the computation stop after the bit of weight 6, thus providing you with the intermediate result you want (here $[91]\mathcal{P}$).

     The value of `simkb` should be greater than (or equal to) 2.

     Most of the time you want to leave this parameter to its default value of 0, which means that the scalar will be parsed entirely.

     Notes:

     (a) Obviously if you restrict the size of the scalar using this parameter the simulation will fail, as the testbench will find that $[k]\mathcal{P}$ result is not as expected in the input test-vector file.

     (b) It doesn't make sense either to restrict the size of the scalar if you simultaneously use blinding. Indeed the scalar on which applies the restriction of `simkb` is the blinded one, so what the IP will actually compute isn't $[k + \alpha q]\mathcal{P}$ anymore but $[(k + \alpha q) \mod 2^{(\texttt{simkb} + 1)}]\mathcal{P}$ instead, which holds no logic relation with $[k]\mathcal{P}$ (nor with $[k \mod 2^{(\texttt{simkb} + 1)}]\mathcal{P}$ for what it's worth).

4. **Build the microcode of the IP.**

   This is a VHDL file describing the content of a memory block in the form of a big look-up table (using VHDL clauses like `"when address => output <= value"`).

   Browse to folder `hdl/common/ecc_curve_iram/` and simply type `make`:

From your Linux shell prompt

```
$ cd ${IPECC}/hdl/common/ecc_curve_iram/
$ make
  -> Parsing ../ecc_pkg.vhd, ../ecc_customize.vhd and asm_src/vardefs.csv for checking ...
Warning:  BIGNUM_BITS_SIZE mismatches (528 != 256), updating
[+] Parsing of VHDL files done, everything OK
Warning:  operand lambdacu from CSV address differs ("10111" (@23) != "10101" (@21)), fixing
Warning:  operand token (@18, "10010") from CSV missing and added
[+] Parsing of CSV done, everything OK
  -> Assembling file ecc_curve_iram.s
    -> First pass for labels resolution done
    -> Second pass for opcode encoding done
[+] Assembling file ecc_curve_iram.s done in ecc_curve_iram.vhd
[+] Exported VHDL addresses of ecc_curve_iram.s done in ecc_addr.vhd (C header:  ecc_addr.h)
```

The build produces three VHDL files that need to be integrated in your project. These files are:

- ⟨`ecc_curve_iram.vhd`⟩ (this is the actual microcode memory)
- ⟨`ecc_addr.vhd`⟩
- ⟨`ecc_vars.vhd`⟩.

The build process is based on the assemblying of a few source files. The files assembled are the files (all contained in folder `hdl/common/ecc_curve_iram/asm_src/`), whose name is listed in the variable `PFX_SRC_FILES` of the ⟨`Makefile`⟩ local to folder `hdl/common/ecc_curve_iram/`. The assemblying process itself is done by the Python script ⟨`ipecc_assembler.py`⟩ in the same folder (this script is called during the execution of the `make` command).

5. **Import all the required VHDL source files into your simulator software**.

   The way you should do this depends of course of the simulation tool you're using. Please refer to table G.1 in p. 194 (Appendix-G) giving a comprehensive list of the IP VHDL files of the repository you need to import and compile into your simulator tool to perform behavorial similation (these are all the files marked with a ✔ sign in the *Simulation* column). Naturally, in folder `hdl/techno-specific/`, select only the subfolder (among `asic`, `xilinxa/series7`, `xilinxa/ultrascale` and `ialtera`) that fits your target/application.

   Note that the `ialtera` and `asic` cases are similar as they both correspond to a behabovial description of the multiplier-accumulator (files ⟨`macc_ialtera.vhd`⟩ and ⟨`macc_asic.vhd`⟩) and of the large shift-registers (files ⟨`large_shr_ialtera.vhd`⟩ and ⟨`large_shr_asic.vhd`⟩). This is due to the fact that during the last years the Intel-Altera company has severly restricted the way designers can access the low level primitives of the logic fabric in their chip.

   The top-level entity in the testbench is `ecc_tb` (file ⟨`sim/ecc_tb.vhd`⟩).

B.2 Format of the input test vector file

The simulation testbench provided with the IP (see file ⟨`ecc_tb.vhd`⟩ in folder `sim/`) expects to be «fed» with a file of a specific format containing data that describes the computations to submit to the IP and the result that is expected for each of these computations.

The input test-vector file is not expected to provide only entry/input values for the tests (like curve parameters and base point coordinates). The file **should also contain the result expected for each test** (e.g the coordinates of $[k]\mathcal{P}$, or the answer `true` or `false` of a boolean test). The simulation testbench will submit the test input data to the IP, run the expected command, then collect the result back from the IP and compare it with what is provided as the expected output in the input test-vector file (the test program of the real hardware works the same way). Remember that the *SageMath* script ⟨`generate-tests.sage`⟩ allows you to quickly generate an input test-vector file and that it contains an online documentation. Also mind the two files ⟨`ecc_vec_in.txt`⟩ and ⟨`std-curves-test-vectors.txt`⟩ provided in `sim/` folder than you can use as templates.

An example is worth a thousand words, so let's take a look at file ⟨`std-curves-test-vectors.txt`⟩ (folder `sim/` of the Git repository). This file is provided as a short example defining a few curves and tests all of cryptographic sizes, borrowing curve and point data from different ECC standardized protocols like the french curve FRP256v1 or the german *Brainpool curves*. Note that simulating tests with cryptographic sizes cna turn to be quote long. If you expect to simulate more rapidly, you might consider using instead the input test-vector file named ⟨`ecc_vec_in.txt`⟩ (also in `sim/` folder) that defines a serie of tests based on value `nn = 21`. Using small curves during behavorial simulation is not a bad idea, since what one usually hunts down in this phase of the design flow are bugs in the control logic, which are almost independent of the size of the data.

Here are the first 27 lines of file ⟨`std-curves-test-vectors.txt`⟩. We hope you'll find the syntax to be quite human-readable[1].

```
1  == NEW CURVE #0
2  # Name: Brainpool curve 160-bit
3  nn=160
4  p=0xe95e4a5f737059dc60dfc7ad95b3d8139515620f
5  a=0x340e7be2a280eb74e2be61bada745d97e8f7c300
6  b=0x1e589a8595423412134faa2dbdec95c8d8675e58
7  q=0xe95e4a5f737059dc60df5991d45029409e60fc09
8  == TEST [k]P #0.0
9  Px=0xbed5af16ea3f6a4f62938c4631eb5af7bdbcdbc3
10 Py=0x1667cb477a1a8ec338f94741669c976316da6321
11 k=0x5e98fab1e81df9fc17d528542f81c358dc7f91e6
12 kPx=0x244f843b08742d3e3c6f40934dbd16f27cc04b7d
13 kPy=0x50e31100239f46abb15eb93f77d4f888654782f2
14
15 == NEW CURVE #1
16 # Name: Brainpool curve 192-bit
17 nn=192
18 p=0xc302f41d932a36cda7a3463093d18db78fce476de1a86297
19 a=0x6a91174076b1e0e19c39c031fe8685c1cae040e5c69a28ef
20 b=0x469a28ef7c28cca3dc721d044f4496bcca7ef4146fbf25c9
21 q=0xc302f41d932a36cda7a3462f9e9e916b5be8f1029ac4acc1
22 == TEST [k]P #1.0
23 Px=0xc0a0647eaab6a48753b033c56cb0f0900a2f5c4853375fd6
24 Py=0x14b690866abd5bb88b5f4828c1490002e6773fa2fa299b8f
25 k=0xe0ed258a2778c759153d6243591938cc0ce6ac65af6ecd3b
26 kPx=0xa968ed0cc5e1f61998bf6cd912487e9d48f5984df9ca8c93
27 kPy=0x6765b83dee7136f8618881ddb70fe8fda77a46cc6ce40eba
```

The format can be defined as follows:

1. Empty lies are ignored.
2. Lines starting with the `'#'` character are considered as comments and are ignored.
3. A definition of a test must be preceeded by a definition of the curve it belongs to. The curve pertaining to the test is always the last curve that has been defined before this test.
4. **The definition of a curve** starts with the token `== NEW CURVE #` directly followed by a decimal integer number. Numbers here are just an ID, they need not be unique (though it obviously makes sense that they should be).
5. **The definition of a test** starts with the token `== TEST` concatenated with one of the seven following tokens each identifying a type of test among the seven allowed ones:
   - token `[k]P` identifies a $[k]\mathcal{P}$ (scalar multiplication) computation test
   - token `P+Q` identifies a $\mathcal{P} + \mathcal{Q}$ (point addition) computation test
   - token `[2]P` identifies a $[2]\mathcal{P}$ (point doubling) computation test

[1] In case you're wondering why we didn't use a standardized format like XML for the test vector file, this is because VHDL is not at all a language adapted to text parsing.

- token `-P` identifies a $-\mathcal{P}$ (point negate) computation test
- token `isP==Q` identifies a $\mathcal{P} == \mathcal{Q}$ («are points equal?») boolean test
- token `isP==-Q` identifies a $\mathcal{P} == -\mathcal{Q}$ («are points opposite?») boolean test
- token `isPoncurve` identifies a $\mathcal{P} \in \mathcal{C}$ («is point on curve?») boolean test

Following one of these seven identifiers must be a space, followed by the character `'#'`, directly followed by an identifier in the form of a decimal integer number, followed by a dot `'.'`, followed by a second decimal integer number. These numbers have no importance and are just here to allow you to uniquely identify each test. A good practice hence is to use for the first number the same one as the curve the test belongs to, and for the second number an incremental one. In case there's something wrong, the debug log (both in simulation and on the real hardware) will use these two numbers so that you know exactly which tests caused the trouble. In the example file above, there's only one test (a $[k]\mathcal{P}$ one) for each curve, that's why the test IDs are `0.0` (test 0 of curve 0), then `1.0` (test 0 of curve 1), etc.

6. Following the line with token `== NEW CURVE #` (introducing a new curve definition) must follow these five parameters, each on one different line, strictly in that order:

   - token `nn=`, direcly followed by the value of `nn` parameter (this obviously assumes that the dynamic prime size feature is on in your target design, or, in case it's not, that the value set here is the same as the static one the IP was synthesized with).
   - token `p=0x`, directly followed by the hexadecimal value of the prime p (definying the underlying field of the curve).
   - token `a=0x`, directly followed by the hexadecimal value of the curve parameter a.
   - token `b=0x`, directly followed by the hexadecimal value of the curve parameter b.
   - token `q=0x`, directly followed by the hexadecimal value of the order of the curve q. However, if you don't want intend to program a $[k]\mathcal{P}$ test for that curve that would ask for blinding (see below) then you can set a dummy value here (for cryptographic sizes, the order of the curve can be quite long to compute, so if you're generating tests in great number adn automatically, using e.g the script ⟨`generate-tests.sage`⟩ (folder `sage/`) you might consider not using blinding or limiting the value of `nn` above which you won't program blinding for $[k]\mathcal{P}$ tests. See the custom parameter `NN_LIMIT_COMPUTE_Q` at the begining of the script ⟨`generate-tests.sage`⟩ along with the online textual help provided with the parameters portion of the script.

     Note: For parameters p, a, b and q, the number of hexadecimal digits should match exactly the one expected by the value of `nn`, that is $\lceil \frac{nn}{4} \rceil$ (example: for `nn = 160`, exactly 40 hexadecimal digits are expected; same for values `nn = 159` down to `nn = 157`; now for `nn = 156` exactly 39 hexadecimal digits whould be expected).

   Now the format expected for the tests themselves depend on the type of the test. We only describe below the format for $[k]\mathcal{P}$ test.

   Note: For the remaining 6 other test types, we refer you to the example file ⟨`ecc_vec_in.txt`⟩ (in folder `sim/`) which contains not only a $[k]\mathcal{P}$ test but also one example for each of the six other types of tests – you should find interpreting the format to be straightforward.

7. **For $[k]\mathcal{P}$ tests:** following the line with token `== TEST [k]P #` must follow these parameters, each on one different line, strictly in that order:

   - token `Px=0x`, direcly followed by the hexadecimal value of the affine X-coordinate of base point ($x_{\mathcal{P}}$).
   - token `Py=0x`, direcly followed by the hexadecimal value of the affine Y-coordinate of base point ($y_{\mathcal{P}}$).
   - if $\mathcal{P}$ is the null point, replace the two lines `Px=` and `Py=` with a single one containing `P=0`.
   - token `k=0x`, directly followed by the value of the scalar k (even if it's null: token `k=0` is not allowed).

- *[only if blinding is expected for the current test:]* token `nbbld=`, directly followed by the decimal integer number of blinding bits expected (that is, the size in bits of the blinding scalar α that'll be used to compute $k' = k + \alpha q$). The number of bits must be greater (or equal) to 4, and smaller (or equal) to `nn - 4`.
- token `kPx=0x`, direcly followed by the hexadecimal value of the affine X-coordinate of point $[k]\mathcal{P}$ expected as result ($x_{[k]\mathcal{P}}$).
- token `kPy=0x`, direcly followed by the hexadecimal value of the affine Y-coordinate of point $[k]\mathcal{P}$ expected as result ($y_{[k]\mathcal{P}}$).
- if $[k]\mathcal{P}$ is expected to be the null point, replace the two lines `kPx=` and `kPy=` with a single one containing `kP=0`.

Note: try not to tamper with the format described above. Both the simulation testbench and the program testing the real hardware perform many sanity checks when parsing the input test-vector file, but any deviation from the description given hereabove should be considered as leading to undefined results.

B.3 Simulating with GHDL

The simulation using GHDL is straightforward. After following the tests 1-4 of §B.1, just change directory to `sim/` and type `make`. Here's what the compilation & elaboration log should look like:

From your Linux shell prompt

```
$ cd ${IPECC}/sim
$ make
[GHDL-LLVM] ../hdl/common/ecc_customize.vhd
[GHDL-LLVM] ../hdl/common/ecc_utils.vhd
[GHDL-LLVM] ../hdl/common/ecc_vars.vhd
[GHDL-LLVM] ../hdl/common/ecc_log.vhd
[GHDL-LLVM] ../hdl/common/ecc_pkg.vhd
[GHDL-LLVM] ../hdl/common/ecc_trng/ecc_trng_pkg.vhd
[GHDL-LLVM] ../hdl/common/ecc_software.vhd
[GHDL-LLVM] ecc_tb_vec.vhd
[GHDL-LLVM] ecc_tb_pkg.vhd
[GHDL-LLVM] ../hdl/common/mm_ndsp_pkg.vhd
[GHDL-LLVM] ../hdl/common/ecc_shuffle_pkg.vhd
[GHDL-LLVM] ../hdl/common/ecc_axi.vhd
[GHDL-LLVM] ../hdl/common/ecc_curve_iram/ecc_addr.vhd
[GHDL-LLVM] ../hdl/common/ecc_scalar.vhd
[GHDL-LLVM] ../hdl/common/ecc_curve.vhd
[GHDL-LLVM] ../hdl/common/ecc_curve_iram.vhd
[GHDL-LLVM] ../hdl/techno-specific/asic/large_shr_asic.vhd
[GHDL-LLVM] ../hdl/common/ecc_fp.vhd
[GHDL-LLVM] ../hdl/common/ecc_trng/es_trng_sim.vhd
[GHDL-LLVM] ../hdl/common/ecc_trng/ecc_trng_pp.vhd
[GHDL-LLVM] ../hdl/common/syncram_sdp.vhd
[GHDL-LLVM] ../hdl/common/fifo.vhd
[GHDL-LLVM] ../hdl/common/ecc_trng/ecc_trng_srv.vhd
[GHDL-LLVM] ../hdl/common/ecc_trng/ecc_trng.vhd
[GHDL-LLVM] ../hdl/common/ecc_fp_dram.vhd
[GHDL-LLVM] ../hdl/common/ecc_fp_dram_sh_linear.vhd
[GHDL-LLVM] ../hdl/common/virt_to_phys_ram_async.vhd
[GHDL-LLVM] ../hdl/common/ecc_fp_dram_sh_fishy_nb.vhd
[GHDL-LLVM] ../hdl/common/virt_to_phys_ram.vhd
[GHDL-LLVM] ../hdl/common/ecc_fp_dram_sh_fishy.vhd
[GHDL-LLVM] ../hdl/techno-specific/asic/macc_asic.vhd
[GHDL-LLVM] ../hdl/techno-specific/asic/maccx_asic.vhd
[GHDL-LLVM] ../hdl/common/sync2ram_sdp.vhd
[GHDL-LLVM] ../hdl/common/mm_ndsp.vhd
[GHDL-LLVM] ../hdl/common/ecc.vhd
../hdl/common/ecc.vhd:91:38:warning:  attribute for port "irq" must be specified in the entity [-Wspecs]
../hdl/common/ecc.vhd:93:43:warning:  attribute for port "irq" must be specified in the entity [-Wspecs]
[GHDL-LLVM] ecc_tb.vhd
[GHDL-LLVM] -e ecc_tb
../hdl/common/ecc_trng/ecc_trng.vhd:204:16:warning:  instance "t0" of component "es_trng" is not bound [-Wbinding]
../hdl/common/ecc_trng/ecc_trng.vhd:79:14:warning:  (in default configuration of ecc_trng(rtl))
ecc_tb.vhd:546:8:warning:  instance "pt0" of component "pseudo_trng" is not bound [-Wbinding]
ecc_tb.vhd:35:14:warning:  (in default configuration of ecc_tb(sim))

  Compilation & Elaboration completed.
  You can now run the simulation with this command line:

    $ ghdl-llvm -r ecc_tb -ieee-asserts=disable
```

As told at the end of the compilation log above, all you have to do is to run the simulation executable by hand, using command `ghdl-llvm -r ecc_tb -ieee-asserts=disable`:

From your Linux shell prompt

```
$ ghdl-llvm -r ecc_tb -ieee-asserts=disable
[        ecc.vhd ]:  Config:  nn = 528 (nn_dynamic ON), ...  debug OFF
[        ecc.vhd ]:  Config:  Microcode memory size:  512 opcodes ...  32 large-numbers
[        ecc.vhd ]:  Config:  TRNG fifo sizes:  raw=32768-bit, ...  32768 words of 5-bit
[     ecc_tb.vhd ]:  Out-of-reset
[     ecc_tb.vhd ]:  Waiting for init
[     ecc_tb.vhd ]:  Init done
[     ecc_tb.vhd ]:  Reading test-vectors from input file:  "/tmp/ecc_vec_in.txt"
[     ecc_tb.vhd ]:  ==== NEW CURVE #0
[     ecc_tb.vhd ]:  nn=21
[     ecc_tb.vhd ]:  p=0x1ce54b
[     ecc_tb.vhd ]:  a=0x0ec20f
[     ecc_tb.vhd ]:  b=0x1bb973
[     ecc_tb.vhd ]:  q=0x1ce256
[ ecc_scalar.vhd ]:  Entering state 'cst' (3225000000 fs)
[ ecc_scalar.vhd ]:  Returning to state 'idle' (86065000000 fs)
[ ecc_scalar.vhd ]:  Entering state 'cst' (86565000000 fs)
[ ecc_scalar.vhd ]:  Returning to state 'idle' (87285000000 fs)
[     ecc_tb.vhd ]:  **** NEW TEST [k]P #0.0
[     ecc_tb.vhd ]:  Px=0x00851a
[     ecc_tb.vhd ]:  Py=0x0a0e0f
[     ecc_tb.vhd ]:  k=0x1c0ac1
[     ecc_tb.vhd ]:  nbbld=14
[     ecc_tb.vhd ]:  Expecting result:  [k]P.x = 0x0acc93
[     ecc_tb.vhd ]:  Expecting result:  [k]P.y = 0x0e007f
[ ecc_scalar.vhd ]:  Returning to state 'idle' (88585000000 fs)
[     ecc_tb.vhd ]:  Acquired masking token:  0x17b198
[    ecc_axi.vhd ]:  k (as set by software) = 0x0001c0ac1
[    ecc_axi.vhd ]:  Mask of k (additive) = 0x6de3ffe2b
[    ecc_axi.vhd ]:  Masked value of k = 0x6de5c08ec
[ ecc_scalar.vhd ]:  Entering state 'set' (91545000000 fs)
[ ecc_scalar.vhd ]:  Entering state 'kp' (91555000000 fs)
[ ecc_scalar.vhd ]:  Entering substate 'checkoncurve' (91555000000 fs)
[ ecc_scalar.vhd ]:  Input point IS on curve, Entering substate 'blindinit' (96875000000 fs)
[ ecc_scalar.vhd ]:  Entering substate 'blindbit' (109045000000 fs)
[ ecc_scalar.vhd ]:  Blinding bits #0 ...  13
[ ecc_scalar.vhd ]:  Entering substate 'ssetup' (163605000000 fs)
[ ecc_scalar.vhd ]:  Entering substate 'joyecoz' (189945000000 fs)
[ ecc_scalar.vhd ]:  Processed scalar bits #2 ...  15
[ ecc_scalar.vhd ]:  Processed scalar bits #16 ...  31
[ ecc_scalar.vhd ]:  Processed scalar bits #32 ...  34
[ ecc_scalar.vhd ]:  Entering substate 'exits' (899435000000 fs)
[     ecc_fp.vhd ]:  [k]P.x = 0xacc93
[     ecc_fp.vhd ]:  [k]P.y = 0xe007f
[ ecc_scalar.vhd ]:  Output point IS on curve
[ ecc_scalar.vhd ]:  Returning to state 'idle' (935805000000 fs)
[ ecc_scalar.vhd ]:  Returning to substate 'idle' (935805000000 fs)
[ ecc_scalar.vhd ]:  PERF: 84370 clock cycles (935805000000 fs)
[     ecc_tb.vhd ]:  R_STATUS shows STATUS_ERR_UNKNOWN_REG error (935875000000 fs)
[     ecc_tb.vhd ]:  Read-back on AXI interface:  [k]P.x = 0x0acc93
[     ecc_tb.vhd ]:  Read-back on AXI interface:  [k]P.y = 0x0e007f
[     ecc_tb.vhd ]:  **** END TEST [k]P #0.0 - SUCCESSFULL: [k]P point coordinates ...  input test-vectors file.
        ...
[     ecc_tb.vhd ]:  **** NEW TEST isP==-Q #0.6
[     ecc_tb.vhd ]:  Px=0x123a44
[     ecc_tb.vhd ]:  Py=0x18f264
[     ecc_tb.vhd ]:  Qx=0x123a44
[     ecc_tb.vhd ]:  Qy=0x03f2e7
[     ecc_tb.vhd ]:  Expecting answer:  TRUE.
[ ecc_scalar.vhd ]:  Entering state 'pop' (1046235000000 fs)
[ ecc_scalar.vhd ]:  Returning to state 'idle' (1048395000000 fs)
[     ecc_tb.vhd ]:  **** END TEST isP==-Q #0.6 - SUCCESSFULL: RTL test answer matches ...  (both are TRUE).
[     ecc_tb.vhd ]:  End of testbench simulation (EOF) (1048505000000 fs)
[     ecc_tb.vhd ]:  Tests statistics:
[     ecc_tb.vhd ]:  ok = 7
[     ecc_tb.vhd ]:  nok = 0
[     ecc_tb.vhd ]:  total = 7
```

Hit `Ctrl-C` to abort the simulation.

B.4 Simulating with Vivado

To perform behavorial similation in Vivado (Xilinx-AMD targets) please refer to the tutorial in Appendix F, section *Simulation of the IP* in pp. 124 sq.

B.5 Simulation log

During behavioral simulation, the IP logs the details of all instructions of the microcode it executes in a dedicated file. This job is done by component `ecc_fp`. The log output file is the one to which parameter `simlogfile` points to in ⟨`ecc_customize.vhd`⟩.

Below is a short excerpt from the log file generated using ⟨`sim/ecc_vec_in.txt`⟩ (small curve tests with `nn = 21`) as input test-vector file:

```
....
1431    [0x0a1]    NNADD 0x00396bfa  (16 <- 16  +  24)  [211975000000 fs]
1432    [0x0a2]    NNSUB 0x001c86af  (22 <- 16  -  00)  [212145000000 fs]
1433    [0x0a3]    NNADD 0x001c86af  (16 <- 22  +  31)  [212315000000 fs]
1434    [0x0a4]      NOP                                 [212355000000 fs]
1435                STOP
1436
1437    .zadduL [0x0a5]
1438    [0x0a5]       JL 0x0a7                           [212445000000 fs]
1439    [0x0a7]   FPREDC             (26 <- 08  x  26)  [212525000000 fs]
1440    [0x0a8]   FPREDC             (08 <- 08  x  08)  [212685000000 fs]
1441              FPREDC 0x001a0546  (26             )  [213105000000 fs]
1442              FPREDC 0x00195b9d  (08             )  [213265000000 fs]
1443    [0x0a9]   FPREDC             (17 <- 16  x  16)  [213295000000 fs]
1444              FPREDC 0x000888ca  (17             )  [213875000000 fs]
1445    [0x0aa]   FPREDC             (09 <- 07  x  08)  [213925000000 fs]
1446    [0x0ab]    NNADD 0x00181c17  (20 <- 05  +  31)  [214185000000 fs]
1447    [0x0ac]    NNADD 0x002d0326  (21 <- 06  +  31)  [214355000000 fs]
1448    [0x0ad]   FPREDC             (07 <- 20  x  08)  [214425000000 fs]
1449              FPREDC 0x0025bd11  (09             )  [214565000000 fs]
1450              FPREDC 0x00150a8e  (07             )  [215015000000 fs]
1451    [0x0ae]    NNSUB 0xfff37e3c  (17 <- 17  -  07)  [215165000000 fs]
1452    [0x0af]    NNADD 0x002d48d2  (17 <- 17  +  24)  [215335000000 fs]
1453    [0x0b0]    NNSUB 0x00078bc1  (05 <- 17  -  09)  [215505000000 fs]
1454    [0x0b1]    NNADD 0x00078bc1  (05 <- 05  +  31)  [215675000000 fs]
1455    [0x0b2]    NNSUB 0x0010b283  (09 <- 09  -  07)  [215845000000 fs]
1456    [0x0b3]    NNADD 0x0010b283  (09 <- 09  +  31)  [216015000000 fs]
1457    [0x0b4]   FPREDC             (04 <- 21  x  09)  [216085000000 fs]
1458    [0x0b5]    NNSUB 0x000d7ecd  (08 <- 07  -  05)  [216345000000 fs]
1459    [0x0b6]    NNADD 0x000d7ecd  (08 <- 08  +  31)  [216515000000 fs]
1460    [0x0b7]   FPREDC             (06 <- 16  x  08)  [216585000000 fs]
1461              FPREDC 0x002113e9  (04             )  [216725000000 fs]
1462              FPREDC 0x0008189b  (06             )  [217175000000 fs]
1463    [0x0b8]    NNSUB 0xffe704b2  (06 <- 06  -  04)  [217325000000 fs]
1464    [0x0b9]    NNADD 0x0020cf48  (06 <- 06  +  24)  [217515000000 fs]
1465    [0x0ba]      RET (0x0a6)                         [217565000000 fs]
1466    [0x0a6]      NOP                                 [217605000000 fs]
1467                STOP
1468
1469    [VHD-CMP-SAGE] R0/R1 coordinates after ZADDU of BIT 3 (kap3 = 0,  kap'3 = 0)
1470    [VHD-CMP-SAGE]     @ 7   XR0 = 0x00150a8e
1471    [VHD-CMP-SAGE]     @ 4   YR0 = 0x002113e9
1472    [VHD-CMP-SAGE]     @ 5   XR1 = 0x00078bc1
1473    [VHD-CMP-SAGE]     @ 6   YR1 = 0x0020cf48
1474    [VHD-CMP-SAGE]     @ 26 ZR01 = 0x001a0546
1475
1476    .pre_zaddcL [0x0bb]
1477    [0x0bb]    NNSUB 0x000d7ecd  (21 <- 07  -  05)  [217825000000 fs]
1478    [0x0bc]    NNADD 0x000d7ecd  (21 <- 21  +  31)  [217995000000 fs]
1479    [0x0bd]    NNSUB 0xfff09982  (22 <- 21  -  00)  [218165000000 fs]
1480    [0x0be]    NNADD 0x000d7ecd  (21 <- 22  +  00)  [218335000000 fs]
....
```

Based on this snippet of execution log, we can describe the format as below (starting from the leftmost column):

- The first column gives the hexadecimal address in microcode memory from which the instruction opcode was fetched.

  When this address is detected as matching the entry point of one of the main routines whose execution is triggered by component `ecc_scalar`, this label is displayed on a previous separate line (see e.g line 1437 for label `.zadduL` (corresponding to opcode address `0x0a5`) and line 1476 for label `.pre_zaddcL` (corresponding to opcode address `0x0bb`).

  Now if you open source file ⟨`hdl/common/ecc_addr.vhd`⟩ and search for string `ZADDU` in it, you should find the line below (mind the binary and hexadecimal values at the end of the line). Remember that file ⟨`ecc_addr.vhd`⟩ is a automatically generated when building the IP microcode (when you type `make` in folder `hdl/common/ecc_curve_iram/`.

```
32   constant ECC_IRAM_ZADDU_ADDR : std_logic_vector(IRAM_ADDR_SZ - 1 downto 0) := "010100101"; -- 0xa5
```

If now you open source file ⟨`hdl/common/ecc_scalar.vhd`⟩ and follow the track of character string `ECC_IRAM_ZADDU_ADDR`, it'll in turn lead you to string `ZADDU_ROUTINE`, that you can see is latched several times into register `r.int.faddr`, which is the bus driven out by `ecc_scalar` into `ecc_curve` to set the address at which to start execution of a new portion of microcode. The static constant array `EXEC_ADDR` is used to that purpose, see for instance:

```
791   v.int.faddr := EXEC_ADDR(ZADDU_ROUTINE);
```

- The second column gives the mnemonic of the instrucion executed, e.g `FPREDC` (Montgomery multiplication) `NNADD`, etc.
- For sake of clarity, let's describe the fourth column before the third: in fouth column, the operands of the instruction appears in the form (taking the example of the addition `NNADD`):

$$\textit{Destination Operand} \leftarrow \textit{1}^{st}\ \textit{Source Operand} \ + \ \textit{2}^{nd}\ \textit{Source Operand}.$$

  As an example, line 1456 describes the execution of the addition of the large number at addres 9 with the large number at address 31, with the result stored also at address 9 (by the way this shows that instructions can operante *in-place*, meaning that the destination address can be identical to the address of one of he two source operands (who can also happen to be equal).

  Note that the symbol used for the `FPREDC` instruction is x but keep in mind that this is not a simple integer multiplication but rather a Montgomery one.

- The third column gives the arithmetical result of the execution of the instruction. As you can see, this concerns all arithmetical and logical instructions with the exception of `FPREDC`. This is because each `FPREDC` opcode is *posted* to a Montgomery hardware multiplier that carries out the multiplication asynchronously to the flow of execution. The result of the REDC operation hence appears later in the log: see for instance the `FPREDC` instruction in line 1457, whose operation is to do a Montgomery multiplication of large numbers at addresses 21 and 9, the result to be transferred in the large number of address 4. The result of this multiplication appeards four lines below (see line 1461), the result being large number `0x002113e9` (the address being recalled on the right of the arithmetical result value).

  For the `RET` instruction (return to jump-and-link instruction) the value indicated is the return address (e.g address `0x0a6` in line 1465, the instruction fetched after the `RET` instruction indeed being the opcode at address `0x0a6`, see line 1466). Likewise, for the jump-and-link instruction `JL` (see e.g line 1438) the value displayed is the jump address.

- The fifth column displays the simulation time at wich the event happened. For arithmetical operations, this corresponds to the clock cycle of the last burst of data transfer of the result of the opcodes into the memory of large numbers.

- Now as you can see sometimes in the current of the log the display changes, with lines starting with the flag `[VHD-CMP-SAGE]`. These are intermediate values of the 5 coordinates `XR0`, `YR0`, `XR1`, `YR1` and `ZR01` of the two sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$. This allows comparison to be made, using simple «grepping», with the *SageMath* tool which we used as the finite-precision software-proof of the RTL logic throughout the IPECC project. Please refer to the tutorial in Appendix-F, more precisely steps **21** to **25** (pp. 127 sq.)

Appendix C

Registers

Table C.1 below shows the register map of the IP. All registers in this table are mapped in the AXI address space at the offset shown in the center column, and are thus accessible to software. Registers that are only accessible in write mode are displayed in the left column, read-only registers are given in the right column.

	Write-only registers	Address offset	*Read-only* registers
(1)	W_CTRL	0x000	R_STATUS
	W_WRITE_DATA	0x008	R_READ_DATA
	W_RO_NULL	0x010	R_CAPABILITIES
	W_R1_NULL	0x018	R_HW_VERSION
(2)	W_PRIME_SIZE	0x020	R_PRIME_SIZE
(1)	W_BLINDING	0x028	*reserved*
	W_SHUFFLE	0x030	
	W_ZREMASK	0x038	
	W_TOKEN	0x040	
	W_IRQ	0x048	
	W_ERR_ACK	0x050	
	W_SMALL_SCALAR	0x058	
	W_SOFT_RESET	0x060	
	reserved	0x068 - 0x0f8	*reserved*
(3)	W_DBG_HALT	0x100	R_DBG_CAPABILITIES_0
	W_DBG_BKPT	0x108	R_DBG_CAPABILITIES_1
	W_DBG_STEPS	0x110	R_DBG_CAPABILITIES_2
	W_DBG_TRIG_ACT	0x118	R_DBG_STATUS
	W_DBG_TRIG_UP	0x120	R_DBG_TIME
	W_DBG_TRIG_DOWN	0x128	R_DBG_RAWDUR
	W_DBG_OP_WADDR	0x130	R_DBG_FLAGS
	W_DBG_OPCODE	0x138	R_DBG_TRNG_STATUS
	W_DBG_TRNG_CTRL	0x140	R_DBG_TRNG_RAW_DATA
	W_DBG_TRNG_CFG	0x148	R_DBG_FP_RDATA
	W_DBG_FP_WADDR	0x150	R_DBG_IRN_CNT_AXI
	W_DBG_FP_WDATA	0x158	R_DBG_IRN_CNT_EFP
	W_DBG_FP_RADDR	0x160	R_DBG_IRN_CNT_CRV
	W_DBG_CFG_XYSHUF	0x168	R_DBG_IRN_CNT_SHF
	W_DBG_CFG_AXIMSK	0x170	R_DBG_FP_RDATA_RDY
	W_DBG_CFG_TOKEN	0x178	R_DBG_EXP_FLAGS
	W_DBG_RESET_TRNG_CNT	0x180	R_DBG_TRNG_DIAG_0
	reserved	0x188	R_DBG_TRNG_DIAG_1
		0x190	R_DBG_TRNG_DIAG_2
		0x198	R_DBG_TRNG_DIAG_3
		0x1a0	R_DBG_TRNG_DIAG_4
		0x1a8	R_DBG_TRNG_DIAG_5
		0x1b0	R_DBG_TRNG_DIAG_6
		0x1b8	R_DBG_TRNG_DIAG_7
		0x1c0	R_DBG_TRNG_DIAG_8

Notes: (1) These are so-called *nominal* registers, meaning that they're always present, whatever the configuration of the IP.

(2) Register `W_PRIME_SIZE` is present only if you choose to set a runtime-modifiable size for prime field p, that is when `nn_dynamic = TRUE` in ⟨`ecc_customize.vhd`⟩. Register `R_PRIME_SIZE` always exists.

(3) These registers are only present in debug mode, that is when `debug = TRUE` in ⟨`ecc_customize.vhd`⟩.

Table C.1: IPECC register bank

As explained in the notes of table C.1, the upper black part, with address-offsets ranging from `0x000` to `0x060` are registers which always exist (they are always instanciated in the hardware) while the bottom part (interleaved gray and white lines) with address-offsets ranging from `0x100` to `0x1c0`, are the ones that only exist when the IP is instanciated in debug mode, meaning when `debug = TRUE` in ⟨`ecc_customize.vhd`⟩.

The number of bits that the IP samples from the address bus `AWADDR` (resp. `ARADDR`) of the *Write-address channel* (resp. of the *Read-address channel*) of the AXI interface to decode the register which is being accessed during an AXI transaction hence depends directly on `debug` parameter. When `debug` = `TRUE`, 6 bits are decoded, which are bits 8 down to 3. When `debug` = `FALSE`, 4 bits are decoded, which are bits 6 down to 3. Note that all address offsets in table C.1 are aligned on 8 bytes boundaries, therefore bits 2 down to 0 are never sampled by the IP. Furthermore no byte-lanes are made indivdually accessible inside the AXI 32-bit or 64-bit data words, but that's a restriction which is inherent to the lite version of the AXI4 protocols (see e.g [ARM11] §B1.1, p.-B1-122). Please refer to the inline ASCII figure in file ⟨`ecc_pkg.vhd`⟩ (section entitled «`AXI interface`») for more information on the IP decoding of the AXI address signals.

Nominal registers (as opposed to debug registers) are described hereafter one after the other:

- *write-only* registers are described in §C.1.1
- *read-only* registers are described in §C.1.2.

Debug registers are described in a separate section, see §C.2.

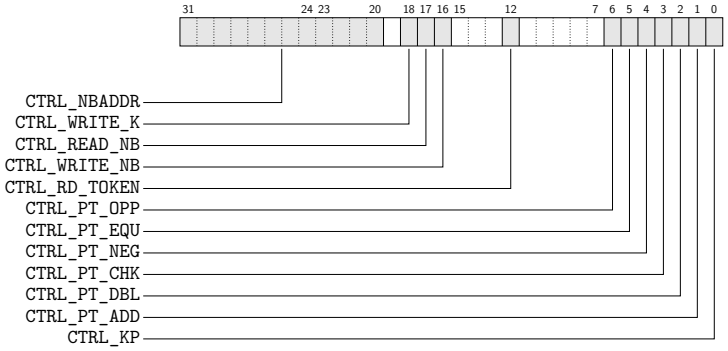
C.1 Nominal registers

By definition nominal registers are present in the design whatever the configuration of the IP, whereas debug registers are only present in debug (unsecure) mode (see §C.2).

C.1.1 Write-only registers

C.1.1.1 W_CTRL register (offset 0x000)

This is the main control register, through which curve and point operations and tests are programmed. All bits are active high.



Bit(s)	Name	Description	Operation details	Notes
0	CTRL_KP	Run $[k]\mathcal{P}$ computation	R1 ← [k]R1	4,5,7
1	CTRL_PT_ADD	Run two-point addition	R1 ← R0 + R1	
2	CTRL_PT_DBL	Run point doubling	R1 ← [2]R0	
3	CTRL_PT_CHK	Test if point is on curve	Test is made on R0	
4	CTRL_PT_NEG	Run point negate	R1 ← -R0	
5	CTRL_PT_EQU	Test if two points are equal	Test R0 == R1	
6	CTRL_PT_OPP	Test if two points are opposite	Test R0 == -R1	
12	CTRL_RD_TOKEN	To ask for token read		1,2
16	CTRL_WRITE_NB	To ask for a large number write		1,3
17	CTRL_READ_NB	To ask for a large number read		1,2
18	CTRL_WRITE_K	To ask for write of the scalar k		3
31–20	CTRL_NBADDR	Address of the large number to read or write		1,6
all others	-	*reserved*		

Notes:

(1) Field CTRL_NBADDR must be set valid at the same time that CTRL_WRITE_NB or CTRL_READ_NB is asserted.

(2) To read the token, software must assert both bits CTRL_RD_TOKEN and CTRL_READ_NB. Once this action is done through W_CTRL register, the next large number served by the IP (through software access(es) to register R_READ_DATA) will be the token.

(3) To write the scalar k, software must assert bits CTRL_WRITE_K and CTRL_WRITE_NB and also set CTRL_NBADDR field with the address of the scalar (which is 4).

Address 4 is overloaded (it can designate the address of the scalar and the address of XR0 (the X-coordinate of point $\mathcal{R}_0$).

Hence if bit CTRL_WRITE_K is not asserted when bit CTRL_WRITE_NB is asserted and field CTRL_NBADDR is set to value 4, then the write will be interpretated by the IP as targeting large number XR0 instead of the scalar.

(4) As a general rule, actions performed through W_CTRL register requires that software first read R_STATUS register to ensure that BUSY bit is deasserted.

(5) Since a total of 11 different actions can be programmed by writing to register W_CTRL which are all exclusive of another, software driver should consider asking simultaneously for two or more actions (e.g asserting CTRL_KP and CTRL_PT_ADD at the same time) as «leading to undefined behaviour». In fact the IP observes a strict ordering when parsing the value of W_CTRL written by software in order to ensure consistency of operations. But obviously software is advised not to tamper with the one-action-at-a-time rule.

(6) The number of bits of field `CTRL_NBADDR` actually depends on the number of large numbers that IP internal memory can store, which you can control statically with parameter `nblargenb` in ⟨`ecc_customize.vhd`⟩. The default value is 32 hence by default only bits 24 down to 20 are used. Becareful with modifying value of parameter `nblargenb`, don't modify it unless you really know what you're doing.

(7) Please refer to §E for complete pseudocode sequences showing how to program the different operations of the IP.

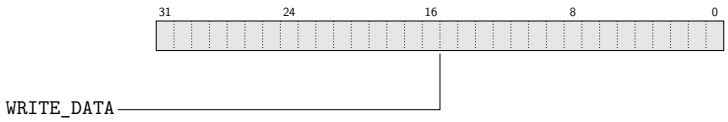
Possible errors: illegitimate accesses to `W_CTRL` may raise the following error(s) in `R_STATUS` register:

- `ERR_I_NOT_ENOUGH_RANDOM_WK`
- `ERR_I_RDNB_FBD`
- `ERR_I_TOKEN`
- `ERR_I_KP_FBD`
- `ERR_I_POP_FBD`
- `ERR_I_WREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.2 W_WRITE_DATA register (offset 0x008)

This register is used to transfer a chunk of a large number into the internal memory of the IP.



Bit(s)	Name	Description	Notes
31–0	WRITE_DATA	32-bit data chunck, part of a large number to be transferred to the IP internal memory)	1,2

Notes:

(1) By default the size of this register (and the size of the data chunk software can write one at a time) is 32-bit. It's 64-bit instead if the IP is configured in 64-bit mode, that is if `axi32or64 = 64` in ⟨`ecc_customize.vhd`⟩. Please refer to inline documentation of source file ⟨`ecc_customize.vhd`⟩ (look for parameter `axi32or64`, especially paragraph `"IMPORTANT NOTE"`).

(2) Please refer to §E for complete pseudocode sequences showing how to program the different operations of the IP.

See also:

- ☐ register `R_READ_DATA` (§C.1.2.2)

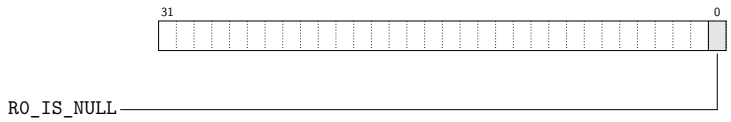
Possible errors: illegitimate access to `W_WRITE_DATA` may raise the following error(s) in `R_STATUS` register:

- ■ `ERR_I_WREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.3 W_R0_NULL register (offset 0x010)

This register is used to declare curve point $\mathcal{R}_0$ as being the null point, aka the point at infinity.



Bit(s)	Name	Description	Notes
0	R0_IS_NULL	When asserted (1) declares $\mathcal{R}_0$ as being the null point. When deasserted (0), declares $\mathcal{R}_0$ as different from the null point.	1–3
all others	-	*reserved*	

Notes:

(1) There exist no affine coordinates of the null point on an elliptic curve.

Hence when software declares point $\mathcal{R}_0$ to be the null point, the IP will disregard arithmetical values XR0 and YR0 of $\mathcal{R}_0$ affine coordinates that are stored in the IP internal memory of large numbers (at addresses 4 for XR0 and 5 for YR0).

On the other hand, when point $\mathcal{R}_0$ is declared not to be null, then the affine coordinates XR0 and YR0 stored in the IP internal memory are then considered to be the legitimate affine coordinates of a point on the curve.

(2) Starting a write cycle operation into either large number XR0 or large number YR0 immediately turns point $\mathcal{R}_0$ into a non-null point, even if register W_R0_NULL had been previously set with the bit R0_IS_NULL asserted high.

(3) As a conclusion, software should observe the following behaviour:

- To declare point $\mathcal{R}_0$ as being the null point, simply write register W_R0_NULL with bit R0_IS_NULL asserted high (and do not write large numbers XR0 nor YR0).
- To transfer a non-null $\mathcal{R}_0$ point to the IP, simply perform a write access to large numbers XR0 and YR0. Now optionnaly, and for sake of robustness, consistency and code readability, software driver might also write register W_R0_NULL with bit R0_IS_NULL deasserted (low).

See also:

- ☐ bits STATUS_R1_IS_NULL and STATUS_R1_IS_NULL in register R_STATUS
- ☐ register W_R1_NULL (§C.1.1.4)

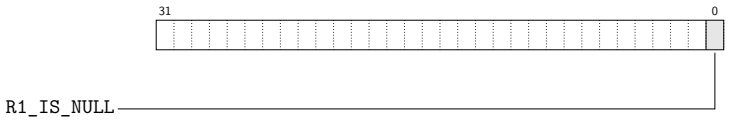
Possible errors: illegitimate accesses to W_R0_NULL may raise the following error(s) in R_STATUS register:

- ■ ERR_I_WREG_FBD

Please refer to §C.1.2.1 for description of R_STATUS register.

C.1.1.4 W_R1_NULL register (offset 0x018)

This register is used to declare curve point $\mathcal{R}_1$ as being the null point, aka the point at infinity.
If you've already read §C.1.1.3 about W_R0_NULL register, W_R1_NULL is exactly the same thing but for point $\mathcal{R}_1$.



Bit(s)	Name	Description	Notes
0	R1_IS_NULL	When asserted (1) declares $\mathcal{R}_1$ as being the null point. When deasserted (0), declares $\mathcal{R}_1$ as different from the null point.	1–3
all others	-	*reserved*	

Notes:

(1) There exist no affine coordinates of the null point on an elliptic curve.
Hence when software declares point $\mathcal{R}_1$ to be the null point, the IP will disregard arithmetical values XR1 and YR1 of $\mathcal{R}_1$ affine coordinates that are stored in the IP internal memory of large numbers (at addresses 4 for XR1 and 5 for YR1).
On the other hand, when point $\mathcal{R}_1$ is declared not to be null, then the affine coordinates XR1 and YR1 stored in the IP internal memory are then considered to be the legitimate affine coordinates of a point on the curve.

(2) Starting a write cycle operation into either large number XR1 or large number YR1 immediately turns point $\mathcal{R}_1$ into a non-null point, even if register W_R1_NULL had been previously set with the bit R1_IS_NULL asserted high.

(3) As a conclusion, software should observe the following behaviour:

- To declare point $\mathcal{R}_1$ as being the null point, simply write register W_R1_NULL with bit R1_IS_NULL asserted high (and do not write large numbers XR1 nor YR1).
- To transfer a non-null $\mathcal{R}_1$ point to the IP, simply perform a write access to large numbers XR1 and YR1. Now optionnaly, and for sake of robustness, consistency and code readability, software driver might also write register W_R1_NULL with bit R1_IS_NULL deasserted (low).

See also:

- ☐ bits STATUS_R1_IS_NULL and STATUS_R1_IS_NULL in register R_STATUS
- ☐ register W_R0_NULL (§C.1.1.3)

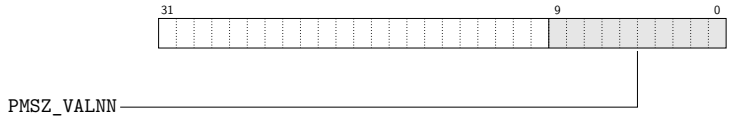
Possible errors: illegitimate accesses to W_R1_NULL may raise the following error(s) in R_STATUS register:

- ■ ERR_I_WREG_FBD

Please refer to §C.1.2.1 for description of R_STATUS register.

C.1.1.5 W_PRIME_SIZE register (offset 0x020)

This register only exists if *dynamic prime size* feature is set, that is if parameter nn_dynamic is set to TRUE in ⟨ecc_customize.vhd⟩. In this case its purpose is to dynamically set (i.e at runtime) the size of large numbers involved in all subsequent operations of the IP.



Bit(s)	Name	Description	Notes
9–0	PMSZ_VALNN	Dynamic value of nn, that is the bit size of all large numbers involved in curve and point operations (curve parameters, point coordinates, etc)	1–3
all others	-	*reserved*	

Notes:

(1) The formal definition for the size nn of large numbers (and therefore the value that should be written in this register) is:

$$\mathtt{nn} \;=\; \max(\lceil \log_2(p) \rceil\,, \lceil \log_2(q) \rceil)$$

where p is the prime number defining the underlying field in the definition of the cruve and q is the order of the curve, that is the number of its points. The Hasse theorem (see e.g [HMV04] §3.1.3, p. 82) indeed states that the number q of points of an elliptic curve is such that:

$$p + 1 - 2\sqrt{p} \;\leq\; q \;\leq\; p + 1 + 2\sqrt{p}$$

and therefore the binary representation of q can exceed that of p of one bit. Software must take that fact into consideratioh when writing register W_PRIME_SIZE otherwise $[k]\mathcal{P}$ values computed by the IP when blinding countermeasure is on will be wrong.

(2) The IP ensures that the dynamic value of nn set at runtime with this register is less than or equal to the static one the IP was synthesized with (the one set in ⟨ecc_customize.vhd⟩). Trying to violate this rule will raise error ERR_I_NNDYN in R_STATUS register.

(3) The number of bits implemented (synthesized) in field PMSZ_VALNN is actually the number of bits in binary representation of the static value of nn. Here 10 bits are shown (9–0) which correspond to nn = 521, as this is the greatest value one can find nowadays in elliptic curves based cryptographic protocols.

See also:

- ☐ register R_PRIME_SIZE (§C.1.2.5)
- ☐ register W_SMALL_SCALAR (§C.1.1.12)

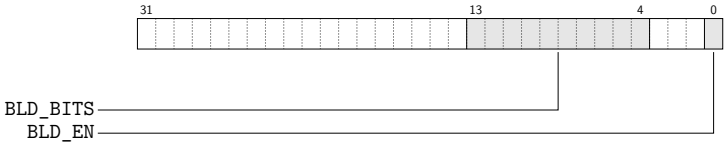
Possible errors: illegitimate accesses to W_PRIME_SIZE may raise the following error(s) in R_STATUS register:

- ■ ERR_I_WREG_FBD
- ■ ERR_I_NNDYN

Please refer to §C.1.2.1 for description of R_STATUS register.

C.1.1.6 W_BLINDING register (offset 0x028)

This register is used to enable and configure the blinding SCA countermeasure.



Bit(s)	Name	Description	Notes
0	BLD_EN	When asserted (1) enables blinding for subsequent $[k]\mathcal{P}$ computations. When deasserted (0) disables blinding.	
13–4	BLD_BITS	Number of bits of the random blinding scalar α	1–2
all others	-	*reserved*	

Notes:

(1) Blinding countermeasure being defined as drawing a random large number α and computing $[k + \alpha q]\mathcal{P}$ instead of $[k]\mathcal{P}$ (in order to mask scalar k), field BLD_BITS is defined as the bit width of parameter α (the blinding scalar) that is $\lceil \log_2(\alpha) \rceil$.

(2) The value set for the size of the blinding scalar must follow at any time:

$$4 \leq \lceil \log_2(\alpha) \rceil \ \leq \mathtt{nn}$$

where nn here denotes the dynamic size of large numbers. The IP enforces that rule and will raise the error ERR_I_BLN whenever software violates it.

See also:

- ☐ register W_PRIME_SIZE

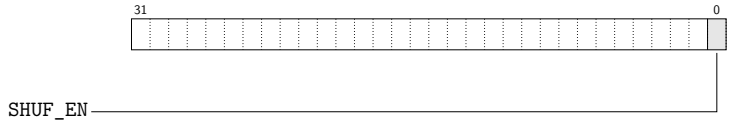
Possible errors: illegitimate accesses to W_BLINDING may raise the following error(s) in R_STATUS register:

- ■ ERR_I_WREG_FBD
- ■ ERR_I_BLN

Please refer to §C.1.2.1 for description of R_STATUS register.

C.1.1.7 W_SHUFFLE register (offset 0x030)

This register is used to enable and configure the shuffling SCA countermeasure.



Bit(s)	Name	Description	Notes
0	SHUF_EN	When asserted (1) enables shuffling. When deasserted (0) disables shuffling.	1
all others	-	*reserved*	

Notes:

(1) In debug (unsecure) mode, shuffling can be enabled or disabled according to the wishes of the software driver, and naturally if `shuffle_type` was set in ⟨`ecc_customize.vhd`⟩ to a value other than `none`.

In production (secure) mode however, a minimum security rule is enforced based on the static configuration of the IP:

- shuffling cannot be disabled if it was statically forced (`shuffle = TRUE` in ⟨`ecc_customize.vhd`⟩);
- shuffling can be enabled even if it was not statically forced (`shuffle = FALSE` and `shuffle_type != none` in ⟨`ecc_customize.vhd`⟩).

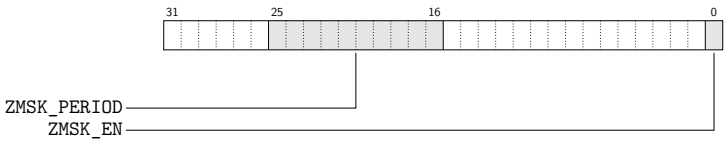
Possible errors: illegitimate accesses to `W_SHUFFLE` may raise the following error(s) in `R_STATUS` register:

- `ERR_I_WREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.8 W_ZREMASK register (offset 0x038)

This register is used to enable and configure the «Z-remask» SCA countermeasure.



Bit(s)	Name	Description	Notes
0	ZMSK_EN	When asserted (1) enables Z-remask, when disabled (0) disables Z-remask.	XXX
25–16	ZMSK_PERIOD	Period (expressed in number of bits of the scalar) at which jacobian coordinates of sensible points $\mathcal{R}_0$ and $\mathcal{R}_1$ are re-randomized	1-3
all others	-	*reserved*	

Notes:

(1) Basically the IP always randomize the coordinates of the base point $\mathcal{P}$ before starting any $[k]\mathcal{P}$ computation. The «Z-remask» countermeasure is offered in order to allow for a hardening of that basic countermeasure.

By asserting bit `ZMSK_EN` and setting the appropriate value in field `ZMSK_PERIOD`, software driver can impose the IP to randomize the jacobian coordinates (X:Y:Z) for both sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$ on a periodic base.

(2) The value that must be set in field `ZMSK_PERIOD` is the value of the desired period **minus 1.** For instance, if `ZMSK_PERIOD` is set to 2, the coordinates will be randomized after each triplet of consecutive bits parsed in the scalar (e.g 85 times if `nn = 256`). Obviously the smaller the value set in `ZMSK_PERIOD`, the greater security level achieved, but also the greater the performance loss. A value of 0 means a re-randomization after each bit of the scalar.

(3) In debug (unsecure) mode, «Z-remask» countermeasure can be enabled or disabled according to the wishes of the software driver. In production (secure) mode however, a minimum security rule is enforced based on the static configuration of the IP:

- Z-remask cannot be disabled if it was statically forced (`zremask > 0` in ⟨`ecc_customize.vhd`⟩);
- The period for Z-remask cannot be increased as compared to what it was set statically – hence the value set in `Z_REMASK` register at runtime must be less or equal to the value of parameter `zremask` -1 in ⟨`ecc_customize.vhd`⟩.

  Example: if `zremask = 12` in ⟨`ecc_customize.vhd`⟩, this means a period of 12 bits was configured statically, in which case software won't be authorized to set a value larger than 11 strictly (only valid values will be 0 to 11 included).

  Violating these rules will make the IP issue error `ERR_I_ZREMASK`.
- Z-remask can be enabled even if it was not statically forced (`zremask = 0` in ⟨`ecc_customize.vhd`⟩).

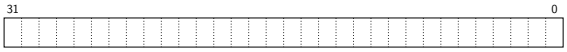
Possible errors: illegitimate accesses to `XXX` may raise the following error(s) in `R_STATUS` register:

- `ERR_I_WREG_FBD`
- `ERR_I_ZREMASK`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.9 `W_TOKEN` register (offset `0x040`)

This register is used by software to inform the IP that it whishes to read the random masking token for next $[k]\mathcal{P}$ computation.



Bit(s)	Name	Description	Notes
all	-	*reserved*	1–3

Notes:

(1) The token feature consists for the IP in first drawing a random large number (called the token) that the software must read before programming a new $[k]\mathcal{P}$ computation, then, once the following computation is completed, masking both affine coordinates of $[k]\mathcal{P}$ that are returned to software with the token. The masking is linear (it's a bitwise XOR).

This is a defense-in-depth countermeasure. Even if it's quite weak by itself (the token being transferred as a plain value between the IP and the software) it allows to protect the result of $[k]\mathcal{P}$ computation from software eavesdropping when $[k]\mathcal{P}$ result is used in a Diffie-Hellman exchange (intermediate semi-key or final one). The token feature complicates the work of malevolent software (one that is not legitimate to acquire the $[k]\mathcal{P}$ result) by forcing it to spy communications between the IP and the legitimate driver twice (before programming the $[k]\mathcal{P}$ computation and after) rather than just once (after the computation). As several milliseconds may be spent between the start and the end of computation, it's not necessarily trivial for malicious software to accomplish a two-step attack.

(2) This register is not divided in any bit/field because the sole write to this register is what is decoded by the IP (written value is meaningless, and software can use any value e.g 0).

The software driver must write-access register `W_TOKEN` to inform the IP that it is ready to accept a new random token. Then it must poll register `R_STATUS` until it shows that its `BUSY` bit is deasserted (meaning that the token is avaiable for read). Note that not only the bit `BUSY` will be asserted during the process of generating the token, but also the bit `STATUS_TOKEN_GEN`. Then the software driver must perform a read cycle of a large number through register `W_CTRL` with bit `CTRL_RD_TOKEN` asserted (and whatever the address set in field `CTRL_NBADDR_LSB`). Then software must program a $[k]\mathcal{P}$ computation, and when reading back the coordinates `XR1` and `YR1` of the result point, unmask them with the token it was previously given.

(3) In production (secure) mode, the token feature is always present and cannot be deactivated. In debug (unsecure) mode, it can be disabled using debug register `W_DBG_CFG_TOKEN`.

See also:

- ☐ register `W_DBG_CFG_TOKEN` (§**??**)
- ☐ bit `CTRL_RD_TOKEN` in `W_CTRL` register (§C.1.1.1).

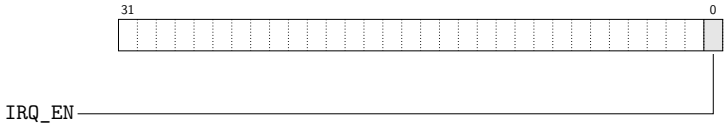
Possible errors: illegitimate accesses to `W_TOKEN` may raise the following error(s) in `R_STATUS` register:

- ■ `ERR_I_TOKEN`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.10 `W_IRQ` register (offset `0x048`)

This register is used to configure generation of interrupt requests by the IP upon completion of operations programmed by software.



Bit(s)	Name	Description	Notes
0	`IRQ_EN`	When asserted (1) enables IRQ generation. When disabled (0) disables IRQ generation	1–2
all others	-	*reserved*	

Notes:

(1) An interrrupt request can be raised by the IP to signal three different kind of events:

- **Completion of a $[k]\mathcal{P}$ operation** previously programmed by sofware. This is probably the most justified situation where software should use IRQs, due to the delay in computing the scalar multiplication, which tends to disqualify polling.
- **Completion of any other point operation** that $[k]\mathcal{P}$ that was previously programmed by software, including: $[k]\mathcal{P}$ computation, point addition, point doubling, point negate ($-\mathcal{P}$) along with logical tests (*is the point on curve*, *are the two points equal* and *are the two points opposite*).
- **Completion of Montgomery constants computation.** This is not a direct operation programmed by software. Computation of Montgomery constants happens automatically each time software has completed a write cycle of large number p (see §E.2 for how software performs write cycle of large numbers).

(2) Any IRQ generated by the IP consists in asserting high its `irq` top-level output signal for 4 cycles of the main clock (which is also the clock of the AXI interface). Hence the IRQ should be considered an **edge-sensitive** one (aka edge-triggered, as opposed to level-sensitive aka level-triggered). This means that the IRQ event is defined by the rising edge of `irq`, not by its level being high.

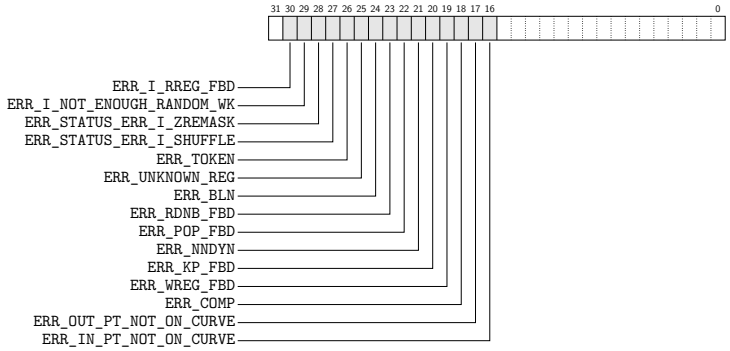
Possible errors: illegitimate accesses to `W_IRQ` may raise the following error(s) in `R_STATUS` register:

- `ERR_I_WREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.11 `W_ERR_ACK` register (offset `0x050`)

This register is used to acknowledge the different possible errors raised by the IP in `R_STATUS` register. All bits are active high.



Bit(s)	Name	Description	Notes
16	`ERR_IN_PT_NOT_ON_CURVE`	Each of these bits has an equivalent homonym bit in `R_STATUS` register standing at exactly the same bit position, and each of these bits (the ones in `W_ERR_ACK`) has the effect, when written with a high (1) value, to acknowledge the error signaled by its equivalent bit being positioned to 1 in `R_STATUS`.	–
17	`ERR_OUT_PT_NOT_ON_CURVE`		
18	`ERR_COMP`		
19	`ERR_WREG_FBD`		
20	`ERR_KP_FBD`		
21	`ERR_NNDYN`		
22	`ERR_POP_FBD`		
23	`ERR_RDNB_FBD`		
24	`ERR_BLN`		
25	`ERR_UNKNOWN_REG`		
26	`ERR_TOKEN`		
27	`ERR_STATUS_ERR_I_SHUFFLE`		
28	`ERR_STATUS_ERR_I_ZREMASK`		
29	`ERR_I_NOT_ENOUGH_RANDOM_WK`		
30	`ERR_I_RREG_FBD`		
all others	-	*reserved*	

See also:

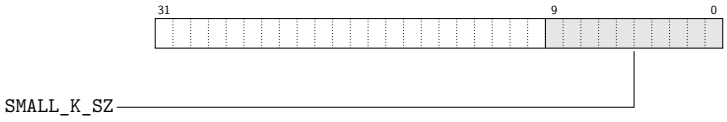
- ☐ register `R_STATUS` (§C.1.2.1)

Possible errors:

none

C.1.1.12 W_SMALL_SCALAR register (offset 0x058)

This register is used as a one-shot indicator by software that it wants to program a $[k]\mathcal{P}$ computation with a scalar of small size.



Bit(s)	Name	Description	Notes
9–0	SMALL_K_SZ	Number of least significant bits in the scalar k that IP should restrict to parse when computing next $[k]\mathcal{P}$	1–5
all others	-	*reserved*	

Notes:

(1) The purpose of this register is to allow software to perform fast $[k]\mathcal{P}$ computations when it is known that the scalar is of small size **without modifying the current value of** `nn`. For instance, software can use this feature to compute $[3]\mathcal{P}$ or to assess if the order of a point is small.

(2) The one-shot property means that the value written in `SMALL_K_SZ` will be considered only once by the IP (for the next $[k]\mathcal{P}$ computation to come) and then forgotten. After next $[k]\mathcal{P}$ computation, `nn` parameter will be again the reference size for the scalar transmitted by software (whether it's the static or the dynamic value).

(3) The value of `SMALL_K_SZ` must be greater or equal to 3 and smaller or equal to current value of `nn` (whether it's the static value of the dynamic one).

(4) The number of bits implemented (synthesized) in field `SMALL_K_SZ` is actually the number of bits in binary representation of the static value of `nn`. Here 10 bits are shown (9–0) but that's just an example that would fit e.g `nn` = 521.

(5) The value written into `W_SMALL_SCALAR` doesn't affect the value read from `R_PRIME_SIZE`.

See also:

- ☐ register `W_PRIME_SIZE` (§C.1.1.5)
- ☐ register `R_PRIME_SIZE` (§C.1.2.5)

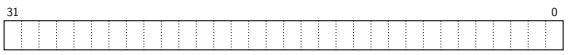
Possible errors: illegitimate accesses to `W_SMALL_SCALAR` may raise the following error(s) in `R_STATUS` register:

- ■ `STATUS_ERR_I_WREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.1.13 `W_SOFT_RESET` register (offset `0x060`)

This register is used to perform a software reset of the IP.



Bit(s)	Name	Description	Notes
all	-	*reserved*	

Notes:

(1) Writing to this register causes what is called a software reset: the entire IP logic will be reset, exactly like what would happen when the top-level input signal `s_axi_aresetn` is asserted, **with the exception of the AXI interface:** component `ecc_axi` is not impacted by a software reset.

(2) This register is not divided in any bit/field because the sole write to this register is what is decoded by the IP (written value is meaningless, and software can use any value e.g 0).

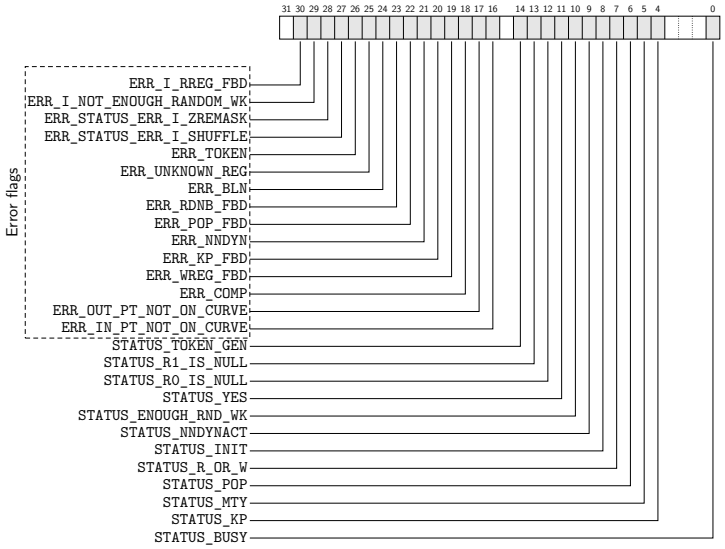
Possible errors:

none.

C.1.2 Read-only registers

C.1.2.1 R_STATUS register (offset 0x000)

This is the main status register, giving information on current readiness or business of the IP, result availability and/or operation status on pending or completed operations, and errors. All bits are active high.



Bit(s)	Name	Description	Notes
0	STATUS_BUSY	IP is busy	1,2
4	STATUS_KP	IP is busy computing a $[k]\mathcal{P}$	
5	STATUS_MTY	IP is busy computing Montgomery constants	
6	STATUS_POP	IP is busy computing a point operation (other than $[k]\mathcal{P}$) or a point test	
7	STATUS_R_OR_W	IP is busy transferring a chunk of a large number written by software into its internal memory, or transferring a chunk of a large number asked by a software-read from its internal memory	
8	STATUS_INIT	IP is busy performing initial post-reset operation	1–3
9	STATUS_NNDYNACT	IP is currently updating some internal signals to take in account the new value of `nn` (written by sofware in register `W_PRIME_SIZE`)	1,2
10	STATUS_ENOUGH_RND_WK	When asserted (1) indicates that enough randomness has been collected to accept a new scalar from software	4
11	STATUS_YES	Answer to the last logical point test (`1 = YES`, `0 = NO`)	5
12	STATUS_R0_IS_NULL	Indicates that point $\mathcal{R}_0$ is currently the null point	6
13	STATUS_R1_IS_NULL	Indicates that point $\mathcal{R}_1$ is currently the null point	
14	STATUS_TOKEN_GEN	The IP is busy generating internally the random token	1,2
16	ERR_IN_PT_NOT_ON_CURVE	Each of these bits has an equivalent homonym bit in `W_ERR_ACK` register standing at exactly the same bit position, and each of these bits (the ones in `W_ERR_ACK`) has the effect, when written with a high (1) value, to acknowledge the error signaled by its equivalent bit being positioned to 1 in `R_STATUS`.	–
17	ERR_OUT_PT_NOT_ON_CURVE		
18	ERR_COMP		
19	ERR_WREG_FBD		
20	ERR_KP_FBD		
21	ERR_NNDYN		
22	ERR_POP_FBD		
23	ERR_RDNB_FBD		
24	ERR_BLN		
25	ERR_UNKNOWN_REG		
26	ERR_TOKEN		
27	ERR_STATUS_ERR_I_SHUFFLE		
28	ERR_STATUS_ERR_I_ZREMASK		
29	ERR_I_NOT_ENOUGH_RANDOM_WK		
30	ERR_I_RREG_FBD		
all others	-	*reserved*	

Notes:

(1) When STATUS_BUSY is high, the IP is busy and can't accept any further command. When the STATUS_BUSY is low, the IP is in idle state and can accept the next command from software.

(2) **Important: software should use STATUS_BUSY as the sole indicator as to whether or not the IP is currently busy.** Bits 4–9 (STATUS_KP , STATUS_MTY , STATUS_POP , STATUS_R_OR_W , STATUS_INIT , STATUS_NNDYNACT) and bit 14 (STATUS_TOKEN_GEN) are provided as simple indicators to complete the information provided by the bit STATUS_BUSY .

(3) Currently the IP performs no particular job upon reset, so STATUS_INIT is asserted high almost instantly when out-of-reset. The purpose of this register is reserved for possible future adjustments or for your personal modifications of the IP (e.g if you want the IP to wait for an output event, or wait until a specific amount of randomness is available, etc).

(4) In production (secure) mode, the IP masks the scalar as soon as it's received from the software, on the fly as it's being transferred into its internal memory. Bit STATUS_ENOUGH_RND_WK gives an indication as to whether enough randomness is available in the IP internal random buffers to perform this masking on the fly, or not. Trying to initiate a write-cycle on large number k (with register W_CTRL) when this bit is deasserted (0) will raise error flag ERR_I_NOT_ENOUGH_RANDOM_WK and cancel the write-cycle. It's hence more prudent to poll this bit before actually intiating a write-cycle of the scalar k.

In debug (unsecure) mode, this masking coutermeasure can be disabled using register W_DBG_CFG_AXIMSK .

(5) Bit STATUS_YES gives the answer to the last logical point test (among: *is point on curve*, *are two points equal*, *are two points opposite*).

(6) The two bits STATUS_R0_IS_NULL and STATUS_R1_IS_NULL may denote the result of the last point operation, however they will also directly reflect what software driver itself writes in registers W_R0_NULL and W_R1_NULL .

See also:

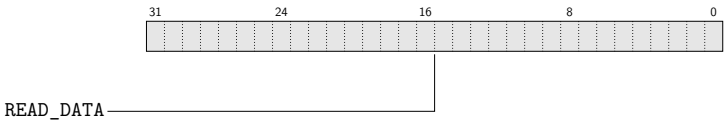
- ☐ register W_ERR_ACK (§C.1.1.11)
- ☐ register W_CTRL (§C.1.1.1)
- ☐ register W_TOKEN (§C.1.1.9)
- ☐ register W_R0_NULL (§C.1.1.3)
- ☐ register W_R1_NULL (§C.1.1.4)

Possible errors:

none.

C.1.2.2 R_READ_DATA register (offset 0x008)

This register is used to store each chunk of a large number from the internal memory to software.



Bit(s)	Name	Description	Notes
31–0	READ_DATA	32-bit data chunck, part of a large number transferred from the IP internal memory for software to read)	1–3

Notes:

(1) By default the size of this register (and the size of the data chunk software can read one at a time) is 32-bit. It's 64-bit instead if the IP is configured in 64-bit mode, that is if `axi32or64 = 64` in ⟨`ecc_customize.vhd`⟩. Please refer to inline documentation of source file ⟨`ecc_customize.vhd`⟩ (look for parameter `axi32or64`, especially paragraph `"IMPORTANT NOTE"`).

(2) Please refer to §E for complete pseudocode sequences showing how to program the different operations of the IP.

See also:

- ☐ register `W_WRITE_DATA` (§C.1.1.2)

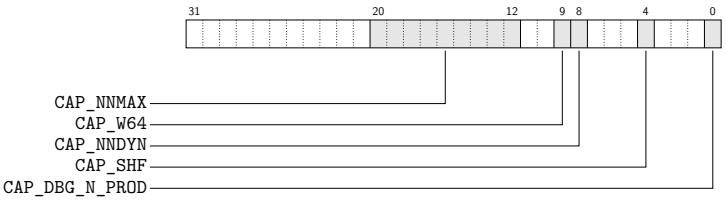
Possible errors: illegitimate accesses to `R_READ_DATA` may raise the following error(s) in `R_STATUS` register:

- ■ `ERR_I_RREG_FBD`

Please refer to §C.1.2.1 for description of `R_STATUS` register.

C.1.2.3 R_CAPABILITIES register (offset 0x010)

This register gives general information on the configuration of the IP. All bits are active high.



Bit(s)	Name	Description	Notes
0	CAP_DBG_N_PROD	When asserted (1) indicates the IP was synthesized in debug (unsecure) mode, otherwise in production (secure) mode	–
4	CAP_SHF	When asserted (1) indicates a shuffling method was synthesized	1
8	CAP_NNDYN	When asserted (1) indicates the dynamic prime size feature is available	2
9	CAP_W64	When asserted (1) indicates the IP AXI interface is a 64-bit one, otherwise it's a 32-bit one	3
20–12	CAP_NNMAX	This field gives the static value of nn	2
all others	-	*reserved*	

Notes:

(1) Bit CAP_SHF being asserted means that the IP was synthesized with parameter shuffle_type set to a value other than none in ⟨ecc_customize.vhd⟩, meaning that one permutation logic was synthesized among the three available. In this situation, the software driver will be allowed to activate shuffling (through W_SHUFFLE register) whatever the configuration (debug or non-debug) the IP was synthesized with.

(2) Bit CAP_NNDYN being asserted means that the IP was synthesized with dynamic prime size feature on (parameter nn_dynamic = TRUE in ⟨ecc_customize.vhd⟩. Software driver is thus allowed to modify dynamically value of parameter nn. The maximum value of nn that can be set at runtime is given by field CAP_NNMAX.

(3) Bit CAP_W64 means that the IP is configured with a 64-bit AXI interface (parameter axi32or64 = 64 in ⟨ecc_customize.vhd⟩). This impacts the size of registers W_WRITE_DATA and R_READ_DATA.

See also:

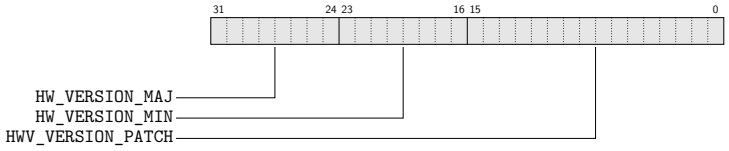
- ☐ register W_SHUFFLE (§C.1.1.7)
- ☐ register W_PRIME_SIZE (§C.1.1.5)
- ☐ register R_PRIME_SIZE (§C.1.2.5)
- ☐ register W_WRITE_DATA (§C.1.1.2)
- ☐ register R_READ_DATA (§C.1.2.2)
- ☐ register W_SMALL_SCALAR (§C.1.1.12)

Possible errors:

none

C.1.2.4 R_HW_VERSION register (offset 0x018)

This register gives the three hardware version numbers of the IP (major, minor & patch, e.g 1.2.26).



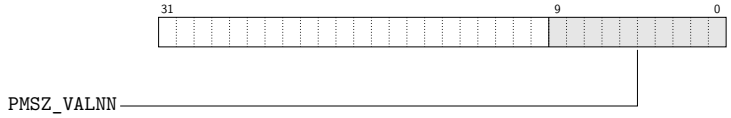
Bit(s)	Name	Description	Notes
31–24	HW_VERSION_MAJ	Major version number	–
23–16	HW_VERSION_MIN	Minor version number	
15–0	HW_VERSION_PATCH	Patch version number	

Possible errors:

none

C.1.2.5 R_PRIME_SIZE register (offset 0x020)

This register gives the current value of parameter `nn` (large numbers size).



Bit(s)	Name	Description	Notes
9–0	PMSZ_VALNN	Dynamic value of `nn`, that is the bit size of all large numbers involved in curve and point operations (curve parameters, point coordinates, etc)	1
all others	-	*reserved*	

Notes:

(1) If the *dynamic prime size* feature is on (`nn_dynamic = TRUE` in ⟨`ecc_customize.vhd`⟩) this field gives the current size of large numbers, that is the dynamic value of `nn`, that software can modify using register `W_PRIME_SIZE`.

If the *dynamic prime size* feature is off (`nn_dynamic = FALSE`) in ⟨`ecc_customize.vhd`⟩) this field gives the static value of `nn`.

See also:

- ☐ register `W_PRIME_SIZE` (§C.1.1.5)
- ☐ register `W_SMALL_SCALAR` (§C.1.1.12)

Possible errors:

none

C.2 Debug registers

C.2.1 Debug write-ony registers

(Redaction in progress, sorry)

C.2.2 Debug read-ony registers

(Redaction in progress, sorry)

Appendix D

Organization of internal memory of large numbers

D.1 Parameters influencing the organization of the memory (nn, ww, multwidth, w, n)

The memory of large numbers by default stores 32 large numbers[1]. Each of these large number is made of data words (or limbs) of ww bits each.

- For FPGA technologies (techno /= asic) the value of ww is automatically set according to the bit-width of the input operands of DSP-blocks. You can change this if you want by simply editing function set_ww from package ecc_utils (file ⟨ecc_utils.vhd⟩) **but** the value must stay below the limit of the bit-width of the input operands of the DSP block for your target technology[2].
- For ASIC technology (techno = asic) you can select explicitly the value you want using parameter multwidth. This value will be used both for the input operands of the multipliers-accumulators inside the Montgomery multipliers and for the bit-width of the words of the memory of large numbers.

Note: Parameter multwidth is only used when techno = asic. It is ignored for FPGA technologies.

Based on the values of nn and ww, package ecc_pkg derives two secondary parameters called w and n (also used throughout VHDL code):

- Parameter w is defined as the number of ww-bit words required to store a whole large number of size (nn+4)-bit in ecc_fp_dram:

$$\mathtt{w} = \left\lceil \frac{\mathtt{nn} + 4}{\mathtt{ww}} \right\rceil$$

Example values: $\begin{cases} \mathtt{nn}=256,\ \mathtt{ww}=17 \rightarrow & \mathtt{w}=16 \\ \mathtt{nn}=192,\ \mathtt{ww}=17 \rightarrow & \mathtt{w}=12 \\ \mathtt{nn}=320,\ \mathtt{ww}=16 \rightarrow & \mathtt{w}=21 \end{cases}$

The reason for adding four extra bits to the size of large numbers nn is twofold:

(1) As explained in §3.4.6, finite field arithmetic involved in $[k]\mathcal{P}$ computation use Montgomery representation with $R > 4p$ (R being a power of 2) which avoids the conditional subtraction by p at the end of each REDC operation (see lines 5-7 of algorithm 4 in §3.4.3). This in turn requires to be able to encode number R, as it will serve as input to the binary extended gcd algorithm (see [MVOV96], [HMV04], [CFA+12]) to compute the first Montgomery constant $-p^{-1} \mod R$ (see §3.4.2). Number R, which by definition

[1] This is the value of parameter nblargnb in ⟨ecc_customize.vhd⟩. You can edit this value but do this only, according to the dedicated formula, if you really know what you're doing.

[2] For instance for the 7-series families of Xilinx FPGAs, the DSP blocks being of size 25 × 18, parameter ww must be set to 17 max to account for the fact that the most significant bit is a sign bit and hence it must always be set to 0. In the current release of IPECC code, ww is set to 16 to ease readabilty (especiallly of simulation) but you can set it to 17 if you feel that an extra bit is worth it, this has been thoroughly tested.

is the smallest power of 2 strictly greater than $4p$, requires 3 extra bits as compared to p[3]. It is also the largest positive number we need to be able to encode in IPECC.

(2) As curve points arithmetics involve field subtractions, a signed representation of large numbers is also required, implying an additional bit to encode their sign. We naturally use two's complement representation.

- Parameter `n` is defined as the power-of-2 which is either equal to or directly greater than `w`:

$$\mathtt{n} = 2^{\lceil \log_2(\mathtt{w}) \rceil}$$

Example values: $\begin{cases} \mathtt{nn}=256,\ \mathtt{ww}=17 \rightarrow & \mathtt{n}=16 \\ \mathtt{nn}=192,\ \mathtt{ww}=17 \rightarrow & \mathtt{n}=16 \\ \mathtt{nn}=320,\ \mathtt{ww}=16 \rightarrow & \mathtt{n}=32 \end{cases}$

⚠ Value of parameters `nn` and `ww` must be chosen so that `w` ≥ 2.

Note that this property is verified upon both simulation and synthesis and that code will refuse to compile if `w` $= 1$. If dynamic prime size feature is activated (`nn_dynamic` set to `TRUE` in ⟨`ecc_customize.vhd`⟩) the IP enforces the test `w` ≥ 2 for each new value of `nn` that is given by software at runtime and, if the test is inconclusive, the IP will refuse to perform any computation before a new value of `nn` is given that makes the property `w` ≥ 2 valid again.

D.2 Organization of memory

Memory `ecc_fp_dram` is organized as an array of 32 × `n` words (or limbs) of `ww` bit each, with large numbers being:

- stored as a sequence of `w` contiguous words in little-endian order (that is, the least significant `ww`-bit word of each large number is stored at the lowest address in memory)
- aligned on `n` words boundaries.

Thus first large number in the array spans the address range 0 to `w - 1`, second large number spans the address range `n` to `n+w-1`, third word spans the address range `2n` to `2n+w-1`, etc. This is illustrated on figure D.1.

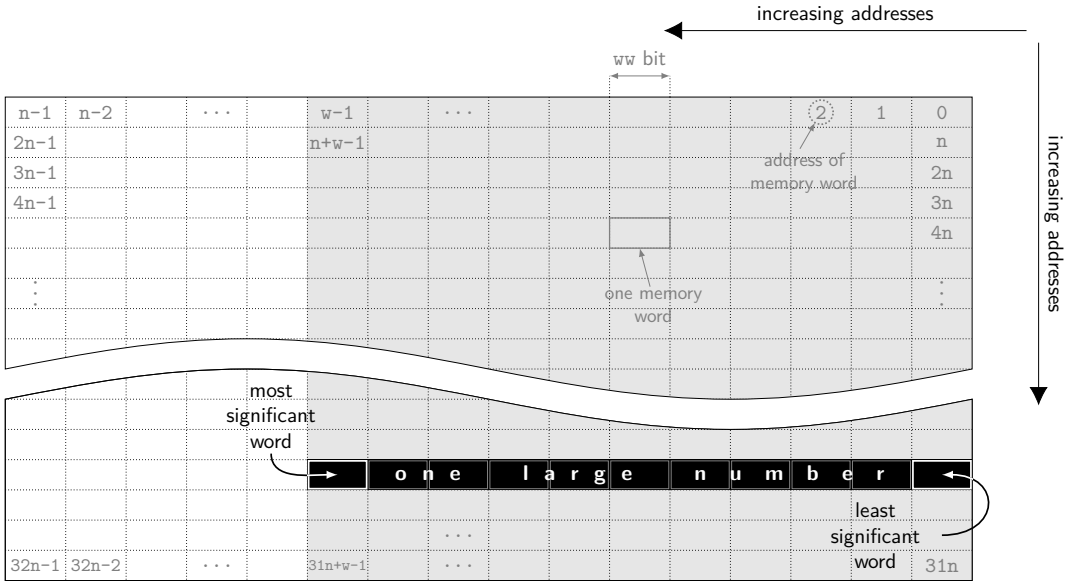
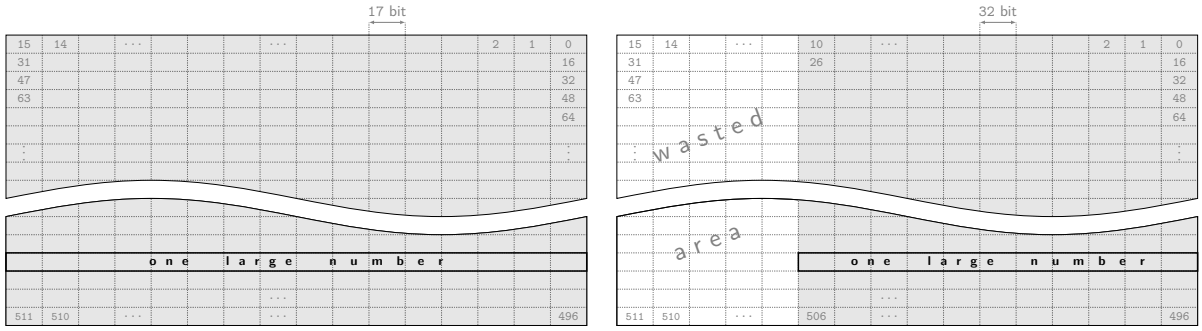


Figure D.1: Organization of IPECC internal memory of large numbers

Figure D.2 below shows the organization of `ecc_fp_dram` memory depending on whether `n` = `w` or `n` ≠ `w`. On figure D.2a, `n=w=16=`2^4. This is the case for instance with `nn=256` and `ww=17`. On figure D.2b, `n` ≠ `w`, `n=16,w=11`. This is the case for instance with `nn=320` and `ww=32`.

[3] p being of size `nn` bits, $4p$ is of size `nn`+2 bits, and R is of size `nn`+3 bits.



(a) n=w=16 (e.g nn=256, ww=17).

(b) n=16, w=11 (e.g nn=320, ww=32). White area is wasted.

Figure D.2: Organization of ecc_fp_dram depends on whether n = w or n ≠ w. Figures (a) and (b) illustrate this with two examples.

Figure D.2b clearly brings out that a large amount of memory could be wasted in ecc_fp_dram if parameter ww is not wisely chosen. This should be especially taken care of if you target an ASIC technology. In this case indeed, you should consider the trade-off between the size of the multipliers you want to get in your design (defined by value of multwidth) and the amount of memory you'll lose by choosing an inappropriate value of multwidth (that is, one that brings a value for w parameter that is not a power of 2).

> Example: assume you target an ASIC with nn=256. You have determined that the optimal width of your hardware multiplier is 16×16, because this is the best area/speed trade-off. If you set multwidth=16 in ⟨ecc_customize.vhd⟩, you'll get w=17, and hence n=32: this represents almost a 50% loss of space in the physical layout of ecc_fp_dram! Now you'd be well advised to set 17 for ww instead: a 17×17 multiplier is not much bigger nor slower than a 16×16 multiplier, but you'll get a 100% usage of the memory array inferred for ecc_fp_dram as w=n=16.

Appendix E

Programming pseudocodes

All the example pseudocodes below assume a 32-bit interface[1]. Codes are presented in bottom-up order: low-level actions are presented first, with last section §E.6 giving a complete sequence of operations including several point operations programmed on a curve, making use of all functions defined from §E.1 to §E.5.

The official IP driver (sources in folder `driver/` of the repository) generally follows the pseudocode guidelines presented herefater (with possibly a few minor differences from here to there). Obviously you should consider the official IP driver as the reference working code base.

The two small macros below are often used in the following pseudocode examples, that's why we define them here:

```
/* Macro to busy-wait before (or after) doing almost any action */
#define IPECC_BUSY_WAIT()  ( while (IPECC_READ_REG(R_STATUS, BUSY) != 0) {} )

/* Macro to compute the number of 32-bit words to be written to the IP
   (or to be read from it) to transfer a large number of size 'sz' (in bits)
   to it (or from it) */
#define NB_W32(sz) ( ((sz) % 32) ? ((sz) / 32) : (((sz) / 32) + 1) )
```

[1] Parameter `axi32or64 = 32` in ⟨`ecc_customize.vhd`⟩.

E.1 Example pseudocode to program a new prime size

```
int ipecc_program_new_prime_size(int nn) {

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Nothing to do if nn is value currently set */
  nn_cur = IPECC_READ_REG(R_PRIME_SIZE)
  if (nn == nn_cur) return success

  /* Read register R_CAPABILITIES & extract its bit-field CAP_NNMAX */
  nn_max = IPECC_READ_REG(R_CAPABILITIES, CAP_NNMAX)
  if (nn > nn_max)
    return error

  /* Read register R_CAPABILITIES & test its bit CAP_NNDYN */
  nn_dynamic = IPECC_READ_REG(R_CAPABILITIES, CAP_NNDYN)
  if (nn_dynamic) {

    /* IP was synthesized with dynamic prime size feature on.
       Write register W_PRIME_SIZE with value of nn */
    IPECC_WRITE_REG(W_PRIME_SIZE, nn)

    /* Poll register R_STATUS until its BUSY bit is deasserted */
    IPECC_BUSY_WAIT()

    /* Was there any error? */
    err = IPECC_READ_REG(R_STATUS, ALL_ERROR_FLAGS)
    if (err) {
      print("Programming new prime size raised errors: 0x%04x", err)
      /* Acknowledge errors */
      IPECC_WRITE_REG(W_ERR_ACK, err)
      return error
    }

  } else {

    /* IP was synthesized with a static nn value */
    if (nn == nn_max) { /* same info available in R_PRIME_SIZE for that matters */
      return success
    } else {
      return error
    }
  }
  return success
}
```

E.2 Example pseudocode to write one large number

```
int ipecc_write_large_number(int* x, int x_sz, int addr, bool scalar) {

  /* Sanity check */
  if (x == NULL) return error

  /* Is arg size compatible with the one currently programmed in the IP? */
  nn_cur = IPECC_READ_REG(R_PRIME_SIZE)
  if (x_sz != nn_cur)
    return error

  /* Is the target number the scalar?
     If so, we must wait until enough random is available
     (Remember address of scalar is overloaded as it's also address of XR0) */
  if (addr == ADDR_SCALAR) && (scalar == TRUE) {
    while (!IPECC_READ_REG(R_STATUS, ENOUGH_RND_WK)) {}
  }

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Select the target in write mode in register W_CTRL */
  wctrl = CTRL_WRITE_NB | ((addr & 0xFF) << CTRL_NBADDR_POS)
  if (addr == ADDR_SCALAR) && (scalar == TRUE) {
    wctrl |= CTRL_WRITE_K
  }
  IPECC_WRITE_REG(W_CTRL, wctrl)

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Was there any error? */
  err = IPECC_READ_REG(R_STATUS, ALL_ERROR_FLAGS)
  if (err) {
    print("Programming number for write raised errors: 0x%04x", err)
    /* Acknowledge errors */
    IPECC_WRITE_REG(W_ERR_ACK, err)
    return error
  }

  /* Loop to transfer number payload using register W_WRITE_DATA */
  for (i=0; i<NB_W32(x_sz), i++) {

    /* Transfer one 32-bit limb at a time */
    IPECC_WRITE_REG(W_WRITE_DATA, x[i])

    /* Wait for IP to be ready */
    IPECC_BUSY_WAIT()
  }

  /* Was there any error? */
  err = IPECC_READ_REG(R_STATUS, ALL_ERROR_FLAGS)
  if (err) {
    print("Transferring number payload raised errors: 0x%04x", err)
    /* Acknowledge errors */
    IPECC_WRITE_REG(W_ERR_ACK, err)
    return error
  }
  return success
}
```

E.3 Example pseudocode to read one large number

```
int ipecc_read_large_number(int* x, int x_sz, int addr, bool token) {

  /* Sanity check */
  if (x == NULL) return error

  /* Is arg size compatible with the one currently programmed in the IP? */
  nn_cur = IPECC_READ_REG(R_PRIME_SIZE)
  if (x_sz != nn_cur)
    return error

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Select the target in read mode in register W_CTRL */
  if (token/*== TRUE*/) {
    wctrl = CTRL_READ_NB | CTRL_RD_TOKEN
  } else {
    wctrl = CTRL_READ_NB | ((addr & 0xFF) << CTRL_NBADDR_POS)
  }
  IPECC_WRITE_REG(W_CTRL, wctrl)

  /* Was there any error? */
  err = IPECC_READ_REG(R_STATUS, ALL_ERROR_FLAGS)
  if (err) {
    print("Programming number for write raised errors: 0x%04x", err)
    /* Acknowledge errors */
    IPECC_WRITE_REG(W_ERR_ACK, err)
    return error
  }

  /* Loop to read number payload using register R_READ_DATA */
  for (i=0; i<NB_W32(x_sz), i++) {

    /* Transfer one 32-bit limb at a time */
    x[i] = IPECC_READ_REG(R_READ_DATA)

    /* Wait for IP to be ready */
    IPECC_BUSY_WAIT()
  }

  /* Was there any error? */
  err = IPECC_READ_REG(R_STATUS, ALL_ERROR_FLAGS)
  if (err) {
    print("Reading number payload raised errors: 0x%04x", err)
    /* Acknowledge errors */
    IPECC_WRITE_REG(W_ERR_ACK, err)
    return error
  }
  return success
}
```

E.4 Example pseudocode to program a new curve

```
int ipecc_program_new_curve(int* p, int* a, int* b, int* q, int p_sz, int q_sz) {

  /* Remember the order of the curve may be larger than the prime! */
  sz = max(p_sz, q_sz)
  if (ipecc_program_new_prime_size(sz) == error) {
    return error
  }

  /* Send prime p */
  if (ipecc_write_large_number(p, sz, ADDR_PRIME_P, FALSE) == error)
    goto err

  /* Send curve parameter a */
  if (ipecc_write_large_number(a, sz, ADDR_CURVE_A, FALSE) == error)
    goto err

  /* Send curve parameter b */
  if (ipecc_write_large_number(b, sz, ADDR_CURVE_B, FALSE) == error)
    goto err

  /* Send curve order q */
  if (ipecc_write_large_number(q, sz, ADDR_CURVE_Q, FALSE) == error)
    goto err

  return success
err:
  return error
}
```

E.5 Example pseudocode to program a $[k]\mathcal{P}$ computation

```
int ipecc_program_kp(int* xp, int* yp, bool p_is_null,     /* input point */
                     int* k,                               /* scalar */
                     int* tok,                             /* some room for token */
                     int nb_blind_bits,                    /* nb of blinding bits */
                     int* xkp, int* ykp, bool* kp_is_null, /* output point */
                     int sz)                               /* common size of all large people */
{

  /* Sanity check */
  if (ipecc_program_new_prime_size(sz) == error)
    return error

  if (p_is_null/*== TRUE*/) {

    /* Tell the IP the input point P is null */
    IPECC_BUSY_WAIT()
    IPECC_WRITE_REG(W_R1_NULL, 0x1 << R1_IS_NULL_POS)

  } else {

    /* Superfluous, for sake of readability */
    IPECC_WRITE_REG(W_R1_NULL, 0)

    /* Send base point coordinate X */
    if (ipecc_write_large_number(xp, sz, ADDR_CURVE_XR1, FALSE) == error)
      goto err

    /* Send base point coordinate Y */
    if (ipecc_write_large_number(yp, sz, ADDR_CURVE_YR1, FALSE) == error)
      goto err
  }

  /* Send scalar k */
  if (ipecc_write_large_number(k, sz, ADDR_SCALAR, TRUE) == error)
    goto err

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Is blinding required? */
  if (nb_blind_bits > 0) {
    wbld = BLD_EN | (nb_blind_bits << BLD_BITS_POS)
    IPECC_WRITE_REG(W_BLINDING, wbld)
  } else {
    IPECC_WRITE_REG(W_BLINDING, 0)
  }

  /* Busy wait, then probe and acknowledge errors */
  ...  (see previous examples)

  /* Get a one-shot random token from the IP */
  IPECC_BUSY_WAIT()
  IPECC_WRITE_REG(W_TOKEN, 0) /* Value doesn't matter here so why not 0 */

  /* Busy wait while the token is being generated */
  IPECC_BUSY_WAIT()

  /* Read the token (note the addr argument has no importance
     when calling ipecc_read_large_number() here */
```

```
  if (ipecc_read_large_number(&tok, sz, /*addr=*/0, /*token=*/TRUE) == error)
    goto err

  /* Run command through W_CTRL register */
  IPECC_BUSY_WAIT()
  IPECC_WRITE_REG(W_CTRL, CTRL_KP)

  /* Busy wait, then probe and acknowledge errors */
  ...  (see previous examples)

  /* Check that the output is not null (point at infinity) */

  if (IPECC_READ_REG(R_STATUS, STATUS_R1_IS_NULL) == 0) {

    /* position output flag */
    *kp_is_null = FALSE

    /* Read back X-coordinate of [k]P result */
    if (ipecc_read_large_number(&xkp, sz, ADDR_XR1, FALSE) == error)
      goto err

    /* Read back Y-coordinate of [k]P result */
    if (ipecc_read_large_number(&xkp, sz, ADDR_YR1, FALSE) == error)
      goto err

    /* Unmask token from coordinates of point [k]P */
    for (i=0; i<NB_W32(sz); i++) {
      xkp[i] = xkp[i] ^ tok[i]
      ykp[i] = ykp[i] ^ tok[i]
    }

  } else {
    /* position output flag */
    *kp_is_null = TRUE
  }

  return success
err:
  return error
}
```

E.6 Everything wrapped up

```
#define  NN  256

/* Large number buffers */
int p[8], a[8], b[8], q[8]; /* prime and curve parameters */
int xp[8], yp[8];           /* point coordinates */
int k[8];                   /* scalar */
int tok[8];                 /* room for token */
int xkp[8], ykp[8];         /* [k]P coordinates (IP to compute) */
bool kp_is_null;

int main() {

  /* Reset the IP */
  IPECC_WRITE_REG(W_SOFT_RESET, 0) /* Value doesn't matter here so why not 0 */

  /* Wait for IP to be ready */
  IPECC_BUSY_WAIT()

  /* Initialize all input large numbers */
  p[0] = ...,

  /* Program prime size */
  if (ipecc_program_new_prime_size(NN) == error) {
    printf("Programming a new prime size to the IP triggered an error")
    exit(FAILURE)
  }

  /* Program the curve */
  if (ipecc_program_new_curve(p, a, b, q, NN, NN) == error) {
    printf("Programming a new prime size to the IP triggered an error")
    exit(FAILURE)
  }

  /* Program [k]P */
  if (ipecc_program_kp(xp, yp, FALSE, k, tok, 96, xkp, ykp, &kp_is_null, NN) == error) {
    print("Programming [k]P computation on the IP triggered an error")
    exit(FAILURE)
  }

  if (kp_is_null) {

    print("[k]P = 0")

  } else {

    print("Coordinates of [k]P: ")
    ...
  }

  exit(SUCCESS)

}
```

Appendix F

A Tutorial: IPECC on Xilinx FPGAs

This chapter is a tutorial that will guide you through the process of using IPECC in an embedded system running Linux. We'll target the **Arty Z7** board from Digilent (fig-F.1) which is among the cheapest Zynq SoC-FPGA boards on the market, available at approx. $199. Official vendor page is: `https://digilent.com/reference/programmable-logic/arty-z7/start`. This board is populated with a Xilinx `xc7z010` SoC-FPGA device in speed grade 1 (an `xc7z020` version exists which is a little more expensive at $399). This device is built on a dual core ARM Cortex-A9 processor running at 667 MHz interfaced with an FPGA matrix of approximately 13K slices, equivalent in its architecture to an Artix-7 logic matrix. **This device does not require the licenced version of the Xilinx CAD tools,** therefore everything hereunder should be achievable without having to pay any extra money other than the price of the board itself.

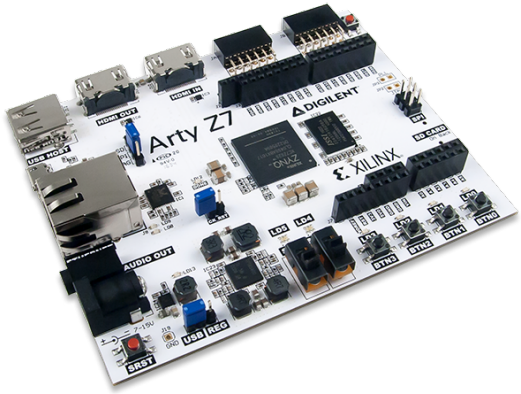


Figure F.1: The Arty Z7 board from Digilent comes in two versions: 7z10 and 7z20.

☞ Everything described in this tutorial should be easily replicable on any other Xilinx Zynq board (the Arty board was only chosen for demonstration purposes).

First you'll instanciate and configure the IP inside the programmable logic part (PL) of the device, and simulate and synthesize the hardware using the Vivado FPGA tool chain. In a second part you'll compile the Linux sources with the kernel configured to match the specificities of the hardware. We'll use PetaLinux, the Yocto-based distribution of embedded Linux provided by Xilinx for their SoC-FPGA devices. Finally, you'll use the **libecc** software library, also available from ANSSI's Github (`https://github.com/ANSSI-FR/libecc.git`) and patch it specifically so that low level calls to scalar multiplication routine are re-routed to IPECC (thus used as a hardware accelerator for elliptic curve cryptography) instead of being computed on the local CPU. You will download both FPGA bitstream and Linux kernel and filesystem to the microSD card, boot the complete system on the board and watch the software run validation tests and display performance statistics over a remote console running on your host PC.

☞ Due to the strong adherence of Tcl project stripting to the version of Vivado, as well as the diversity of bugs and anomalies one can encounter while using FPGA tools depending on their OS and distribution (even on so-called supported Linux distros) we do not provide project configuration script for this tutorial.

Instead we start from a *known free version of Linux* (Ubuntu LTS 22.04.2) running inside a virtual machine, and a *known free version of Vivado* (2022.1 ML standard) running above it, to ensure you get a working thing. From this point you may then iterate to meet your own specific hardware requirements and/or software configuration.

It is important when you enter the FPGA ecosystem to be advised that tools, whatever their vendor, show an impressive diversity of irregularities and disfunctions that often require nasty kludges, workarounds and sometimes voodoo incantations so they can work normally for more than 5 minutes, and the present example is no exception (see below *kludge #1* and *kludge #2*).

Understand this tutorial as a means to provide you quickly and efficiently with a guaranteed real hardware working case of the IP while also offering a short guided tour of the IP and of some of its features.

☞ Everything (host, guest, tools) are assumed to be 64-bit versions.

There are two versions of the tutorial: a short version, probably the one you want to read, where actions to take are given as concisely as we could, and a detailed version, with a lot of technical infos and also troubleshooting sections (all of that you may find a little verbose). This version is more intended for students and/or beginners. The short version of the tutorial is in §F.1, the long version is in §F.2. Both versions use the same numbering so you can directly switch from the short version to the verbose one whenever something's not clear or you wish to go further into details. Figures are common to both versions. They've been moved altogether in §F.3 so that you don't necessarily have to print them, in case you *would prefer not to*.

F.1 Short version of the tutorial

Legend	
CLI	Commands to run on the guest machine (whatever their privileges)
	Actions inside Vivado
	Actions inside Vitis
	File edition
	Actions inside SageMath
	Actions in Linux menuconfig
	Actions inside Internet browser
arty	Commands to run in the board console
	Actions to perform on the board (jumpers, cables, etc)
	Actions inside hypervisor (but out of the guest), here VirtualBox

Ubuntu installation

1. Create a virtual machine with **Ubuntu Desktop 22.04.2**:
   - ⟨ubuntu-22.04.2-desktop-amd64.iso⟩
   - sha256: b98dac940a82b110e6265ca78d1320f1f7103861e922aa1a54e4202686e9bbd3
   - *Normal installation* rather than *Minimal installation*

   The **VM configuration** we used (try at least as powerful):
   - 16 GB RAM, 4 processors, HDD **250 GB** (for install of the Vivado golem + PetaLinux)
   - All VT-x/AMD-V acceleration features enabled, PAE/NX, IO-APIC, nested pagination

   We used **VirtualBox 6.1** (try at least that recent).

   Install the English (US) version of Ubuntu so as to get the proper `en_US.UTF-8` locale.
2. With the guest up and running:

```
CLI $ sudo apt-get update
CLI $ sudo apt-get install gcc dkms build-essential perl vim
```

3. *Devices* ▶ *Insert Guest Additions CD Image* (requires the packages installed in step 2).

IPECC GitHub repo fetch & install

4.
```
CLI $ sudo apt-get install git
CLI $ cd ~/
CLI $ git clone https://github.com/ANSSI-FR/IPECC.git
    Cloning into 'IPECC'...
CLI $ cd IPECC
CLI $ ls
    doc  driver  hdl  LICENSE  README.md  sage  sim  syn
```

☞ From now on we'll refer to the local copy of the IPECC repo. using the `${IPECC}` env. variable

5. Build the **microcode** of the IP (+ a few other automatically generated files):

```
CLI $ cd ${IPECC}/hdl/common/ecc_curve_iram
CLI $ make
    -> Creating ASM main program ecc_curve_iram.s
    -> Parsing ../ecc_pkg.vhd, ../ecc_customize.vhd and ...
    [+] Parsing of VHDL files done, everything OK ...
    ...
```

Ignore the possible yellow warnings.
Check that the six files below (marked with a *) have actually been generated:

```
CLI $ ls -1
    asm_src
    ecc_addr.h*
    ecc_addr.vhd*
    ecc_curve_iram.s
    ecc_curve_iram.vhd*
    ecc_states.h*
    ecc_vars.h*
    ecc_vars.vhd*
    ipecc_assembler.py
    latex
    Makefile
```

The three `.vhd` files are VHDL files required for both simulation and synthesis of the IP.
The three `.h` files are C header files allowing driver compilation to be kept consistant with the hardware API.

6. Optionnally:

```
CLI $ sudo apt-get install texlive-full                    # Takes some time
```

CLI `$ make latex` # Still in folder ecc_curve_iram/
CLI `$ evince latex/ecc_curve_iram.pdf &`

This file ⟨`ecc_curve_iram.pdf`⟩ will give you a neat and colorized pdf listing of the complete assembled microcode of the IP.

7 Folder organization:

```
CLI $ mkdir -p ~/ipecc/hw/ip              # IP sources that we're going to customize (HW)
CLI $ mkdir    ~/ipecc/hw/ip/packaged     # IP sources wrapped in an exportable bundle (HW)
CLI $ mkdir    ~/ipecc/hw/soc             # Sources for complete SoC (HW)
CLI $ mkdir -p ~/ipecc/sw/sage            # Sage scripting (SW)
CLI $ mkdir    ~/ipecc/sw/linux           # Linux install & build (SW)
```

Vivado installation

8 `xilinx.com`, find the Vivado ML download page (actually you might as well g**gle *«Vivado ML»* in).
Select **Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer** (fig-F.4).

- You'll have to create an account (for free)
- Installer MD5: `db5056feaaf271fe90ba54bae4768ed2`

```
CLI $ cd ~/Downloads
CLI $ md5sum Xilinx_Unified_2022.1_0420_0327_Lin64.bin
db5056feaaf271fe90ba54bae4768ed2      Xilinx_Unified_2022.1_0420_0327_Lin64.bin
CLI $ chmod +x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
CLI $ ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

9 Ignore the *Get Latest* proposition (fig-F.5). Also ignore red warning threatening you that you are on an unsupported OS (fig-F.6). In the install tool select **Vitis** (includes **Vivado**).

10 When selecting the devices to install (fig-F.7) select only the «Zynq-7000» box (keep all other boxes unchecked). The component we'll target on the Arty board is part number `xc7z010` which belongs to the Zynq-7000 family.
Keep box *«Install devices for Alveo and Xilinx edge accel. platforms»* checked (it installs required libs).

- Required disk space should amount to a little more than 165 GB (`:-.`)
- Install process may be quite long depending on your bandwidth (1 h or so) so be patient.

Running Vivado

11 Using the Xilinx tools requires that you first source a specific script to set some paths. This script is named ⟨`settings64.sh`⟩ for Bash and is located in the Vitis/Vivado install tree. It's not a good idea to have this script automatically sourced by your shell run-control script (`~/.bashrc[+]` for Bash) because the Xilinx script sets the `PATH` and the `LD_LIBRARY_PATH` environment variables in such a way that it might interfere with other programs. So each time you need to launch Vitis ou Vivado first do this:
CLI `$ source ~/xilinx/Vitis/2022.1/settings64.sh`
That last command assumes that you previously gave `~/xilinx/` as the root path to the Vivado installation in steps 8-10 (obviously adapt this line to your own install dir).

12 A Vivado prerequisite that Xilinx forgot to tell you about is the `libtinfo.so.5` shared lib:
CLI `$ sudo apt-get install libtinfo5` # Vivado won't start without it
CLI `$ vivado &`
The GUI opens and displays the welcome layout (fig F.8).

Vivado project creation

13 [Welcome layout] *Create Project* using these settings:

- Section **Project Name:** Project name: `ecc`
  Project location: `ipecc/hw/ip`

Uncheck box *Create project subdirectory*

- Section **Project Type:** RTL Project

  Check box *Do not specify sources at this time*
- Section **Default Part:** Use Search bar to select `xc7z010clg400-1`

Flow Navigator ▶ *Project Manager* ▶ *Add sources* ▶ *Add or create design sources* ▶ *Add Files*

⚠ Keep the *Copy sources into project* box checked (so as not to interfere with the git repo). The box is kept unwritable until you start specyfing files to add.

Files will be copied in `ipecc/hw/ip/ecc.srcs/sources_1/imports/hdl/`.

Below is a flatten list of the source files you should select (see also fig-F.9). You'll have to proceed in 4 steps (denoted 1/4 to 4/4 below), each time clicking on button *Add Files* to add the files of one of the four directories below, and in the end click *Finish*.

1/4. These are the files you must add from folder `${IPECC}/hdl/common`:

- `ecc.vhd`
- `ecc_axi.vhd`
- `ecc_curve.vhd`
- `ecc_customize.vhd`
- `ecc_fp.vhd`
- `ecc_fp_dram.vhd`
- `ecc_fp_dram_sh_fishy.vhd`
- `ecc_fp_dram_sh_fishy_nb.vhd`
- `ecc_fp_dram_sh_linear.vhd`
- `ecc_log.vhd`
- `ecc_pkg.vhd`
- `ecc_scalar.vhd`
- `ecc_shuffle_pkg.vhd`
- `ecc_software.vhd`
- `ecc_utils.vhd`
- `ecc_vars.vhd`
- `fifo.vhd`
- `mm_ndsp.vhd`
- `mm_ndsp_pkg.vhd`
- (let aside file `pseudo_trng.vhd`)
- `sync2ram_sdp.vhd`
- `syncram_sdp.vhd`
- `virt_to_phys_ram.vhd`
- `virt_to_phys_ram_async.vhd`

2/4. These are the files you must add from folder `${IPECC}/hdl/common/ecc_trng`:

- `ecc_trng.vhd`
- `ecc_trng_pkg.vhd`
- `ecc_trng_pp.vhd`
- `ecc_trng_srv.vhd`
- `es_trng.vhd`
- `es_trng_aggreg.vhd`
- `es_trng_bitctrl.vhd`
- `es_trng_sim.vhd`

3/4. These are the files you must add from folder `${IPECC}/hdl/common/ecc_curve_iram`:

- `ecc_addr.vhd`
- `ecc_curve_iram.vhd`
- `ecc_vars.vhd`

4/4. These are the files you must add from folder `${IPECC}/hdl/techno-specific/xilinxa/series7`:

- `es_trng_bit_series7.vhd`
- `large_shr_series7.vhd`
- `macc_series7.vhd`
- `maccx_series7.vhd`

The last batch of files is taken from folder `xilinxa/series7/` because we're targeting a Zynq device. Select `xilinxa/ultrascale/` instead if you're targeting an UltraScale device (this'll mainly impact the structural HDL description of the TRNG and the instanciation of the DSP block).

Give Vivado a minute after clicking *Finish* to let it parse the files and create the design hierarchy (Updating).

14 ⚠ *Flow Navigator* ▶ *Project Manager* ▶ *Settings* ▶ *Tool Settings* ▶ *Text Editor* ▶ *Syntax Checking*: choose *Vivado* instead of *Sigasi* (fig-F.13) – the Sigasi third party tool can hang Vivado sometimes.

While you're at it:

Settings ▶ *Project Settings* ▶ *Simulation* ▶ *Simulation* tab: Click on parameter `xsim.simulate.runtime` default value (probably 1000 `ns`) then click on the gray cross in the right side of the box to cancel the option ▶ OK.

Close the project (*File* ▶ *Close Project*) and open it again (*File* ▶ *Project* ▶ *Open Recent*) to have the new settings taking effect.

15 Edit ⟨`ecc_customize.vhd`⟩ from folder ~/ipecc/hw/ip/ecc.srcs/sources_1/imports/hdl/`common`. Parameters you must edit are shown on fig. F.18 p. 162, apply these carefully.

☞ Refer to step **15** of the verbose version of the tutorial for more information on these parameters.

⚠ Note that parameter `notrng` must only be set to `TRUE` when running simulation. We'll later turn it back to `FALSE` when running synthesis (see step **26**).

Simulation of the IP

Simulating the IP requires to first generate two input files, ⟨/tmp/`random.txt`⟩ and ⟨/tmp/`ecc_vec_in.txt`⟩.

16 **Generation of random file** ⟨/tmp/`random.txt`⟩:

CLI
```
$ od -t u1 -w1 -v /dev/urandom | awk '{print $2}' | head -$((16*1024*1024)) > /tmp/random.txt
```

☞ You may take a glimpse at file ⟨${IPECC}/sim/`HOWTO-random.txt`⟩ for more info.

17 **Generation of the input test-vectors file**

CLI
```
$ sudo apt-get install sagemath                                    # Takes half an hour...
```

CLI
```
$ cp -Rf  ${IPECC}/sage/*  ~/ipecc/sw/sage/.
```

CLI
```
$ cd ~/ipecc/sw/sage
```

Edit ⟨`generate-tests.sage`⟩ from folder ~/ipecc/sw/`sage`, search for the following parameters and set each one to the value that is given alongside:

```
nn_constant = 21      # meaning all generated curves will be of size nn = 21 bits
NBCURV = 1            # meaning only one curve will be generated
NBKP = 1              # meaning only one scalar-multiplication test generated per curve
NBADD = 1             # meaning only one point-addition test generated per curve
NBDBL = 1             # meaning only one point-doubling test generated per curve
NBNEG = 1             # meaning only one point-negate (−P) test generated per curve
NBCHK = 1             # meaning only one boolean "is P on curve?" test generated per curve
NBEQU = 1             # meaning only one boolean "are points equal?" test per curve
NBOPP = 1             # meaning only one boolean "are points opposite?" test per curve
NO_EXCEPTION = True   # meaning no exception tests generated
```

Run the script as an input to *Sage*:

CLI
```
$ sage generate-tests.sage >/tmp/ecc_vec_in.txt
Generating curves for nn = 21
```

☞ Alternatively if you wish to observe strictly the same simulation results as in the tutorial you can use the same test-vector file as we do, which you'll find in ${IPECC}/`sim`. Simply copy it in `/tmp/` or have parameter `simvecfile` in ⟨`ecc_customize.vhd`⟩ to point to ${IPECC}/sim/`ecc_vec_in.txt`.

☞ Also note that folder ${IPECC}/`sim` contains a file named ⟨`std-curves-test-vectors.txt`⟩ that you can also use as test-vector input file with curves and points of cryptographic sizes, but the simulation will be considerably much longer.

18 Edit the input test vector file you've elected to use and get a glimpse at the format (fig-F.19 on p. 162).

☞ You may find more information on this topic in step **18** of the verbose version of the tutorial, along with detailed explanations on the different steps of the simulation (the $[k]\mathcal{P}$ computation in particular).

19 **Adding simulation sources**

Project Manager ▶ *Add Sources* ▶ *Add or create simulation sources* (this is the **third line**) (fig-F.15) ▶ *Next* ▶ *Add Files* ▶ Browse to ${IPECC}/`sim` folder and select the four files hereafter:

- `ecc_tb.vhd`
- `ecc_tb.wcfg`
- `ecc_tb_pkg.vhd`
- `ecc_tb_vec.vhd`

Be sure to check box *Copy sources into project* box. Also keep default check of option *Include all design sources for simulation* (fig-F.16). Validate with *Finish*.

Vivado will copy these 4 files into ~/ipecc/hw/ip/`ecc.srcs/sim_1/imports/sim/`.

20 Running simulation

Flow Navigator ▶ *Simulation* ▶ *Run Simulation* ▶ *Run Behavorial Simulation*

This launches compilation of all the source files and elaboration (linking). A waveform viewer is displayed according to the configuration set in the waveform file ⟨`sim/ecc_tb.wcfg`⟩.

TCL prompt (`Type a Tcl command here`) (bottom of Vivado): enter command `run 800 us`.

Alternatively you can also enter the value 800 µs using the simulation time bar in the menu bar of Vivado (`800 us`) and click on the button. Also note that depending on whether or not the $[k]\mathcal{P}$ test generated in your own version of the input test-vector file is set with blinding (this is randomly decided) you might have to run the simulation longer that 800 µs (because the blinding much increases the computation time of the scalar multiplication). If you want to be sure to observe the complete computations run the simulation for 1600 µs.

The simulation takes one or two minutes. The waveform is continuously updated as simulation moves forward (fig-F.20) and informations are logged onto the Vivado simulation console (fig-F.21). The numbers displayed in violet on the two figures F.20 and F.21 correspond with each other (meaning they match the same steps of the simulation).

☞ If you're interested in the description of the different stages of the $[k]\mathcal{P}$ computation and the events during the simulation, please refer to same step **20** of the verbose tutorial.

21 Getting more details on arithmetical operations

Open file ⟨`/tmp/ecc.log`⟩. This trace file provides fine-grain informations on IP execution. In particular all the software routines executed by `ecc_curve` and the result of all arithmetical instructions processed by components `ecc_fp` and `mm_ndsp` are logged into it.

☞ Refer to the same step **21** of the verbose tutorial for more detail. At least do grep the regular expression `"[XY]R[01] ="` and also character string `"[VHD-CMP-SAGE]"` in this file to observe the log of intermediate cryptographic values. This is useful to compare the intermediate results obtained in simulation with the ones obtained by *Sage* sofware-proof execution. We'll look into that later on (c.f steps **23**-**24**).

22 Deep inspection and validation of intermediate arithmetical terms

CLI `$ cd ~/ipecc/sw/sage`

CLI `$ cp kp.py kp21.py`

Edit ⟨`kp21.py`⟩ and fill in the informations according to the values set in the test-vector file (fig-F.22 p. 165). Set `ww=16` (this is the bit width of the limbs forming large numbers inside the IP memory). To ease the extraction of random data from file `/tmp/ecc.log`, grep regular expression `"\.random_.*L"`:

CLI `$ grep -A 3 "\.random_.*L" /tmp/ecc.log`

```
.random_alphaL [0x031]
[0x031]    NNRND 0xbbd431af  (12 <-  random  )  [96705000 ps]
[0x032]    NNADD 0x001ce256  (08 <- 03  +  31)  [96875000 ps]
[0x033]    NNADD 0x00000000  (09 <- 31  +  31)  [97045000 ps]
--
.random_muL [0x041]
[0x041]    NNRND 0x7dd0807a  (26 <-  random  )  [117645000 ps]
[0x042]    TESTPAR           (26 is   even   )  [117755000 ps]
[0x043]    NNRND 0x1b570add  (27 <-  random  )  [117885000 ps]
--
.random_phiL [0x047]
[0x047]    NNRND 0x7f4e8d2f  (10 <-  random  )  [118585000 ps]
[0x048]    NNRND 0xc75870d7  (11 <-  random  )  [118715000 ps]
[0x049]    TESTPAR           (04 is    odd   )  [118825000 ps]
--
.random_lambdaL [0x071]
[0x071]    NNRND 0x00110c4a  (21 <-  random  )  [141345000 ps]
[0x072]    NNSUB 0xfff426ff  (22 <- 21  -  00)  [141515000 ps]
[0x073]    NNADD 0x00110c4a  (21 <- 22  +  00)  [141685000 ps]
...
(ignore remaining matches of .random_lambdaL)
```

You can see on the above excerpt that the correspondence with the parameters that need to be set in *Sage* is quite clear (fig-F.23 p. 165).

☞ Again you're encouraged to refer to step **22** of the verbose tutorial for the meaning of these parameters.

23 CLI `$ cd ~/ipecc/sw/sage`

CLI `$ sage --preparse kp.sage`

CLI `$ mv kp.sage.py kpsage.py`

CLI `$ python3 kp21.py | grep VHD-CMP-SAGE >/tmp/sage21.log`

CLI `$ grep VHD-CMP-SAGE /tmp/ecc.log >/tmp/simu21.log`

24 Observe the difference between the two files ⟨`/tmp/simu21.log`⟩ and ⟨`/tmp/sage21.log`⟩:

CLI `$ vimdiff /tmp/simu21.log /tmp/sage21.log`

What you should see (approximately of course, since the random values you have won't be the same) can be seen on fig-F.24 (p.166).

- Notice how the four coordinates `[XY]R[01]` of points $\mathcal{R}_0$ and $\mathcal{R}_1$ match in both log files from start of files until computation reaches the scalar bit of weight 6. Then the effect of the so-called «Z-remask» countermeasure takes effect (remember we set `zremask=4` in ⟨`ecc_customize.vhd`⟩ in step 15). The IP starts parsing the scalar from its third least significant bit (viz. the bit of weight 2)[1] so the first time where the coordinates of $\mathcal{R}_0$ and $\mathcal{R}_1$ are re-randomized happens to be the bit of weight 6.
- Also notice that past the first step of computation (first six lines of each file) the addresses of the four coordinates `[XY]R[01]` always differ from one file to the other. On the right (*Sage* emulation) they keep their initial static values (`@XR0 = 4`, `@YR0 = 5`, `@XR1 = 6` and `@YR1 = 7`) while on the left (hardware simulation) they are continuously randomized. This is the effect of the so-called «XY-shuffling» countermeasure.
- Now the final results naturally match in both files (fig-F.25, p. 166).

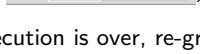
25 We're now going to disable the «XY-shuffling» and «Z-remask» countermeasures, one after the other, so as to observe complete match between the hardware on one side, and the *Sage* script (which should be considered as its software-proof emulation) on the other.

For the «Z-remask» countermeasure, we could simply edit file ⟨`ecc_customize.vhd`⟩ and set `zremask=0` in it.

For the «XY-shuffling» however it's not that simple because as already mentioned before (see step 22 above) the countermeasure can only be disengaged if the IP has been instancied in debug (unsecure) mode. That's why we'll start by switching the IP into debug mode.

Edit ⟨`ecc_customize.vhd`⟩ [2] and set `debug` to `TRUE` (fig-F.26 p. 167)

Edit ⟨`ecc_tb.vhd`⟩ [3] and search for character string `"End of IP initialization & config"` then add a call to procedure `configure_zremasking()` before that comment (fig-F.27 p. 167).

Relaunch the simulation: shortcut `Alt-R-U` recompiles the design, and, once this is done, `Shift-F2` re-runs the simulation from time 0, for the amount of time that you can set in the simu time bar (e.g 

).

Once execution is over, re-grep the hardware simu log:

CLI `$ grep VHD-CMP-SAGE /tmp/ecc.log >| /tmp/simu21.log`

Now observe the difference between the two log files (fig-F.29-F.30 p. 168):

CLI `$ vimdiff /tmp/simu21.log /tmp/sage21.log`

The effect of «Z-remask» has disappeared as coordinates match all along the computation in both log files. Go ahead and disable «XY-shuffling»:

Edit again ⟨`ecc_tb.vhd`⟩ and add a call to procedure `debug_disable_xyshuf()` (fig-F.28 p. 167)

The two functions we've added a call to in ⟨`ecc_tb.vhd`⟩ are defined in the VHDL simu package ⟨`ecc_tb_pkg.vhd`⟩. If you open this file and skim through its content you will see that it contains many helper functions that you can call when simulating the IP. Each of these functions performs an action on the IP through the AXI interface, thus emulating what a software driver would do to perform the same action when running from a CPU.

`Alt-R-U` (wait for compilation to complete) then `Shift-F2`.

When simulation has been run again:

CLI `$ grep VHD-CMP-SAGE /tmp/ecc.log >| /tmp/simu21.log`

[1] The reason why the IP starts parsing the scalar only at its bit of weight 2 when computing $[k]\mathcal{P}$ is explained in §4.1, see in particular algorithm 12 and the discussion pertaining to it in p.27.

[2] Remember this file is in `ipecc/hw/ip/ecc.srcs/sources_1/imports/hdl/common`.

[3] Remember this file is in `ipecc/hw/ip/ecc.srcs/sim_1/imports/sim/`. This is the source file of the RTL testbench.

Now the two log files should be strictly identical, as attested by the `cmp` command (fig-F.31-F.32 p. F.31):

```
CLI $ cmp /tmp/simu21.log /tmp/sage21.log # cmp stays mute when input files are identical
CLI $ echo $?                             # confirmed by successful 0 return code
0
```

☞ As a conclusion to this part of the tutorial, you can validate the behaviour of the VHDL model of the IP at every main step of $[k]\mathcal{P}$ computation, as long as the two countermeasures «XY-shuffling» and «Z-remask» are disabled.

☞ The support for these two countermeasures may be added to the *Sage* emulation script in a future release.

File ▶ *Close Simulation*.

Synthesis of the IP

To synthesize the IP we must instanciate it inside a system-on-chip (CPU, AXI interconnect, DDR controller etc) so that driving the IP through a simple piece of software will be a straightforward operation. Here the CPU is an ARM Cortex-A9 dual core that the Xilinx ecosystem calls the «PS» (the processing system) as opposed to the «PL» (the programmable logic, i.e the logic fabric).

26 We'll **package the IP** in folder `ipecc/hw/ip/packaged` and later create and edit the SoC design instanciating it in folder `ipecc/hw/soc`.

Edit ⟨`ecc_customize.vhd`⟩ to set `notrng` to `FALSE` (fig-F.33). We now want the real physical TRNG to be part of the design, not the fake simu one that was just reading «random» values from a file (and for that very reason obviously isn't synthesizable).

☞ **Keep the IP in debug mode** (`debug` = `TRUE`). This way you may later explore the different features of the IP when you have it run on a real device, without being constantly rejected by the API of hardware because it thinks that you're attempting illegitimate actions. In particular *only the debug mode makes it possible to read back from software the random bits generated by ES–TRNG and hence assess and validate the quality of its randomness.* Once the placement and routing of the TRNG provides the quality we expect from it, we can «hardware-lock» the design to ensure it will behave identically in subsequent releases of the design.

Tools ▶ *Create and Package New IP...* ▶ *Next* ▶ *Package your current project...* ▶ Set *IP location* to `ipecc/hw/ip/packaged` ▶ *Finish*.

Vivado spawns a new window containing a temporary project named ~/`tmp_edit_project`. Review (simply out of curiosity) the different steps in the column *Packaging Steps* inside tab *Package IP–ecc*. These steps are listed with a «✓» symbol on their side.

Remove file ⟨`es_trng_sim.vhd`⟩ from the sources:

✓*File Groups* ▶ *Standard* ▶ *Synthesis* ▶ *Right-click* ⟨`src/es_trng_sim.vhd`⟩ ▶ *Remove File*.

Review and Package ▶ *Package IP* (ignore possible blue warning saying that IP has been modified).

Back to main window:

File ▶ *Close Project*.

27 **Integrating the IP in the SoC**

Follow same steps as in **13** (p. 102) to create new project through the wizard, using these settings:

- Section **Project Name:** Project name: `az7-ecc-axi`
  Project location: `ipecc/hw/soc`
  Keep box *Create project subdirectory* unchecked
- Section **Project Type:** RTL Project
  Keep box *Do not specify sources at this time* checked
- Section **Default Part:** select **Boards** (instead of Parts) ▶ bottom left *Refresh* button (you'll need an Internet connexion here for Vivado to refresh its catalog, fig-F.36) ▶ *Vendor:* `digilentinc.com` ▶ *Name:* `ArtyZ7-10` ▶ ⬇ button (installs/updates the board) ▶ Select the board by clicking on the ArtyZ7-10 line (this line must be highlighted in blue, fig-F.37) ▶ *Next* ▶ Review the choices you made (check that part `xc7z010clg400-1` is actually set!) ▶ *Finish*.

- *Flow Navigator* ▶ *IP Integrator* ▶ *Create Block Design* ▶ OK (default settings)
- *Diagram* tab ▶ Button [+] (center of the plain white zone) ▶ *ZYNQ7 Processing System*.
- *Run Block Automation* ▶ OK (default settings).
- Right-click ZYNQ block ▶ Customize Block...
  - ▶ Interrupts ▶ Fabric Interrupts (check box + unroll) ▶ PL-PS Interrupt Ports ▶ Check box `IRQ_F2P[15:0]` (fig-F.39)
  - ▶ Clock Configuration ▶ PL Fabric Clocks ▶ Check box `FCLK_CLK1` ▶ Set Requested Frequency(MHz) to `200` (fig-F.40) ▶ OK (ignore possible warning about negative DDR DQS delay, fig-F.41).

  The Zynq processor is upgraded (fig-F.42) with an AXI initiator port (named `M_AXI_GP0`), an interrupt signal (named `IRQ_F2P`) and two clock outputs (named `FCLK_CLK0` and `FCLK_CLK1`).

☞ Refer to step **27** of the verbose tutorial for more info.

28 Now **let's bring IPECC into the system.**

- *Flow Navigator* ▶ *Project Manager* ▶ *Settings* ▶ IP (unroll) ▶ Repository ▶ + ▶ Browse to `ipecc/hw/ip/packaged` ▶ Select (fig-F.43) ▶ Apply ▶ OK.
- Block Design diagram ▶ Toolbar button [+] ▶ Search bar: `ecc` ▶ `ecc_v1_0`.
- *Run Connection Automation* ▶ OK (default settings).

The Block Design is enhanced with two new blocks implementing the AXI interconnect plus clocks and reset as well as signals and buses interconnecting the different parts together.

- Right-click anywhere in empty white zone of Block Design ▶ *Regenerate Layout* (for sake of visibility, fig-F.45).

You can see that the `FCLK_CLK0` clock output of the Zynq PS block is now connected to the `s_axi_aclk` input clock of the IP.

- Hover mouse over `FCLK_CLK1` output of Zynq PS block ▶ [Pencil pointer] Click and hold the mouse button down to pull a signal line from here to clock input `clkmm` of block `ecc_0` ▶ release button.
- Same operation from `IRQ_F2P[0:0]` input of Zynq to output `irq` of `ecc_0`.
- Right-click `busy` port of `ecc_0` ▶ *Make External* (an output port is created and given name `busy_0`, we'll later assign a package pin to it to drive an on-board LED and visually monitor the activity of the IP).

Keep remaining output ports (`dbgtrigger`, `dbghalted` and `dbgptrdy`) of the IP unconnected. These are outputs[4] so it doesn't matter if they stay unconnected (synthesizers will trim them out).

The Block Design should now look like fig-F.46. Does that look all right to you? Give it a try:

- Right-click anywhere in empty white zone of Block Design ▶ *Validate Design*.

☞ Vivado isn't satisfied (fig-F.47) because there are two remaining input ports of the IP that need to be connected, the 8-bit data bus `dbgptdata` and its strobe signal `dbgptvalid`. As opposed to output ports, it's a sound requirement that input ports be rigorously and unambiguously connected! Even if we're not going to use pseudo-TRNG feature here, better connect these signals to the ground:

- Block Design diagram ▶ Toolbar button [+] ▶ Search bar: `constant` ▶ `Constant` ▶ Double-click new block `xlconstant_0` ▶ Set `Const Width` = 1, `Const Val` = 0.
- Same operation to add a second `Constant` block ▶ Double-click new block `xlconstant_1` ▶ Set `Const Width` = 8, `Const Val` = 0x00.
- Connect output `dout[0:0]` of `xlconstant_0` to input `dbgptvalid` of IPECC.
- Connect output `dout[7:0]` of `xlconstant_1` to input bus `dbgptdata` of IPECC.
- Repeat *Regenerate Layout*, fig-F.48.
- Retry *Validate Design* (shortcut `F6`).

Vivado is now happy.

- `Ctrl+S` (or button 💾 in main GUI toolbar) to save the Block Design.

[4] Vivado by default place output ports on the right side of the block while input ports are placed on the left side.

29 **Synthesis and place-and-route**

Block Design ▶ *Sources* tab ▶ select *Hierarchy* view (at bottom) ▶ Unroll *Design Sources* ▶ Right-click `design_1 (design_1.bd)` ▶ *Create HDL Wrapper* ▶ OK (default settings).

Wait a few seconds while Vivado is Updating its files structure. Then a VHDL wrapper is created for the complete PS + PL Block Design which embeds everything that was edited graphically, along with the IP sources. This wrapper is elected as the top-level of hierarchy (as indicated by its name being displayed in bold (fig-F.49).

Sources tab ▶ Unroll `design_1_wrapper` ▶ Right-click `design_1_i:design_1 (design_1.bd)` ▶ *Generate Output Products* ▶ *Generate* (default settings).

The operation takes 1 to 2 minutes and then you're being informed that *«Out-of-context modules were launched for generating output products»* (whatever that means).

Flow Navigator ▶ *RTL ANALYSIS* (main GUI left panel) ▶ *Open Elaborated Design* ▶ OK.

Be patient as the operation takes ∼10', then the elaborated design is opened, which turns to be an abstraction level very close to a synthesized netlist.

Right-top corner of Vivado: select *I/O planning* layout (fig-F.50) ▶ *I/O Ports* tab (bottom layout of Vivado) ▶ Unroll *Scalar Ports* to reveal `busy_0` port ▶ Set columns *I/O Std* to `LVCMOS33` and *Package Pins* to `R14` (see fig-F.51 taken from the online Reference Manual of the board). See also fig-F.52.

`Ctrl+S` (or button in main GUI toolbar) ▶ *Save Constraints* ▶ Set *File name* to `ecc-constraints`, keep `XDC` for *File Type* and ⟨`Local to Project`⟩ for *File location* (fig-F.53) ▶ OK.

A new constraint file named ⟨`ecc-constraints.xdc`⟩ is created and added to the project, as you can see in the *Sources/Hierarchy* tab under the *Constraints* heading (fig-F.54).

Double-click on this file to display its content, the two `set_property` Tcl commands (fig-F.55) are more or less human-readable.

Flow Navigator ▶ *SYNTHESIS* ▶ *Run Synthesis* ▶ *Launch Runs* ▶ Set *Launch Runs on local host* with the max number of jobs (that should be the nb of CPUs in your guest machine) ▶ OK.

Synthesis takes a few minutes, when this is done:

Synthesis Completed window ▶ *Run Implementation* ▶ OK ▶ *Launch Runs* window ▶ Check *Launch runs on local host* + set *Number of jobs* to the max ▶ OK.

This launchs the place & route operations which take ∼5' (you can get short infos on what Vivado is currently doing in right top corner of main GUI, e.g Running opt_design):

Implementation Completed window (fig-F.57) ▶ *Open Implemented Design* ▶ OK (just agree if a pop-pup window appears inviting you to first close the elaborated or synthesized design).

Vivado opens the implemented design and displays the logic layout now mapped in the PL part of the device (see *Device* tab).

☞ **It is highly probable that Vivado will display two critical messages windows** (fig-F.58 and fig-F.59). Close these with OK.

30 We won't detail here the messages regarding the Methodology Violations, please see same numbered step **30** of the verbose version of the tutorial for an in-depth discussion.

31 Likewise, step **31** of the verbose tutorial will give you comprehensive information on the reasons why Vivado is issuing timing errors.

Bottom line: there are two clock-domains exchanging signals in the IP and we haven't set any timing constraint on these signals. When software configures the IP by writing some of its registers, this might affect signals that are driving logic into the clock-domain of the Montgomery multipliers. But this is not a big deal because by the time the Multipliers will carry out the next REDC operation, the asynchronous signals will already be stabilized a long ago (the RTL doesn't even resynchronise them in the target clock-domain). In other words, we simply need to declare the clock-crossing paths of these signals as **false paths**:

Edit file ⟨`ecc-constraints.xdc`⟩ and add this line at the bottom of the file (just below the two `set_property` lines):

```
set_clock_groups -asynchronous -group [get_clocks clk_fpga_0] -group [get_clocks clk_fpga_1]
```

Run Implementation again (Vivado left column) (ignore the *Synthesis is Out-of-date* warning with *Yes*) ▶ When Vivado has finished re-running the whole synthesis and implementation flow: *Implementation Completed* ▶ *Open Implemented Design* again ▶ OK.

This time, the timing requirements warning shouldn't be issued. As for the critical warnings, from now on we'll simply ignore them. The layout of the design can be seen in the *Device* tab of Vivado right panel (fig-F.66).

32 Please refer to step **32** of the verbose tutorial if you're interested in exploring a bit the physical layout of the design and assess the share of the different subcomponents of the IP in the overall area it occupies.

33 **Bitstream generation and export**

Flow Navigator ▶ *PROGRAM AND DEBUG* ▶ *Generate Bitstream* ▶ *Launch Runs* ▶ OK (default settings) [this is quick] ▶ window *Bitstream Generation Completed* (fig-F.70) ▶ *View Reports* ▶ OK.

File ▶ *Export* ▶ *Export Hardware*.

Window *Export Hardware Platform* ▶ *Next* ▶ *Include bitstream* (not *Pre-synthesis*) ▶ *Next* ▶ Set *XSA file name* to `az7-ecc-axi` (do not add the `.xsa` suffix yourself), keep default export path (~/ipecc/hw/`soc`) ▶ *Next* ▶ *Finish*.

Vivado performs the export quietly so don't be surprised not to get any sucessfull-like message (you can check anyway that file ⟨`az7-ecc-axi.xsa`⟩ has been actually created where expected).

☞ Here ends the hardware part of the tutorial. Next paragraph will give a demonstration of how to use the IP from standalone (aka bare metal) software.

If you intend to drive the IP from Linux afterwards, **gain some time** by starting the compilation of Linux **now**: for that make a small detour to step **44** then resume back the tutorial here, leaving the build running in the background.

Software part: driving IPECC from bare metal software

34 *Tools* ▶ *Launch Vitis IDE*

Be patient if you've taken the advice of previous section and Linux is being compiled in the background, as this considerably slows Vitis down ...

Keep default choice for the workspace (~/`workspace`, fig-F.72).

35 *Create Application Project* (fig-F.73) ▶ *New Application Project* window ▶ *Next* ▶ tab *Create a new platform from hardware (XSA)* ▶ XSA file: ~/ipecc/hw/soc/`az7-ecc-axi.xsa` (leave *Generate Boot Components* box checked, fig-F.74) ▶ *Next* ▶ *Application project name*: `ecc-test-stdalone` (fig-F.75) ▶ *Next* ▶ *Domain*: nothing to do ▶ *Next* ▶ *Templates:* default `Hello World` ▶ *Finish*.

36 *Explorer* (fig-F.78) ▶ unroll `ecc-test-stdalone_system` › `ecc-test-stdalone` › `src` ▶ `helloworld.c` double-click

Customize the printf message e.g `"Hello World, this is IPECC test program.\n\r"`. This is to be sure that what you get is what you compiled and not some default program already residing on the board. You may also remove the second printf (fig-F.79).

☞ We're going to run this very simple *Hello World* program as is (without any IPECC related stuff for now) to first **test/establish the connectivity of the board with your personal machine.**

See steps **37**-**38** of the verbose tutorial for more info on host config to allow the guest to acquire the USB interface of the board when you plug it in.

37 *See same numbered step* **37** *in the detailed version of the tutorial.*

38 *See same numbered step* **38** *in the detailed version of the tutorial.*

39 Set jumper `JP4` as illustrated on fig-F.84 (JTAG configuration).

Remove microSD card from slot `J9` if any.

Power on the board: depending on position of jumper `JP5` it can be powered from either a USB cable or from a AC-DC wall wart adapter: set jumper on `REG` in the former case (fig-F.132) and to `USB` in the latter (fig-F.133).

Now, whatever the way you chose to power the board, plug it into your machine using a USB cable (micro-USB `J14` connector, labelled **PROG UART**) because we now want to communicate with it.

Try an enumeration of the `ttyUSB`-like pseudo device files, on the **Host** machine for now (not on the guest):

CLI `$ ls -1 /dev/ttyUSB*` `# On host, not on guest`

```
/dev/ttyUSB0
/dev/ttyUSB1
```

40 Have the guest snatch the USB interfaces of the board from the host:

Devices ▶ *USB* ▶ *Digilent Adept USB Devices [0700]*.

See step 40 of the detailed version of the tutorial to see how to create a USB VirtualBox filter for having the guest VM «snatching» the USB interfaces automatically upon each plug.

41 **Xilinx cable installation** (adapt this to your Vivado install path)

CLI `$ cd ~/xilinx/Vivado/2022.1/data` `# Again in the guest`

CLI `$ cd xicom/cable_drivers/lin64/install_script/install_drivers`

CLI `$ sudo ./install_drivers`

```
[sudo] password:
```

(fig-F.88)

Unplug the board and plug it back again to have the configuration changes to take effect (and perform again step 40 if you didn't set any VBox USB filter).

42 CLI `$ sudo apt-get install screen`

CLI `$ screen /dev/ttyUSB0 115200`

Right-click on project name `ecc-test-stdalone` ▶ *Build Project* (fig-F.89 & fig-F.90).

Right-click again on project name `ecc-test-stdalone` ▶ *Run As* ▶ *Run Configurations* (fig-F.91) ▶ *Run Configurations* window ▶ *Single Application Debug* (fig-F.92) ▶ button ▶ *Run* (fig-F.93).

After a few seconds you should see the message string from the `printf` appearing in the `screen` console (fig-F.95).

43 If you got the Hello World message, it means the Xilinx tools can interact properly with hardware, so **let's play a bit with IPECC now!**

CLI `$ cd ~/workspace/` `# Or whatever place you chose for Vitis workspace`

CLI `$ cd ecc-test-stdalone/src`

CLI `$ ln -s ${IPECC}/driver/hw_accelerator_driver.h hw_accelerator_driver.h`

CLI `$ ln -s ${IPECC}/driver/hw_accelerator_driver_ipecc.c hw_accelerator_driver_ipecc.c`

CLI `$ ln -s ${IPECC}/driver/hw_accelerator_driver_ipecc_platform.c hw_accelerator_driver_ipecc_platform.c`

CLI `$ ln -s ${IPECC}/driver/hw_accelerator_driver_ipecc_platform.h hw_accelerator_driver_ipecc_platform.h`

CLI `$ ln -s ${IPECC}/driver/hw_accelerator_driver_socket_emul.c hw_accelerator_driver_socket_emul.c`

CLI `$ ln -s ${IPECC}/driver/stdalone/ecc-test-stdl.c ecc-test-stdl.c`

CLI `$ ln -s ${IPECC}/driver/stdalone/ecc-test-stdl.h ecc-test-stdl.h`

CLI `$ ln -s ${IPECC}/hdl/common/ecc_curve_iram/ecc_addr.h ecc_addr.h`

CLI `$ ln -s ${IPECC}/hdl/common/ecc_curve_iram/ecc_vars.h ecc_vars.h`

CLI `$ ln -s ${IPECC}/hdl/common/ecc_curve_iram/ecc_states.h ecc_states.h`

As you add along symbolic links (with the `ln` commands above) you can see that Vitis/Eclipse dynamically detects the presence of the new source files and automatically adds them to the project (fig-F.98).

Right-click on file ⟨`helloworld.c`⟩ ▶ *Delete* ▶ OK .

Right-click again on project name `ecc-test-stdalone` ▶ *Properties* ▶ Unroll *C/C++ Build* (left colum) ▶ *Settings* ▶ *ARM v7 gcc compiler* ▶ *Symbols* ▶ *Defined symbols (-D)* (right-section) ▶ button ▶ pop-up window: enter value `WITH_EC_HW_ACCELERATOR` ▶ OK.

Repeat the operation ( ) to also add the three preprocessor symbols `WITH_EC_HW_ACCELERATOR_WORD32`, `WITH_EC_HW_STANDALONE`, `WITH_EC_HW_STANDALONE_XILINX`.

Apply and Close.

Rebuild application: right-click on `ecc-test-stdalone` ▶ *Build Project*.

Check that the `screen` application we opened in step 42 is still running (otherwise relaunch it).

Right-click again on `ecc-test-stdalone` ▶ *Run As* ▶ *Run Configurations* ▶ *Target Setup* ▶ check the four boxes are checked as in fig-F.99 (in particular the *Program FPGA* one) ▶ *Run*.

Vitis/Eclipse programs the FPGA, transfers the binary image of the application in DDR memory and branch the Cortex processor to its starting point. The application here consists in programming IPECC with a sequence of tests which are defined in the source file ⟨`ecc-test-stdl.h`⟩. If you take a look at this file, you'll see that it contains at the bottom the definition of an array named `ipecc_all_tests` of elements of type `ipecc_test` (which is defined at the top of the file). Two macros named `IPECC_TEST_VECTOR_NOQ` and `IPECC_TEST_VECTOR_Q` are defined that allow defining IP tests using a simple one line C statement. As stated earlier, the tests here are defined statically (at compilation time). You can see that several tests are defined but only three are uncommented by default, a test on a 24-bit curve with a base point of order 2, a test on a 127-bit curve and a test on a 256-bit curve. The log you should get from the application as displayed on the `screen` console is shown on fig-F.100.

Let's check correctness of e.g the 256-bit $[k]\mathcal{P}$ result:

CLI `$ sage`

Use `Ctrl`+`Mouse` to copy-paste from the `screen` console into the *Sage* one all the parameters of the 256-bit test (from line `nn=256` to line `Pouty=0x...`):

```
sage: %colors Linux                          # Better put this in your ⟨~/.sage/init.sage⟩
sage: nn=256
....: a=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
....: b=0xee353fca5428a9300d4aba754a44c00fdfec0c9ae4b1a1803075ed967b7bb73f
....: p=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c03
....: q=0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1
....: k=0xf1adb2506355162d0de14468748fb171f730bd40f6595fe1732651df00589fcf
....: Px=0xb6b3d4c356c139eb31183d4749d423958c27d2dcaf98b70164c97a2dd98f5cff
....: Py=0x6142e0f7c8b204911f9271f0f3ecef8c2701c307e8e4c9e183115a1554062cfb
....: Poutx=0x4c0338872b9f13aa0c01f74c7f504eeae9d947f54e9bb80dfce61c3f2e5d151d
....: Pouty=0xa9571e1e0447b7fb302eab3a57dae4866f238eac48f67993a37cc660a831879b
....:
```

Now enter the following commands one by one under *Sage* prompt (fig-F.101):

```
sage: Fp=GF(p)                                                  # Defines the field
sage: EE=EllipticCurve(Fp, [a, b])                              # Defines the curve
sage: P=EE(Px, Py)                                              # Defines the point
sage: Q=k*P                                                  # Computes [k]P result
sage: hex(Q[0])                                  # Display X-coordinate of the result
'0x4c0338872b9f13aa0c01f74c7f504eeae9d947f54e9bb80dfce61c3f2e5d151d'
sage: hex(Q[1])                                  # Display Y-coordinate of the result
'0xa9571e1e0447b7fb302eab3a57dae4866f238eac48f67993a37cc660a831879b'
sage: Q[0] == Poutx                          # Compare X-coord of Sage result with ours
True                                                          # It's a match indeed
sage: Q[1] == Pouty                          # Compare Y-coord of Sage result with ours
True                                                            # It's also a match
sage: Q == EE(Poutx, Pouty)                                          # Double check
True                                                         # 'think we are safe
```

Notice on the `screen` console (fig-F.100) how the IP detects possible nullity of the resulting point. For $[k]\mathcal{P}$ computations result is stored in point $\mathcal{R}_1$, hence we can see that for instance in the `nn=32` case the result is the null point. This was expected from the fact that the input point is of order 2 (as shown by its Y-coordinate being 0) and that the scalar k is even.

Software part: driving IPECC on top of Linux

44 PetaLinux prerequisites

Make sure you have a working **Internet connexion** during PetaLinux config and build. PetaLinux tools download a lof of stuff, especially during the first build.

CLI `$ sudo dpkg-reconfigure dash                  # Select 'No' to replace dash with bash`

CLI `$ sudo apt-get install gawk net-tools autoconf libtool gcc gcc-multilib zlib1g:i386`

45 PetaLinux download and install

Xilinx PetaLinux download web page (g**gle-in) ▶ select **PetaLinux Tools - Installer 2022.1 Full Product**. **Installation** (fig-F.103)

CLI `$ cd ~/Downloads # Or wherever the place you download things`

CLI `$ md5sum petalinux-v2022.1-04191534-installer.run`

```
5ea0aee3ab9d4c1b138119b0b6613a17    petalinux-v2022.1-04191534-installer.run
```

CLI `$ mkdir ~/petalinux`

CLI `$ mv ~/Downloads/petalinux-v2022.1-04191534-installer.run ~/petalinux`

CLI `$ cd ~/petalinux`

CLI `$ chmod +x petalinux-v2022.1-04191534-installer.run`

CLI `$ ./petalinux-v2022.1-04191534-installer.run`

```
INFO: Checking installation environment requirements...
WARNING: This is not a supported OS ...
WARNING: No tftp server found - please refer to "UG1144 PetaLinux...
```

(ignore the two warnings).

CLI `$ source ~/petalinux/settings.sh # For every new terminal where you run Peta tools`

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-create --type project --template zynq --name az7-ecc-axi`

```
INFO: Create project:  az7-ecc-axi
INFO: New project successfully created in /home/.../ipecc/sw/linux/az7-ecc-axi
```

CLI `$ cd az7-ecc-axi`

CLI `$ petalinux-config --get-hw-description ~/ipecc/hw/soc/az7-ecc-axi.xsa` (fig-F.104)

Don't configure anything for now, save a default config: ⟨Exit⟩ + ⟨Yes⟩.

CLI `$ petalinux-build`

```
[INFO] Sourcing buildtools ...
```

> ☞ The complete build takes from **two to three hours**. If you ran this step as the detour that we advised in step **33**, you can now resume back the tutorial to the standalone driver part in page 110, otherwise you'll have to be patient.

46 Customizing Linux for our hardware

In a new terminal:

CLI `$ cd ~/ipecc/sw/linux/az7-ecc-axi`

CLI `$ source ~/petalinux/settings.sh`

CLI `$ petalinux-config -c kernel # Be patient as Yocto parses its recipes`

```
...
```

Device drivers ▶ *Userspace I/O drivers* ▶ `<*>` (figs-F.105-F.106, possibly already selected).

Device drivers ▶ *Userspace I/O drivers* ▶ *Userspace I/O platform driver with generic IRQ handling* ▶ `<M>` (fig-F.107, possibly already selected)

Exit + Save

CLI `$ sudo apt-get install device-tree-compiler`

CLI `$ cd images/linux`

CLI `$ dtc -I dtb -O dts -o system.dts system.dtb # Ignore the many warnings`

Open ⟨`./system.dts`⟩, search for first occurence of "`ecc@`" (fig-F.108). Notice value `0x40000000` of IP base address. This matches value set by Vivado in SoC Design Block[5]. More importantly, observe the `compatible` field: it is set with a dummy value (`xlnx,ecc-1.0`) we need to change. Close wo/ saving.

CLI `cd ~/ipecc/sw/linux/az7-ecc-axi`

CLI `cd project-spec/meta-user/recipes-bsp/device-tree/files`

Edit file ⟨`./system-user.dtsi`⟩ and add these three lines at the bottom:

[5] If you open project `az7-ecc-axi` back in Vivado, open the Block Design and go to the *Address Editor* tab (fig-F.109) you'll see that the AXI response port of hardware instance `ecc_0` of the IP was given base address `0x40000000` in the AXI system bus address space, along with a size of 4KB that corresponds to the usual page size on 32-bit systems.

```
/* IPECC */
&ecc_0 {
   compatible = "generic-uio";
};
```

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-build`

When the build is over, decompile again file ⟨`system.dtb`⟩:

CLI `$ cd images/linux`

CLI `$ dtc -I dtb -O dts -o system.dts system.dtb` `# Ignore the many warnings`

Open ⟨`./system.dts`⟩ and confirm that `compatible` field is now set to `generic-uio` (fig-F.110).

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-config` (fig-F.104)

Image Packaging Configuration ▶ *Root filesystem type* ▶ `EXT4 (SD/eMMC/SATA/USB)` (fig-F.111).

Exit + Save

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-build`

47 **Customizing rootfs** – We're now going to generate the root filesystem by adding the following components/programs to our small embedded distribution:

- a real shell like `bash`
- an SSH server so as to be able to log onto the Arty board through the network
- The `netcat` program (the swiss-army knife for network applications).

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-config -c rootfs`

Filesystem Packages ▶ *base* ▶ *shell* ▶ *bash* ▶ `[*]` (fig-F.114)

Filesystem Packages ▶ *net* ▶ *netcat* ▶ *netcat* ▶ `[*]` (fig-F.115)

Filesystem Package ▶ *misc* ▶ *packagegroup-core-ssh-dropbear* ▶ *packagegroup-core-ssh-dropbear* ▶ `[*]` (fig-F.116)

Filesystem Packages ▶ *misc* ▶ *coreutils* ▶ *coreutils* ▶ `[*]`

Exit + Save

CLI `$ petalinux-build -c rootfs`

When the build is over, the root filesystem should be in different tarball formats in folder `images/linux` along with all other binary files (FSBL, U-boot, kernel, etc).

48 **Customizing bootargs of the kernel**

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-config`

DTG Settings ▶ *Kernel Bootargs* ▶ Unselect «generate boot args automatically» ▶ Edit argument line (press Enter to start edit).
Replace what probably looks like this:

`console=ttyPS0,115200 earlycon root=/dev/ram0 rw`

by this (fig-F.112 and fig-F.113):

`earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk0p2 rw rootwait uio_pdrv_genirq.of_id=generic-uio`

Exit + Save

CLI `$ petalinux-build`

49 Wrap everything into a Xilinx specific file ⟨`BOOT.BIN`⟩:

CLI `$ cd ~/ipecc/sw/linux`

CLI `$ petalinux-package --force --boot --fsbl --u-boot --fpga images/linux/system.bit`

Ignore the two warnings you'll probably get about overlapped partitions range (`:-.`)

50 **Creating a bootable SD card** – As this is the «expert» version of the tutorial, we'll skip everything related to the creation and partitioning of the microSD card (please refer to step **50** of the detailed version of the tutorial if you need to). We'll assume, before going ahead with next step, that you got a microSD card (Arty board isn't shipped with one) of let's say 4GB, and partitioned it according to these settings:

- first partition: `fat32`, 500 MiB, label e.g `BOOT`, set `boot` flag
- second partition: `ext4`, all remaining space, label e.g `ROOTFS`.

51 **Finalizing the bootable SD card** – Assuming mount points `/media/myself/BOOT` and `/media/myself/ROOTFS` for the two SD card partitions:

CLI `$ cd ~/ipecc/sw/linux/images/linux`

CLI `$ cp BOOT.BIN image.ub boot.scr /media/myself/BOOT/`

CLI `$ sync # To flush-write all files onto the SD card`

CLI `$ sudo tar -C /media/myself/ROOTFS -xzf rootfs.tar.gz`

CLI `$ sync`

Edit file ⟨`.bashrc`⟩ of folder `/media/myself/ROOTFS/home/petalinux` and uncomment line 7 and lines 9 to 11. Also modify value of the shell prompt variable `PS1` to `'\[\033[01;33m\]\h\$\[\033[00m\] '` (fig-F.129), save and exit.

CLI `$ sync`

CLI `$ sudo umount /dev/mmcblk0p1`

CLI `$ sudo umount /dev/mmcblk0p2`

52 **Booting Linux**

Remove the microSD card and plug it into slot `J9` of the Arty board (fig-F.130).

Change position of jumper `JP4` as illustrated on fig-F.131 to now have the board to boot from SD card.

Depending on position of jumper `JP5` the board can be powered from either a USB cable or from a AC-DC wall wart adapter: set jumper on `REG` in the former case (fig-F.132) and to `USB` in the latter (fig-F.133).

Plug the USB cable supplied with the board into your host machine on its type-A end () and into the `J14` connector of the Arty board on its micro-type-B end ().

With the guest still running, and given the presets we made in steps **37**-**38**, the VM should immediately capture the two interfaces `/dev/ttyUSB0` and `/dev/ttyUSB1`, in which case you can run in a new terminal:

CLI `$ screen /dev/ttyUSB1 115200`

Here are the steps of the boot (fig-F.134):

1. Xilinx FSBL configures the FPGA very fast (see `DONE` green led)
2. U-boot catches script file ⟨`boot.scr`⟩ and executes it
3. The kernel is loaded and starts its boot process
4. The login prompt is displayed, use `petalinux` and select a password (fig-F.135)

53 **Connect the board to your local network**

☞ Refer to step **53** of the verbose tutorial for more information on how to do this.

54 **Building IPECC driver**

You can't compile the driver without first editing the Makefile (as explained hereafter)

CLI `$ sudo apt-get update`

CLI `$ sudo apt-get install gcc-arm-linux-gnueabihf`

CLI `$ cp -Rf ${IPECC}/driver ~/ipecc/sw/linux/driver`

CLI `$ cd ~/ipecc/sw/linux/driver`

Edit `./Makefile` and modify variable `VHD_DIR` to point to the folder where IP microcode was built (see step **5**), meaning `/home/myself/IPECC/hdl/common/ecc_curve_iram`.

Do not use a relative path when setting the variable! Only absolute paths are supported by `make`.

CLI `$ make ecc-test-linux-uio`

```
arm-linux-gnueabihf-gcc -Wall -Wextra -Wpedantic -O3 ...
```

```
hw_accelerator_driver_ipecc.c:78:27:  warning:  ...
```

The executable is named ⟨ecc-test-linux-uio⟩ and is built in place (check it's actually been created).

Get network IP address of the board:

```
arty $ ip a                                                    # In the screen console
```

We'll assume that the IP is 192.168.111.38 (adapt this to your settings).

```
CLI $ scp ecc-test-linux-uio petalinux@192.168.111.38:.
CLI $ ssh petalinux@192.168.111.38                  # Put aside screen console after that
arty $ sudo chmod 666 /dev/uio0
```

Run the IPECC test application program:

```
arty $ stdbuf -oL nc -l -p 20000 | ./ecc-test-linux-uio        # 20000 is an arbitrary port
Driver in UIO mode
IP in debug mode (HW version 1.2.25)
```

The test application stays idle, waiting for test-vectors to be pushed to its standard input in the same format as the one we used when we simulated the IP (steps 16 - 25).

55 Testing the IP over the network

```
CLI $ cd ~/ipecc/sw/sage
```

Edit file ⟨generate-tests.sage⟩.

☞ More info on this script, the way it works and what it's used for is given in the verbose tutorial, step 55.

Now search for the following parameters below in the script and set each one to the value that is given alongside:

```
nnmaxabsolute = 256    # no curve/test will be of field size nn > 256
nnminmax = 256         # after a while only 256-bit curves will be generated
NNMINMOD = 10          # nnmin will increase every round of 10 curves...
NNMININCR = 32         # ...and it will be increased by 32
NNMAXMOD = 5           # nnmax will increase every round of 5 curves...
NNMAXINCR = 32         # ...and it will be increased by 48
nn_constant = 0        # we don't want a constant value of nn
NBCURV = 0             # the script will iterate indefinitly
NBKP = 100             # 100 scalar-multiplication tests generated per curve
NBADD = 10             # 10 point-addition tests generated per curve
NBDBL = 10             # 10 point-doubling tests generated per curve
NBNEG = 10             # 10 point-negate (−P) tests generated per curve
NBCHK = 10             # 10 boolean "is P on curve?" tests generated per curve
NBEQU = 10             # 10 boolean "are points equal?" tests per curve
NBOPP = 10             # 10 boolean "are points opposite?" tests per curve
NO_EXCEPTION = False   # the tests will include exception cases
```

Save and exit.

```
CLI $ sage generate-tests.sage | tee /tmp/arty-z7.txt | stdbuf -oL nc 192.168.111.38 20000
Generating curves from nn = 32 to 48
Generating curves from nn = 32 to 80
Generating curves from nn = 64 to 112
...
```

In the SSH console on the board you should start seeing the IP test program displaying statistics on the tests received (fig F.137). At the begining tests received are on small curves (nn ∼ 32). Cryptographic sizes (e.g nn=256) are reached after a few minutes. The point operations performed by the IP are organized in colums, from left to right:

1. Scalar multiplication (computes $[k]\mathcal{P}$) in column labelled [k]P
2. Point addition (computes $\mathcal{P}+\mathcal{Q}$) in column labeled P+Q
3. Point doubling (computes $[2]\mathcal{P}$) in column labeled [2]P
4. Point negation (computes $-\mathcal{P}$) in column labeled -P

and the three logical tests:

5. Are the two points equal? (tests if $\mathcal{P}=\mathcal{Q}$) in column labeled P==Q
6. Are the two points opposite? (tests if $\mathcal{P}=-\mathcal{Q}$) in column labeled P==-Q

7. Does this point belong to the curve? (tests if $\mathcal{P} \in \mathcal{C}$) in column labeled `PonC`.

For each operation the array shows the number of tests submitted to the IP, the number of correct tests (in green) and the number of errors (in red). The default behaviour of the test program is to break execution as soon as an error is encountered.

> ☞ Needless to say no error is expected to occur! **The IP has been thoroughly tested both in simulation and on real hardware FPGA platforms.** Many errors, corner cases and side effects were found and fixed, and the RTL is now **highly stable**. At time of version 1.2.25 there remains two known bugs of minor importance which are described in §?.
>
> Should you encounter a new error case anyway, please don't hesitate to submit a bug report to the authors.

If you observe the content of file `/tmp/arty-z7.txt` while the test vectors are being transfered from the host machine to the board, e.g by using command `tail -f /tmp/arty-z7.txt` in a new terminal, you may notice that the value of `nn` seems to increase faster in the file than what it seems on the board according to the average value of `nn` that is being displayed. This is because the board computes $[k]\mathcal{P}$ much slower than the host machine does in *Sage*! The FPGA on the board runs the two Montgomery multipliers inside the IP at 250 MHz which is one magnitude of order less than let say the 2 GHz which clocks a x86 multicore processor these days. The FPGA being the bottleneck, test-vectors are buffered both by the embedded Linux and the host/guest machines, which is why they'll be dumped in file `/tmp/arty-z7.txt` sometimes several minutes before being actually processed by the IP. You can assess this by hitting `Ctrl-C` in the host/guest terminal running the *Sage* script (not on the board), the IP will continue processing test vectors for several minutes before quitting.

You can interrupt the IP test program by hitting `Ctrl-C` in the board console, which will also interrupt the sending process on the host machine as result of TCP socket termination.

56 **Updating PL bitstream at runtime** – See step **56** of the verbose tutorial.

F.2 Detailed version of the tutorial

Ubuntu installation

1 Download the **Ubuntu Desktop 22.04.2** Linux distribution and install it as a virtual machine.

- Use file `ubuntu-22.04.2-desktop-amd64.iso`
- Source: all over the Internet, spec. here: `https://releases.ubuntu.com/jammy/`
- sha256 fingerprint: `b98dac940a82b110e6265ca78d1320f1f7103861e922aa1a54e4202686e9bbd3`

This is the **VM configuration** we used (use a configuration as closest as possible to these settings or at least as performant):

- 16 GB of memory, 4 processors, 250 GB of disk (Vivado disk occupation size has outrageously increased the past few years, not to mention the 60 GB you'll need to install and compile PetaLinux)
- All VT-x/AMD-V acceleration features enabled, PAE/NX, IO-APIC, nested pagination
- 64 MB of Video memory

We used **VirtualBox 6.1** as our virtual machine manager but the choice of another one obviously is up to you.

Install the **English (US) version** of Ubuntu so as to get the proper `en_US.UTF-8` locale. FPGA tools occasionaly show to be sensitive to locales. Working with a US version will save you some trouble.

During install configuration, select default *Normal installation* (instead of *Minimal installation*) but uncheck the *Download updates while installing Ubuntu* (fig-F.2). This is to ensure that you get exactly the same software distribution as we used on our side and that no side effect appears with Vivado processings due to other installation. Later on, once Ubuntu is up and running, it will frequently prompt you with a window (fig-F.3) proposing you to update the system. For the same reason, select then *Remind Me Later*.

2 Once installation is complete (takes ∼ 15') and your system is up and running in the virtual machine, install the packages below:

From your Linux shell prompt

```
$ sudo apt-get update
$ sudo apt-get install gcc dkms build-essential perl vim
```

3 Make sure to add the *Guest Additions* features so that you get an enriched user-experience of your virtual system. Particularly the screen resolution should become more confortable. Also performances of the guest machine will improve. In VirtualBox, this is obtained within the *Devices* menu of the virtual client by choosing *Insert Guest Additions CD Image*. This will have the guest OS compile some drivers, this requiring the packages you previously installed in step 2 (`vim` being added for source files editing only). Alternatively you can do this through command line interface:

From your Linux shell prompt

```
$ sudo apt-get install virtualbox-guest-utils
```

Troubleshooting. The management of the Virtual Guest Additions CD by VirtualBox sometimes happens to be mischievous. The following help page might be of some assistance if you get stucked with a VirtualBox insistently reporting it can't achieve to insert the Guest Additions CD:
`https://askubuntu.com/questions/573596/unable-to-install-guest-additions-cd-image-on-virtual-box`

IPECC GitHub repo fetch & install

☞ All subsequent actions will now obviously have to be realised in the virtual machine (unless otherwise mentioned, we won't specify «in the VM» anymore).

4 If you haven't fetched the official IPECC GitHub repository already, let's do it now (we'll isolate our local copy of the repository at user's home root folder, and later on in the tutorial we will work elsewhere not to interfere with the repo sources):

From your Linux shell prompt

```
$ sudo apt-get install git
...
$ cd
$ git clone https://github.com/ANSSI-FR/IPECC.git
Cloning into 'IPECC'...
...
$ cd IPECC
$ ls
doc driver hdl LICENSE README.md sage sim syn
```

☞ From now on we'll refer to the local copy of the IPECC repo. using the `${IPECC}` shell variable.

From your Linux shell prompt

```
$ export IPECC=~/IPECC                    # Better put this in your ⟨~/.bashrc+⟩ file
```

5 The first thing to do after downloading the repository is to go to folder `hdl/common/ecc_curve_iram/` and run the `make` command to **build the microcode of the IP** from the assembly source files as long as two VHDL files that also depend on these:

From your Linux shell prompt

```
$ cd hdl/common/ecc_curve_iram/
$ make
-> Creating ASM main program ecc_curve_iram.s
-> Parsing ../ecc_pkg.vhd, ../ecc_customize.vhd and asm_src/vardefs.csv for
checking/updating our constants
[+] Parsing of VHDL files done, everything OK
...
```

Ignore the possible yellow lines warnings (the log of Make command above was shortened for sake of readability). Simply check that the three VHDL files named ⟨`ecc_curve_iram.vhd`⟩, ⟨`ecc_addr.vhd`⟩ and ⟨`ecc_vars.vhd`⟩ have been generated (listed in blue below) as long as the three C header files ⟨`ecc_addr.h`⟩, ⟨`ecc_states.h`⟩ and ⟨`ecc_vars.h`⟩ (listed in orange below):

From your Linux shell prompt

```
$ ls -1
asm_src
ecc_addr.h
ecc_addr.vhd
ecc_curve_iram.s
ecc_curve_iram.vhd
ecc_states.h
ecc_vars.h
ecc_vars.vhd
ipecc_assembler.py
latex
Makefile
```

The blue files are required for both simulation and synthesis of the IP. The orange files allow maintaining the driver consistant with the hardware API (they are only used by software).

6 Optionnally you can also type `make latex` to build a neat and colorized pdf listing of the complete assembled microcode, which will be more agreeable to read than the assembly source files located in `asm_src/` folder. But this will require that you first install the `texlive-full` package, which will take half an hour):

From your Linux shell prompt

```
$ sudo apt-get install texlive-full
...  [takes a while]
$ make latex
| Process ASM files
| Generate the texify awk script
| Texify the source
| LATEX ecc_curve_iram.tex
$ evince latex/ecc_curve_iram.pdf &
```

Two remarks:

- Don't mistake the fact that the first page of the generated pdf file is empty with thinking that the whole document is. The fact that the first page is empty is a small bug but that's of no importance.
- You shouldn't run `make` directly in folder `latex` or you will obtain an empty pdf. This is because the `Makefile` in folder `ecc_curve_iram/latex` requires some variables that are passed by the `Makefile` in folder `ecc_curve_iram` which normally calls it.

7 We'll copy what we need along the way from the git repo in a different folder, let's call it ~/`ipecc`, so as not to interfere with the repo itself:

From your Linux shell prompt

```
$ cd
$ mkdir -p ipecc/hw/ip           # (HW) IP sources that we're going to customize
$ mkdir    ipecc/hw/ip/packaged  # (HW) IP sources wrapped in an exportable bundle
$ mkdir    ipecc/hw/soc          # (HW) Sources for complete SoC
$ mkdir -p ipecc/sw/sage         # (SW) Sage scripting
$ mkdir    ipecc/sw/linux        # (SW) Linux install & build
```

Folder `ipecc/hw` (resp. `ipecc/sw`) will contain everything related to hardware (resp. software). In folder `ipecc/hw/ip` we will edit and create a version of IPECC customized to our hardware configuration. In `ipecc/hw/soc` we will create and edit a Vivado project targeting the Zynq device of the Arty Z7 board, including not only the IP but also the Cortex processor and the interconnect fabric to have the two pieces of hardware communicate with each other. Folder `ipecc/sw/linux` will be used for configuration and compilation of Linux along with an application sample code accessing the IP from the Linux userspace using the *generic device UIO* (user space I/O generic driver) API to the kernel.

Vivado installation

8 Go to the Xilinx Vivado download web page and choose the **Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer** (fig-F.4). You'll have to register to the Xilinx web site first, so create an account (for free) in case you don't already own one.

Change dir. to the folder where you've downloaded the file, verify its checksum, add the executable flag to it and then run the script (you don't need to run it as sudoer):

From your Linux shell prompt

```
$ md5sum Xilinx_Unified_2022.1_0420_0327_Lin64.bin
db5056feaaf271fe90ba54bae4768ed2 Xilinx_Unified_2022.1_0420_0327_Lin64.bin
$ chmod +x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
$ ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
```

9 Ignore the pop-up window inviting you to install the latest release of Vivado (fig-F.5). Also don't bother about the red warning threatening you that you are on an unsupported operation system. It seems that whatever the OS you're running Vivado on you'll get it anyway (fig-F.6).

Choose **Vitis**. Vitis includes Vivado and adds the SDK part of the development tools in the form of an Eclipse based software IDE (name Vitis was introduced with release 2019.2 of the Xilinx toolchain to replace what was merely designated as SDK). Choosing to install Vivado alone will save you 40 GB of disk space. We could compile IP test software using Debian package ARM GCC cross-compiler but we'll need Vitis to download the software to the Arty board through the JTAG cable ...

10 The type of devices you choose to get support for will also have a great influence on the amount of disk space occupied by the Xilinx tools. For this tutorial we need to target the `xc7z010` device which is of Zynq-7000 family, so you can simply check the Zynq-7000 box and have all other boxes unchecked (fig-F.7). Be sure however to leave the box *Install devices for Alveo and Xilinx edge acceleration platforms* checked, as it'll install required libraries.

The required disk space should amount to a little more than 165 GB (:-.) The installation process may be quite long depending on your bandwidth (approx. 1 h) so be patient.

Running Vivado

11 Using the Xilinx tools requires that you first source a specific script to set some paths. This script is named ⟨`settings64.sh`⟩ for Bash and is located in the Vitis/Vivado installation folder. However it's not a good idea to have this script automatically sourced by your shell run-control script (⟨~/`.bashrc[+]`⟩ for Bash) because the Xilinx script sets the `PATH` and the `LD_LIBRARY_PATH` environment variables in such a way that it might interfere with other programs. In particular it has already been observed that some FPGA vendors were using their own specific brand of the libc library and also of the Java virtual machine (:-.) It's therefore a better practice to source the file ⟨`settings64.sh`⟩ in a terminal window only at the time you intend to use Vivado:

From your Linux shell prompt

```
$ source ~/xilinx/Vitis/2022.1/settings64.sh
```

☞ If you've installed Vivado only (without Vitis) it's probable that you'll have to replace `Vitis` with `Vivado` in the path above. In any case you can go in the folder where you made the installation and run a `find . -name settings64.sh`. This will give the location of the ⟨`settings64.sh`⟩ file you need to source. You may get several answers from `find` if you've installed both Vivado and Vitis, in which case any of them will do.

The last command above assumes that you previously gave ~/`xilinx` as the root path to the Vivado installation tool in steps **8**-**10** (obviously adapt this line to your own installation directory).

12 **Kludge #1** – From this point, launching Vivado from the command prompt will still not work (as seen on sample terminal code below) because a dynamically shared library is still missing (which you wouldn't even be notified of if you tried to open the tool from the GUI desktop icon) which is why Vivado will crash (and leave you with a few unterminated zombie processes in the background):

From your Linux shell prompt

```
$ vivado &
application-specific initialization failed:  couldn't load file "librdi_commontasks.so"
libtinfo.so.5:  cannot open shared object file:  No such file or directory

[1]+ Stopped vivado
$
```

This can be fixed by manually installing the `libtinfo5` package:

From your Linux shell prompt

```
$ sudo apt-get install libtinfo5
$ vivado &
[1] 14585
$
****** Vivado v2022.1 (64-bit)
  **** SW Build 3526262 on Mon Apr 18 15:47:01 MDT 2022
  **** IP Build 3524634 on Mon Apr 18 20:55:01 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc.  All Rights Reserved.

start_gui
```

The GUI opens and displays the Vivado welcome layout (fig F.8).
If necessary refer to Xilinx answer record nb. 76585 for more information on this bug and how to solve it (`https://support.xilinx.com/s/article/76585?language=en\char‘_US`). You can also refer to this link: `xilinx.com/htmldocs/xilinx2020_2/vitis_doc/acceleration_installation.html#juk1557377661419` to collect more information on the Xilinx tools installation process.

Vivado project creation

13 **Project creation:** click on *Create Project* to open the project wizard manager. As project name, choose for instance `ecc` and for project location, set `ipecc/hw/ip`. Uncheck the *Create project subdirectory* box, so as to get every subsequent files created into local folder `ip/`. In *Project Type* section choose the default *RTL Project*. In *Add Sources* section choose *VHDL* for both *Target language* and *Simulation language*. Click *Add Files* and then navigate to the `hdl` folder of your local copy of the IPECC repository (if you've followed step **4** exactly, this should be ~/`IPECC/hdl`).

Below is a flatten list of the source files you should **integrate into the project** (see also fig-F.9). Vivado will automatically build the hierarchy of entity-architecture pairs so don't bother with the order in which you add the files from the wizard.

You'll have to proceed in 4 steps, each time clicking on button *Add Files* to add the files of one of the four directories below, and in the end click *Finish*.

Step 1 on 4: these are the files you must add from folder ${IPECC}/hdl/common:

- ecc.vhd
- ecc_axi.vhd
- ecc_curve.vhd
- ecc_customize.vhd
- ecc_fp.vhd
- ecc_fp_dram.vhd
- ecc_fp_dram_sh_fishy.vhd
- ecc_fp_dram_sh_fishy_nb.vhd
- ecc_fp_dram_sh_linear.vhd
- ecc_log.vhd
- ecc_pkg.vhd
- ecc_scalar.vhd
- ecc_shuffle_pkg.vhd
- ecc_software.vhd
- ecc_utils.vhd
- ecc_vars.vhd
- fifo.vhd
- mm_ndsp.vhd
- mm_ndsp_pkg.vhd
- (let aside file pseudo_trng.vhd)
- sync2ram_sdp.vhd
- syncram_sdp.vhd
- virt_to_phys_ram.vhd
- virt_to_phys_ram_async.vhd

Step 2 on 4: these are the files you must add from folder ${IPECC}/hdl/common/ecc_trng:

- ecc_trng.vhd
- ecc_trng_pkg.vhd
- ecc_trng_pp.vhd
- ecc_trng_srv.vhd
- es_trng.vhd
- es_trng_aggreg.vhd
- es_trng_bitctrl.vhd
- es_trng_sim.vhd

Step 3 on 4: these are the files you must add from folder ${IPECC}/hdl/common/ecc_curve_iram:

- ecc_addr.vhd
- ecc_curve_iram.vhd
- ecc_vars.vhd

Notes:

1. Ignore any ⟨.h⟩ file from folder ecc_curve_iram.
2. If you don't find the three VHDL files above that means you passed over step **5** above, so go back there to generate these files, and resume back here when you're done. Also note that these three files each have a link of the same name in folder hdl/common/ which is why you'll also see them there (Vivado will detect if these links are broken, i.e if their target file is missing because they were not built yet, and in that case will pertinently refuse to add them to the project).

Step 4 on 4: these are the files you must add from folder ${IPECC}/hdl/techno-specific/xilinxa/series7:

- es_trng_bit_series7.vhd
- large_shr_series7.vhd
- macc_series7.vhd
- maccx_series7.vhd

Notes:

3. Ignore for the moment source file ⟨pseudo_trng.vhd⟩ of folder hdl/common (meaning you don't need to import it in the project right now).
4. Obviously the last batch of files is taken from folder xilinxa/series7 because we're targeting a Zynq device (otherwise adapt this to the device you're targeting). Select xilinxa/ultrascale instead if you're targeting an UltraScale device.

Check the *Copy sources into project* box! This will make Vivado copy all the above files (including their containing directories) into a local folder named ecc.srcs/sources_1/imports/hdl, thus allowing you to locally edit the source files without creating any side effect with your original local copy of the git repository.

Project configuration: skip the *Add Constraints* section and switch to *Default Part*. It does not really make sense to specify a precise device for the project we're about to create as it's not intended for a specific hardware and aims instead at creating an exportable IP that we can later integrate into specific FPGA designs. However Vivado requires us to choose something here so **select the** xc7z010clg400-1 **device** which is the one cabled on the Arty Z7 board (fig-F.10). Review the *New Project Summary* (fig-F.11) then click *Finish*. Vivado opens the newly created project. After a few seconds, by unrolling *Design Sources* in the *Source* panel you should see (fig-F.12) the hierarchy Vivado created from a summary compilation of all the source files (mind the *Hierarchy* tab at the bottom of the *Sources* panel).

14 **Kludge #2** – Once Vivado has created the project, click on the *Settings* link at the top of the *Flow Navigator/Project Manager* column (left of the GUI) and in the *Settings* dialog box that opens, go to *Tool Settings* ▶ *Text Editor* ▶ *Syntax Checking*. In the *Syntax Checking* scrolling menu, choose *Vivado* instead of *Sigasi* (see fig-F.13).

This should prevent Vivado from hanging itself upon performing some actions like adding or removing source files to or from your project. *Sigasi* appears to be a third-party tool that is used to pre-compile on-the-fly the sources in your project to detect and highlight possible errors in your VHDL code before you actually ask for project compilation. We don't need this stuff here, all the more that it does more harm than good. Click *Apply* and OK. As warned by Vivado, you'll have to exit and then relaunch the tool for the project settings modification to be applied.

While you're at it, also disable that undesirable behaviour that Vivado has to run your simulation for 1 us each time it recompiles it without you asking for it: still in the *Settings* dialog box, go to *Project Settings* ▶ Simulation and in the *Simulation* tab, remove the default value of `1000 ns` of parameter `xsim.simulate.runtime*` by clicking on the small gray cross at the right of the value field (fig-F.14). Click OK. You'll probably have to close the project and reopen it to have the new settings taking effect.

15 **Edit source file** ⟨`ecc_customize.vhd`⟩ in folder `ecc.srcs/sources_1/imports/hdl/common`. This file contains every parameter you need to set in order to customize the IP for a specific hardware (see Appendix A for more information on IP customization). Fig. F.18 on p. 162 shows the content of this file as it should be updated now to match the hardware we are targeting for this tutorial.

The important parameters, the ones you should absolutely set same we do in order to observe the same behaviour and obtain something that actually works on the hardware, are explicitly pointed to on the figure by a violet hand pointing symbol. The other less important parameters (because they concern performances and/or security options) are pointed to by the same symbol but smaller and in black. Take care particularly of the highlighted lines because for these the default value of the parameter differs from what we want to set here. We're going to list these parameters and give a few explanations about them (note that the source file ⟨`ecc_customize.vhd`⟩ contains, after its packaged declaration, a long detailed description of each of its parameters, so you're naturally encouraged to refer to it).

To summarize the configuration, we'll use the IP:

- with the **possibility to dynamical set the level of cryptographic security** (`nn_dynamic = TRUE`), 256-bit being the maximum achievable level of security (`nn = 256`)
- for a Xilinx 7-series component (`techno = series7`)
- with **two instanciated Montgomery multipliers** (`nbmult = 2`) **each using 6 DSP blocks** (`nbdsp = 6`) and with their own clock-domain (`async = TRUE`), the frequency of which will be set while instanciating the IP in a larger design (input signal clock `clkmm`, see later on)
- with **debug mode disabled** (`debug = FALSE`), meaning any static configuration for side-channels we set in this file won't be disengageable at runtime by software (along with every other built-in countermeasure not listed here) – furthermore, all debug features will be pruned out from hardware during synthesis
- with **blinding countermeasure hardware-unlocked** (`blinding = 0`) but nonetheless engageable at runtime by software
- with **shuffling memory countermeasure enabled (`shuffle = TRUE`) and hardware-locked** (viz. non-disenga-geable at runtime), with shuffling here meaning (`shuffle_type = permute_lgnb`) a permutation of large numbers inside the IP internal memory inbetween each two consecutive bits of the scalar
- with **projective coordinates of sensitive points periodically randomized**, the period (`zremask = 4`) being «each four bits of the scalar».

> ☞ As a general rule, the non-debug (secure) mode enforces that no runtime change in the configuration can be made by the software that would decrease the level of security. On the other hand, increasing the security (by setting for instance a larger blinding scalar or a smaller value for Z-remask using the `W_ZREMASK` register) can be done even if the IP was synthesized in non-debug mode.

The last parameters concern simulation only. During behavorial simulation:

- the HDL model of the physical TRNG will be discarded (`notrng = TRUE`) and replaced with a simulation model reading random bytes from a deterministic file (`simtrngfile = /tmp/random.txt`) – as the physical true RNG uses a combinational loop, it can't be simulated.

- all arithmetical operations processed by the IP in the underlying finite field of the curve will be logged in a specific file (`simlogfile` = `/tmp/ecc.log`) along with the address of their input operands in the internal IP memory, their result and a timestamp
- input test-vectors to stimulate the IP with will be read by the simulation testbench from a specific file named `simvecfile` = `/tmp/ecc_vec_in.txt`.

Note:

1. Naturally parameter `notrng` must only be set to `TRUE` when running simulation. We'll later turn it back to `FALSE` when running synthesis (c.f step **26**)

Simulation of the IP

As mentioned in the previous step, simulating the IP requires to first generate two input files, ⟨`/tmp/random.txt`⟩ and ⟨`/tmp/ecc_vec_in.txt`⟩.

16 **Generation of the random file** ⟨`/tmp/random.txt`⟩ – The format expected for this file is very simple. This is an ASCII file containing the unsigned decimal value of one byte (0 to 255) per each line. As mentioned in file ⟨`${IPECC}/sim/HOWTO-random.txt`⟩ of the Git repository, here's how you can quickly generate e.g 16 millions of such formatted values using ⟨`/dev/urandom`⟩ as random source:

From your Linux shell prompt

```
$ od -t u1 -w1 -v /dev/urandom | awk '{print $2}' | head -$((16*1024*1024)) > /tmp/random.txt
(just takes a few seconds...)
$ head -5 /tmp/random.txt   # Obviously your values will differ
116
24
22
174
0
```

17 **Generation of the input test-vectors file** – We won't detail here the format of the input test-vectors file (it's specified in B.2) however it's quite straightforward. You can take a look at file ⟨`std-curves-test-vectors.txt`⟩ in folder `${IPECC}/sim` which gives an example of such a file. But this concerns standard ECC protocols which obviously involve numbers of cryptographic sizes (e.g $>$ 160 bits) for which behavorial simulation is quite long.

We're going to use Python-like scripting with the *SageMath* symbolic computation tool (also named *Sage* for short) and licensed under the GPL, to generate a few test-vectors on a «small» curve, let's say `nn` = `21` bits (why not). This will go much faster. Besides, the script used to generate the test-vectors for the simulation will also happen to be useful when testing the real hardware, because the small test application provided with the IP software driver consumes test-vectors in the exact same format as the RTL testbench does.

First **install the** *Sage* **tool:**

From your Linux shell prompt

```
$ sudo apt-get install sagemath
(takes half an hour...)
```

Once the install is done:

From your Linux shell prompt

```
$ cp -Rf ${IPECC}/sage/* ~/ipecc/sw/sage/.
$ cd ~/ipecc/sw/sage
```

Edit the file ⟨`generate-tests.sage`⟩. Search for the following parameters and set each one to the value that is given alongside:

```
nn_constant = 21      # meaning all generated curves will be of size nn = 21 bits
NBCURV = 1            # meaning only one curve will be generated
NBKP = 1              # meaning only one scalar-multiplication test generated per curve
NBADD = 1             # meaning only one point-addition test generated per curve
NBDBL = 1             # meaning only one point-doubling test generated per curve
NBNEG = 1             # meaning only one point-negate (−P) test generated per curve
NBCHK = 1             # meaning only one boolean "is P on curve?" test generated per curve
NBEQU = 1             # meaning only one boolean "are points equal?" test per curve
NBOPP = 1             # meaning only one boolean "are points opposite?" test per curve
NO_EXCEPTION = True   # meaning no exception tests generated
```

More informations will be given on usage of this file in step **55**. For now close the file after saving it, then run it as input to *Sage* while redirecting the standard output into the file we want to generate, which we said is ⟨/tmp/`ecc_vec_in.txt`⟩:

From your Linux shell prompt

```
$ sage generate-tests.sage >/tmp/ecc_vec_in.txt
Generating curves for nn = 21
```

☞ Alternatively if you wish to observe strictly the same things as in the tutorial you can use the same test-vector file as we do. You'll find this file in `${IPECC}/sim` (simply copy it in `/tmp/` or have parameter `simvecfile` point to `${IPECC}/sim/ecc_vec_in.txt` in ⟨`ecc_customize.vhd`⟩).

18 **Edit the file to get a glimpse at the format** (see fig-F.19 on p. 162). This file was generated at random, your numerical values should look different! You may think we could have used a predefined markup syntax like XML's, but writing an XML parser in VHDL is not a psychologically sane thing to do, and besides the format here is very simple. Only one curve numbered 0 is defined by its parameters `nn`, p, a, b and q, then seven tests are listed, each with the corresponding input point(s) coordinates, and, when it makes sense, values of the output point coordinates, or answer to a boolean test (true or false). The string `"== NEW CURVE"` is used as a tag to identify definition of a new curve, the string `"== TEST"` to identify the definition of a new test. All the tests directly following the definition of a curve naturally pertain to that curve until the definition of a new curve is met, and so on.

The numbers attributed to the curve and the tests, including the '`#`' character, have no importance as the testbench simply interprets these as a string id (so does the test app for the real hardware). Here our *Sage* script simply increments a number for each new curve generated, then has each test associated with this curve named after the number of the curve plus a dot plus a number incremented for each new test. That second number is not reset to 0 when a new curve is generated, thus it uniquely identifies the test in a campaign (while the first number allows to quickly identifies which curve a test is referring to).

The VHDL testbench that we will now run reads the input test-vector file one line at a time, gathering the informations that build up a specific test. For each test, it submits its parameters to the RTL model of the IP through the AXI interface (emulating transactions of a CPU or any other AXI initiator) and has the operation started. Then it reads back the result and compares it with what is expected according to the input test-vector file. Simulation is carried out as long as no mismatch occurs between the two results (the RTL one and the one from the input test-vector file) and of course as long as no end-of-file is met in either the input test-vector file or the input random file.

19 **Adding simulation sources to the project** – Click on *Add Sources* in the left-column of Vivado (*Project Manager*). Select *Add or create simulation sources* (the third line of the **Add Source** pop-up window, see fig-F.15) and click *Next*. Click *Add Files* and navigate to the `sim` folder of your local copy of the IPECC repository then select the four files below:

- `ecc_tb.vhd`
- `ecc_tb.wcfg`
- `ecc_tb_pkg.vhd`
- `ecc_tb_vec.vhd`

Be sure to check the *Copy sources into project* box (also keep the default check of option *Include all design sources for simulation*) (fig-F.16) and then click *Finish*.

After some time, Vivado updates the file hierarchy and you should see, in the *Simulation Sources* section of the *Sources* tab, that `ecc_tb` has been set as the top level entity for simulation (a bold font indicates that) and integrated ⟨`ecc_tb.wcfg`⟩ as the default waveform configuration file (fig-F.17).

20 **Running the simulation** – In the *Flow Navigator* column of Vivado GUI, click on *Simulation* ▶ *Run Simulation* and select *Run Behavorial Simulation*. This launches compilation and elaboration of all the source files. After a few seconds a waveform viewer is displayed according to the configuration set in the waveform file ⟨`ecc_tb.wcfg`⟩.

Under the TCL prompt at the bottom of Vivado (`Type a Tcl command here`) enter command "`run 800 us`". Alternatively you can also enter the value 800 us using the simulation time bar in the menu bar of Vivado (`800 us`) and click on the Play button. Also note that depending on whether the $[k]\mathcal{P}$ test generated in your own version of the input test-vector file is configured with blinding or not, you might have to run the simulation longer that 800 µs (because the blinding much increases the computation time of the scalar multiplication). If you want to be sure run the simulation for 1600 µs.

The simulation takes one or two minutes. The waveform is continuously updated as similation moves forward (see figure F.20) and informations are logged onto the Vivado simulation console (see figure F.21). The numbers displayed in violet on the two figures F.20 and F.21 correspond with each other (meaning they match the same steps of the simulation).

- The signals displayed in kakhi green on the waveform are the signals of the AXI interconnect, with the five channels Address-Write (AW), Write-Data (W), Write-Response (B), Address-Read (AR) and Read-Data (R).
- The signals in green are internal to the main control component `ecc_scalar` in the IP, they give overall information about the computation, such as the state of the main FSM (register `r.ctrl.state`), state of the program FSM (register `r.kp.substate`) and state of the CoZ FSM (register `r.kp.joye.state`).
- The signals in orange and red are internal to the processor `ecc_curve`. The signal in red is the Program Counter (register `r.decode.pc`).

Here's an overview of what is happening during the simulation:

- At instant 0, the top-level entity displays configuration information based on the content of source file ⟨`ecc_customize.vhd`⟩ (see mark ❶ on fig-F.21).
- The testbench's process reads the input test-vector file and found the definition of curve 0. It transmits its parameters to the IP just like a CPU would, emulating AXI transactions (see mark ⓐ on fig-F.20).
- Receiving number p and a triggers in the IP the computation of the two Montgomery constants (see §3.4.2): `ecc_scalar` enters **main FSM** state `cst` (see mark ❷ on figs. F.20 and F.21).
- The testbench's process continues reading the input test-vectors file and find description of test 0, which is a $[k]\mathcal{P}$ computation. It sends coordinates of point $\mathcal{P}$ to the IP as well as the scalar k (see mark ⓑ on fig-F.20) and gives computation a go.
- Main FSM in component `ecc_scalar` enters the state `kp` and has the scalar multiplication carried out by the lower components of the IP: `ecc_curve` fetches and decodes instructions from the microcode memory, `ecc_fp` is basically the ALU actually executing the arithmetical operations on point coordinates in the underlying field $\mathbb{F}_p$, with the Montgomery multiplications being handed over asynchronously to components `mm_ndsp`. The scalar multiplication is the most expensive computation the IP can be programmed to do, here it lasts approx. 50 µs, between instants 100 µs and 600 µs, that is between the vertical black and blue cursors (see mark ❸ on figs. F.20 and F.21). Note that you can read on the waveform, on the registered signal `r.dbg.joyebit`, the index of the scalar bit which is currently being processed. During the most part of $[k]\mathcal{P}$ computation, the **programs FSM** of `ecc_scalar` stays in state `joyecoz`, while the **CoZ FSM** runs cyclically through all its different states for each bit read from the scalar.
- These different states can be seen on the lower half of figure F.20 which offers a detailed view of the processing of one bit of the scalar, the bit of weight 10. From left to right on that part of the figure:
  - While in the `permutation` state, the IP memory of large numbers is shuffled, meaning every one of the 32 large numbers buffered into it is moved at a new random physical address inside the memory, and the indirection memory logic is updated to allow a transparent access to it.
  - While in state `zrmsk`, a new Z coordinates for the two sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$ is withdrawn and their X and Y coordinates are updated according to that new Z coordinate.

- While in `itoh` state, the two versions of the scalar which reside in memory and which are masked according to the ADPA countermeasure are right-shifted and their respective least significant bits κ_i and κ'_i are sampled and transmitted to `ecc_curve` logic.
- While in `prezaddu` state, the differences between the X and Y coordinates of the two sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$ are computed ($X_{\mathcal{R}_0} - X_{\mathcal{R}_1}$ and $Y_{\mathcal{R}_0} - Y_{\mathcal{R}_1}$). This is a preamble to the computation of the ZADDU formulæ actually performed in the following `zaddu` state, which in any way require that these differences are computed, and which allow to detect a possible exception, (like the points $\mathcal{R}_0$ and $\mathcal{R}_1$ being equal or opposite).
- While in `zaddu` state, the ZADDU formulæ are implemented (see §3.3) which compute the addition of the two points $\mathcal{R}_0+\mathcal{R}_1$ along with the Z-update of one of them with the resulting addition point's Z coordinate. The addition and the updated points clobber the previous stale values of $\mathcal{R}_0$ and $\mathcal{R}_1$, with which point among the new $\mathcal{R}_0$ or $\mathcal{R}_1$ is receiving the addition or the update being submitted to the masked value κ'_i of the scalar. The two final points share the same Z coordinate wich is now different from the one the points were sharing when entering the ZADDU computation.
- The `prezaddC` state is quite similar to the `prezaddu` state and allows for detection of exceptions cases before entering the `zaddc` state.
- While in `zaddc` state, the ZADDC formulæ are implemented (see §3.3) which compute the addition $\mathcal{R}_0 + \mathcal{R}_1$ and the subtraction (either $\mathcal{R}_0 - \mathcal{R}_1$ or $\mathcal{R}_1 - \mathcal{R}_0$) of the two points $\mathcal{R}_0$ and $\mathcal{R}_1$ with the two resulting points sharing a new Z coordinate. The addition and the subtraction points clobber the previous stale values of $\mathcal{R}_0$ and $\mathcal{R}_1$, with which point among the new $\mathcal{R}_0$ or $\mathcal{R}_1$ is receiving the addition or the subtraction being submitted to the masked value κ_i of the scalar.

- When $[k]\mathcal{P}$ computation is over (vertical blue cursor on figure F.20) the busy bit in the `R_STATUS` register of `ecc_axi` is released and the testbench's process which is acting as a polling software can detect that the job's done. The result (coordinates of the point $[k]\mathcal{P}$) is read back on the AXI interface (see mark C on fig-F.20).
- The testbench's process resumes parsing of the input test-vector file and succesively finds the description of tests 1 to 6, which are successively submitted to the IP in the same manner as previous $[k]\mathcal{P}$ computation (see marks 4 to 9 on figs. F.20 and F.21). Notice how these operations take a much shorter time than the scalar multiplication.
- Note that a lot of activity seems to be present on the Address-Read and Data-Read channels of the AXI interface, but it's just that the testbench uses polling (instead of asynchronous interrupts) to wait continously for end of processing whenever it has submitted a job to the IP.
- Finally the testbench's process hits end of file in the input test-vector files and ends the simulation after displaying some info on tests statistics (see mark 10 on fig-F.21).

From mark 1 on fig-F.21 you can see that the IP has automatically set the value of parameter `ww` to 16, which is consistent with the DSP blocks of 7-series family of FPGAs which are built on 25 × 18 signed hardware multipliers (the two most significant bits of 18-bit words are always set to 0 as the IP uses DSP blocks only to multiply positive numbers). Actually a value of 17 for `ww` would be the optimal value to use the DSP blocks with, however one bit wouldn't make a real difference and it would be at the price of less readability during simulation and debug. Alternatively you can switch value of 16 for value 17 by editing function `set_ww` in package ⟨`ecc_utils.vhd`⟩.

From parsing of the `R_STATUS` register an error was detected at the end of the $[k]\mathcal{P}$ computation (see orange line on fig-F.21). The error is `STATUS_ERR_UNKNOWN_REG` meaning that an unknown address was accessed by the software (here the testbench's process) either in write or read mode. This is because the testbench performs a few actions targeting debug-only registers at the begining of the simulation, but remember that in step 15 above we've configured the IP in non-debug (secure) mode in file ⟨`ecc_customize.vhd`⟩. Hence debug registers are non instanciated in the design and the IP will procude an error each time we try to access any of them. This does no harm here, the testbench simply acknowledges the error and things can continue normally.

Also note that Vivado translates every action run from the GUI into an equivalent Tcl command (appearing as blue lines in the Tcl console log) using a rich set of options, that you can later use and gather to build scripts, thus saving project setup time.

21 **Getting more details on arithmetical operations** – As you can see on the simulation console log of fig F.21, each component in the simulation hierarchy from top to bottom (from the top testbench instance `ecc_tb` down to component `ecc_fp`, including components `ecc_axi`, `ecc_scalar` and `ecc_curve`) displays a different kind of information on the console during the simulation. However these are quite high-level scheduling informations, and only the values of initial and final point coordinates are displayed through the console. If you need to

investigate more the internals of the computations and track exactly the numerical values of the intermediate terms involved in the different computations on the curve, you can do so by reading the contents of a specific file whose path is given by the value of parameter `simlogfile` in ⟨`ecc_customize.vhd`⟩, which by default points to ⟨`/tmp/ecc.log`⟩. This trace file provides fine-grain informations. In particular all the software routines executed by `ecc_curve` and the result of all arithmetical instructions processed by components `ecc_fp` and `mm_ndsp` are logged to it.

So open file ⟨`/tmp/ecc.log`⟩ and search for multiple instances of the regular expression `"[XY]R[01] ="`. You can see that the x and y coordinates of Co-Z points $\mathcal{R}_0$ and $\mathcal{R}_1$ (designated as `XR0`, `YR0`, `XR1` and `YR1`) are logged, along with the value of the z coordinate (designated as `ZR01`) after each major step of the $[k]\mathcal{P}$ computation, in particular after each iteration of the loop parsing bits of the scalar. Furthermore, some lines of the log file start with the character string `"[VHD-CMP-SAGE]"`. This is to allow you to compare, using simple grepping, the intermediate coordinates of sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$ obtained from the RTL simulation with the ones obtained from the equivalent *Sage* scripting – the latter ones thus providing **finite-precision proof** of the former ones. We'll look into that later on (c.f step **23** of this tutorial).

22 **Deep inspection and validation of intermediate arithmetical terms** – Open a pseudoterminal (Ubuntu default shortcut: `Ctrl-Alt-T`) and change dir to ∼/`ipecc/sw/sage` folder. We're going to use the *Sage* script ⟨`kp.sage`⟩ located in this folder to run an (almost) exact sequence of operations as the RTL. We say «almost» because there are actually two features in the hardware that we cannot emulate yet in software, because they consume a continuous stream of random values in the hardware that it's very difficult to emulate from outside the IP[6]. These are the so-called «XY-shuffling» countermeasure and the so-called «Z-remask» countermeasure:

- The **«XY-shuffling»** countermeasure consists in changing periodically the physical address in the IP memory of large numbers (`ecc_fp_dram`) at which are found the four coordinates $x_{\mathcal{R}_0}, y_{\mathcal{R}_0}, x_{\mathcal{R}_1}$ and $y_{\mathcal{R}_1}$ of $\mathcal{R}_0$ and $\mathcal{R}_1$ sensitive points (or `XR0`, `YR0`, `XR1` and `YR1` according to the name they're given in the assembly source files that build up the microcode memory – all these files are located in folder `hdl/common/ecc_curve_iram/asm_src/*.s`). Here the «periodically» means in between each consecutive iterations of the ZADDU or ZADDC operations. Hence the shuffling happens twice for each bit of the scalar that is being processed. The XY-shuffling is implemented in component `ecc_curve`. This component performs the random draw of forthcoming address for each of the four sensitive coordinates. It also patches dynamically (on-the-fly, during their decoding phase) the address of the operands extracted from the opcodes of the microcode when these operands happen to be one of the sensitive coordinates. This **patching** mechanism relies on a collaborative work between the static writing of the assembly code for ZADDU and ZADDC routines and the definition of the operands' patching of opcodes applied dynamically by `ecc_curve`. If you open assembly files ⟨`zaddu.s`⟩ and ⟨`zaddc.s`⟩ (folder `asm_src`) you may notice that some instructions are tagged with the mark `",p"` followed by a decimal number. These are the «patched opcodes», viz. the ones whose operands are subject to modification by `ecc_curve` before being transmitted to the $\mathbb{F}_p$ ALU (`ecc_fp`) for their actual execution. There is a total of 64 patches implemented by `ecc_curve`, among which 16 are used in ZADDU and 12 in ZADDC. The remaining 32 are used in other parts of the code to serve implementing other kinds of countermeasures.
- The **«Z-remask»** countermeasure consists in multiplicatively refreshing the Z coordinate of the projective representation of sensitive points $\mathcal{R}_0$ and $\mathcal{R}_1$ regularly (and obviously also updating their X and Y coordinates accordingly). As we have seen in step **15** above, «regularly» here means every four bits of the scalar.

Note:

1. Don't confuse the countermeasure called «XY-shuffling » with the one simply called shuffling. The latter consists in randomizing the whole content of the IP memory of large numbers, not only the coordinates of $\mathcal{R}_0$ and $\mathcal{R}_1$. The two countermeasures are independent and complementary, thus illustrating the concept of *defense-in-depth*. Shuffling is the countermeasure to which correspond the parameters `shuffle` and `shuffle_type` in file ⟨`ecc_customize.vhd`⟩ (see e.g fig-F.18 on p. 162 and previous step **15** of the tutorial). The «XY-shuffling» on the other hand does not have associated config parameters in ⟨`ecc_customize.vhd`⟩ because it's not consuming random data as much as shuffling, hence its presence was not left as an option to the hardware designer. In particular, in non-debug (secure) mode, the countermeasure «XY-shuffling» can't never be disabled. The only way to do that is if the IP is configured in debug mode. We'll do this in step **25** of the tutorial.

Since the two previously described countermeasures cannot be emulated in software, we're going to disable them. But first we're going to observe their real effects by comparing the informations logged by `ecc_curve`

[6] This might be implemented in future releases.

in ⟨/tmp/ecc.log⟩ during the RTL simulation and the ones obtained from the IP emulation scripts located in folder ipecc/sw/sage. Change dir to that folder and duplicate the file ⟨kp.py⟩ in order to make a customized version of it, for instance copy it into ⟨kp21.py⟩:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/sage
$ cp  kp.py  kp21.py
```

Edit new file ⟨kp21.py⟩ and fill in all the informations required to define the elliptic curve and the base point according to what is set in the test-vector file (please refer to fig-F.22 on p. 165). You can notice that some parameters in the Python script remain blank (without an assigned value). For ww parameter, simple set 16, as for all Xilinx devices. The remaining variables are random data whose value we're going to extract from the RTL simulation log file ⟨/tmp/ecc.log⟩. Simply grep regular expression "\.random_.*L" on ⟨/tmp/ecc.log⟩ like illustrated below (using swith -A 3 to display a few lines after each match):

From your Linux shell prompt

```
$ grep -A 3 "\.random_.*L" /tmp/ecc.log
.random_alphaL [0x031]
[0x031]    NNRND 0xbbd431af  (12 <-  random  )  [96705000 ps]
[0x032]    NNADD 0x001ce256  (08 <- 03  +  31)  [96875000 ps]
[0x033]    NNADD 0x00000000  (09 <- 31  +  31)  [97045000 ps]
--
.random_muL [0x041]
[0x041]    NNRND 0x7dd0807a  (26 <-  random  )  [117645000 ps]
[0x042]    TESTPAR           (26 is   even   )  [117755000 ps]
[0x043]    NNRND 0x1b570add  (27 <-  random  )  [117885000 ps]
--
.random_phiL [0x047]
[0x047]    NNRND 0x7f4e8d2f  (10 <-  random  )  [118585000 ps]
[0x048]    NNRND 0xc75870d7  (11 <-  random  )  [118715000 ps]
[0x049]    TESTPAR           (04 is    odd   )  [118825000 ps]
--
.random_lambdaL [0x071]
[0x071]    NNRND 0x00110c4a  (21 <-  random  )  [141345000 ps]
[0x072]    NNSUB 0xfff426ff  (22 <- 21  -  00)  [141515000 ps]
[0x073]    NNADD 0x00110c4a  (21 <- 22  +  00)  [141685000 ps]
```

You can see on the above excerpt of our command line interface that the correspondence with the parameters that remain to be set in the *Sage* script is quite clear: see fig-F.23 on p.165. Let's give these values explicitly:

- The instruction NNRND which was fetched from address 0x031 is the one generating the random used to blind the scalar, hence we set in the Python script: alpha0 = 0xbbd431af.
- The instruction NNRND which was fetched from address 0x041 (resp. 0x043) is the one generating the random mu0 (resp. mu1) hence we set in the Python script: mu0=0x7dd0807a and mu1=0x1b570add.
- The instruction NNRND which was fetched from address 0x047 (resp. 0x049) is the one generating the random phi0 (resp. phi1) hence we set in the Python script: phi0=0x7f4e8d2f and phi1=0xc75870d7. These two random values are used for the ADPA countermeasure.
- The instruction NNRND which was fetched from address 0x071 is the one generating the random used to set the initial Z-coordinate of points $\mathcal{R}_0$ and $\mathcal{R}_1$, hence we set in the Python script: lambd=0x00110c4a.

Note:

1. If you followed previous step **5** and built the pdf version of the complete microcode listing, you can observe in it the correspondence with the opcodes (along with their address in the microcode) as they were logged during the RTL execution (fig-F.23 on p.165). Simply open file ⟨ecc_curve_iram.pdf⟩ from folder ${IPECC}/hdl/common/ecc_curve_iram/latex and search for the string ".random" in it.

23 Now that we have filled in all the values in the Python script ⟨kp21.py⟩, we can execute this file. However before doing that we need to build another file: indeed if you take a look at the file header in ⟨kp21.py⟩,

you can see that the first non comment line is an import line saying: `"from kpsage import main"`. Hence we need to produce a Python module file named ⟨`kpsage.py`⟩ before running ⟨`kp21.py`⟩. For this we use the `--preparse` switch of the `sage` command on file ⟨`kp.sage`⟩, thus producing a file named ⟨`kp.sage.py`⟩ that we immediately rename into ⟨`kpsage.py`⟩ to make it a filename compatible with the naming conventions of Python modules:

From your Linux shell prompt

```
$ sage --preparse kp.sage
$ mv kp.sage.py kpsage.py
```

Now run the script ⟨`kp21.py`⟩ through *Python3* - doing so «pipe-grep» your command to filter its output with string `"VHD-CMP-SAGE"` and also redirecting the output into a file named e.g ⟨`/tmp/sage21.log`⟩:

From your Linux shell prompt

```
$ python3 kp21.py | grep VHD-CMP-SAGE >/tmp/sage21.log
```

Similarly grep the RTL simulation log file with `"VHD-CMP-SAGE"` and redirect the ouput into another file e.g ⟨`/tmp/simu21.log`⟩:

From your Linux shell prompt

```
$ grep VHD-CMP-SAGE /tmp/ecc.log >/tmp/simu21.log
```

24 Now observe the difference between the two files ⟨`/tmp/simu21.log`⟩ and ⟨`/tmp/sage21.log`⟩ using for instance `vimdiff`:

From your Linux shell prompt

```
$ vimdiff /tmp/simu21.log /tmp/sage21.log
```

What you should get (approximately of course, since the random values you have won't be the same) can be seen on fig-F.24 (p.166).

Two things can be noticed here. First, the four coordinates `[XY]R[01]` of points $\mathcal{R}_0$ and $\mathcal{R}_1$ match in both log files from the start, until computation reaches the scalar bit of weight 6. Remember indeed that in step **15** of the tutorial we programmed `zremask` = `4` in `ecc_customize.vhd`. Now since the IP starts parsing the scalar from its third least significant bit (viz. the bit of weight 2)[7] the first time where the coordinates of $\mathcal{R}_0$ and $\mathcal{R}_1$ are re-randomized happens to be the bit of weight 6.

Second, except the first step of computation which corresponds to the first six lines of the files, the addresses of the four coordinates `[XY]R[01]` always differ from one file to the other. On the right (*Sage* emulation) they keep their initial static values (`@XR0` = `4`, `@YR0` = `5`, `@XR1` = `6` and `@YR1` = `7`) while on the left (RTL simulation) they have been randomized. This is the effect of the «XY-shuffling» countermeasure.

By scrolling down to the bottom of the file you can nonetheless validate that results $[k]\mathcal{P}$ eventually match on both sides (see fig-F.25, p.166).

25 We're now going to disable the «XY-shuffling» and «Z-remask» countermeasures, one after the other, so as to observe complete match between the hardware and its emulating *Sage* script.
For the «Z-remask» countermeasure, we could simply edit file ⟨`ecc_customize.vhd`⟩ and set `zremask` = `0` in it. For the «XY-shuffling» however it's not that simple because as already mentioned before (see step **22** above) the countermeasure can only be disengaged if this IP has been instanciated in debug (unsecure) mode. That's why we will start by switching the IP into debug mode. Open file ⟨`ecc_customize.vhd`⟩ to change the value of parameter `debug` to `TRUE` as shown on fig-F.26 p.167.

[7] The reason why the IP starts parsing the scalar only at its bit of weight 2 when computing $[k]\mathcal{P}$ is explained in §4.1, see in particular algorithm 12 and the discussion pertaining to it in p.27.

Save the file and quit. In turn edit file ⟨`ecc_tb.vhd`⟩ from folder `${IPECC}/sim` (this is the source file of the RTL testbench), search for character string `"End of IP initialization & config"` and add a call to procedure `configure_zremasking()` just before that comment, as illustrated on fig-F.27 p.167.

Save and quit, then relaunch the simulation (Vivado shortcut: `Alt-R-U`) and when compilation is done, press `Shift-F2` to run the simulation for the amount of time that is currently being displayed in the simulation time-bar (e.g ).

After the one minute or two that it takes to run the simulation, redo the grep on the RTL simu log (as in step **23**):

From your Linux shell prompt

```
$ grep VHD-CMP-SAGE /tmp/ecc.log >/tmp/simu21.log
```

and again observe the difference between the two files ⟨`/tmp/simu21.log`⟩ and ⟨`/tmp/sage21.log`⟩:

From your Linux shell prompt

```
$ vimdiff /tmp/simu21.log /tmp/sage21.log
```

What you should see is now illustrated on fig-F.29 and F.30 (p.168). You can notice that the effect of the Z-remask has disappeared as coordinates match all along the computation in both log files.

Let's go on and again edit ⟨`ecc_tb.vhd`⟩, this time adding a call to procedure `debug_disable_xyshuf()` as illustrated on fig-F.28 p.167.

☞ The two functions we added a call to in ⟨`ecc_tb.vhd`⟩ are defined in the VHDL simulation package ⟨`ecc_tb_pkg.vhd`⟩. If you open this file and skim through its content you will see that it contains many helper functions that you can call when simulating the IP. Each one of this helper functions emulates an action on the IP through the AXI interface just as a software driver would do.

Save and quit, and once again perform an `Alt-R-U` + `Shift-F2` sequence in Vivado, to recompile the project and rerun the simulation. Repeat the grep (wait of course until the simulation is completed):

From your Linux shell prompt

```
$ grep VHD-CMP-SAGE /tmp/ecc.log >/tmp/simu21.log
```

Now the two log files should be strictly identical, as attested by the `cmp` command (whose absence of output means that no difference was found between its two input files):

From your Linux shell prompt

```
$ cmp /tmp/simu21.log /tmp/sage21.log   # cmp stays mute when files are identical
$ echo $?
0
```

You can also refer to fig-F.31 and F.32 (p.169) to confirm that files are identical to the byte.

☞ As a conclusion to this part of the tutorial, you can validate the behaviour of the VHDL model of the IP at every main step of the $[k]\mathcal{P}$ computation, as long as the two countermeasures «XY-shuffling» and «Z-remask» are disabled.

☞ The support for these two countermeasures may be added to the *Sage* emulation script in a future release.

Close the simulation (main menu *File* ▶ *Close Simulation*).

Synthesis of the IP

Synthesizing the IP alone doesn't really make sense so we're going to instanciate it inside a SoC infrastructure that'll include a CPU, an AXI interconnect, a DRAM controller, etc. so that driving the IP through a simple piece of software will be a straightforward operation.

26 **Packaging the IP** – Packaging the IP means to wrap its sources, constraints and other attached metadata and files into a Vivado formatted bundle that allows you to later instanciate the IP like a prefabricated brick into a more complex system (to show you how will be the purpose of steps **27** sq.).

We will package the IP in folder `ipecc/hw/ip/packaged` and later create and edit the SoC design instanciating it in folder `ipecc/hw/soc`.

From your Linux shell prompt

```
$ mkdir ~/ipecc/hw/ip/packaged
```

Edit ⟨`ecc_customize.vhd`⟩ and set parameter `notrng` to `FALSE` (see fig-F.33). This will ensure that the component instanciated in the TRNG part of the IP is actually the ES–TRNG design (see fig-F.34) and not the trivial simulation model.

Keep the IP in debug mode. This way you may later explore the different features of the IP when you have it run on a real device, without being constantly rejected by the hardware API because it thinks that you're attempting illegitimate actions. *In particular only the debug mode makes it possible to read back from the software the random bits generated by ES–TRNG and hence assess and validate the quality of its randomness.* Once the placement and routing of the TRNG provides the quality we expect from it, we can hardware-lock the design to ensure it will behave similarly in all subsequent releases of the design.

Now in Vivado nagivate to *Tools* ▶ *Create and Package New IP*. In the opening dialog box, click *Next* and keep default choice (*Package your current project ...*). For the IP location, navigate and select folder `ipecc/hw/ip/packaged`. Agree to confirm and then click *Finish*.

Vivado spawns a new window containing a temporary project named `tmp_edit_project`. Review the different steps in the column *Packaging Steps* inside the *Package IP – ecc* tab. The steps are listed with a ✔ symbol on their side.

The important thing here is to **remove the file ⟨`es_trng_sim.vhd`⟩ from the exported sources for the IP packaging.** Indeed even if the IP does not include the component this file describes (thanks to the setting `notrng = FALSE` we've made in ⟨`ecc_customize.vhd`⟩, see fig-F.34) Vivado will attempt to synthesize it and naturally find it can't (it contains file I/O) which will make the IP synthesis fail.

Click on entry ✔ *File Groups* and unroll the *Standard* ▶ *Synthesis* level of the file hierarchy. In the sources list, locate the file ⟨`src/es_trng_sim.vhd`⟩, right-click on it and select *Remove file* from the contextual menu (see fig-F.35). The file is taken out of the list.

The last step *Review and Package* presents you with a *Package IP* button, click on it (ignore the blue warning saying that the IP has been modified). Agree to closing the temporary project and, once back to the main Vivado window, close project `ecc` using *File* ▶ *Close Project*. This should take you back to the welcome layout of Vivado.

27 **Integrating the IP in the SoC**

Follow same steps as in step **13** (p. 121) to create a new project through the project wizard manager. Use the following settings:

- *Project name*: `az7-ecc-axi`
- *Project location*: ~/`ipecc/hw/soc` (do not create project subdirectory)
- *Project Type*: *RTL project*, check the *Do not specify sources at this time* box
- *Default Part:* select **Boards** rather than Parts then click on the *Refresh* button at the left bottom of the window (please refer to fig-F.36). This will have Vivado fetch the latest metadata files on the official supported boards. Note that you'll need to be connected to the Internet for this to work. After a few minutes, the window should be updated with many more boards than the default content. In the Vendor menu, now select `digilentinc.com`. Locate the Arty Z7-10 entry and click on button ⬇ to install the Arty Z7-10 board we wish to target. Now **make sure to select the board by explicitly**

clicking on the Arty Z7-10 line (this line must be highlighted in blue, as in fig-F.37). Note that the board's name is a hyperlink to the official web page of the board. Alternatively extensive information can be found here: `https://digilent.com/shop/arty-z7-zynq-7000-soc-development-board/`). Click *Next*, review the choices you made (check that part `xc7z010clg400-1` is actually set!) and then click *Finish*.

We're now going to instanciate a complete SoC including both PS and PL parts of the FPGA. We will do this using the graphical editor of Vivado which greatly eases the instanciation, edition and building of complex systems by automating the process of their interconnection. In particular, wiring together pairs of AXI master and slave components is quite time saving. Schematic designs in Vivado are called *Block Designs*.

In the *IP INTEGRATOR* top section of Vivado left column, click on *Create Block Design*. Accept the default settings (`design_1`, *Local to project*) and click OK.

In the *Diagram* tab that just opened, click on the [+] button which is at the center of the plain white zone and also in the diagram toolbar (

).

The IP catalog opens, displaying the list of all available IPs. Scroll down to the bottom of the window and double-click on *ZYNQ7 Processing System*. This will instanciate in the block-design a Zynq-7 ARM dual Cortex-9 processor (c.f fig-F.38) which corresponds to the so-called PS part (Processing System) of the device. The other part is the PL (programmable logic) and corresponds to the actual logic & routing fabric that matches more the idea of what an FPGA is. Obviously the PL is where our instance of IPECC will be located after synthesis. Click on the blue *Run Block Automation* proposal that appeared in a green banner. In the pop-up window that opens keep everything as set by default and click OK. After a few seconds Vivado updates the Zynq IP with connexions named `DDR` and `FIXED_IO` (whatever that means).

Now double-click on the Zynq blue box. This will open the *Re-customize IP* window which contains a rich set of features describing the configuration of the double ARM Cortex-A9 SoC, including the many peripheral controllers and embedded memories.

If you navigate a bit through the different tabs and entries of the window you will see that many features have been preselected by Vivado dues its internal knowledge of the board's configuration and schematics. In particular the *Peripheral I/O Pins* tab shows that the different on-board hardware controllers (Ethernet, UART, USB, etc) have their I/Os now hardwired to specific I/O pins of the device, with specific voltages for each.

Troubleshooting. If the preselection didn't work in your case, it means that the ArtyZ7-10 board was not properly added to your Vivado installation. In this case you have two choices:

(a) You can try to configure manually the Zynq processor by editing by hand each parameter one at a time, using the informations you'll find on the Arty board on the web. Search for the hardware configuration of the Arty Z7 10 and you can't but manage to find the proper informations. See for instance the online Reference Manual of the board:
`https://digilent.com/reference/programmable-logic/arty-z7/reference-manual`
(section *Zynq APSoC Architecture* in particular for the MIO pinning).

(b) The second solution is to search on the Internet for a way to manually install the Vivado preset board files for the Arty Z7 10. These files are provided by Digilent and are now available through their GitHub. Try this weblink:
`https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-sdk` (section *Installing Digilent Board Files*).
Also try the Digilent associated repository: `https://github.com/Digilent/vivado-boards`

Two important things remain to be configured here for our design to work that Vivado cannot be aware of:

1. We must enable the support for **interrupt requests** on the Zynq processor. Select the *Interrupt* entry at the bottom of window's left column and check the *Fabric Interrupts* box. Unroll the *PL-PS Interrupt Ports* and check the `IRQ_F2P[15:0]` box. See fig-F.39.

2. If you remember from step **15** we configured the IP to be in the *asynchronous* mode (`async = TRUE` in `ecc_customize.vhd`) meaning the Montgomery multipliers have their **own clock domain** that can run at a higher frequency than the rest of the IP. The clock used by the remaining logic being identical to the one of the IP's AXI interface, it seems natural to use the `s_axi_aclk` of the AXI interconnect for it. In the Zynq system, this clock is named `FCLK_CLK0` and it's selected by default in the configuration window

(see *Clock Configuration* entry in the window's left column, you need to unroll *PL Fabric Clocks*). The default frequency is 100 MHz and we'll keep that setting.

Now the two clocks don't need to be related at all. However we will have the Zynq processor also providing the second one, as this is simply done by checking the `FCLK_CLK1` box here. For the frequency, enter value 200 MHz. For the source PLL, keep the default `IO-PLL`[8].

Click OK and ignore the warning about the negative DQS delay on the DDR pins (fig-F.41). The Zynq processor in the block design has been upgraded with a few features (see fig-F.42), in particular:

1. There is an AXI initiator port (named `M_AXI_GP0`)
2. There is an interrupt signal (named `IRQ_F2P`)
3. There are two clock outputs (`FCLK_CLK0` and `FCLK_CLK1`)

28 Now **let's bring IPECC into the system.** Click on the *Settings* link at the top of the *Flow Navigator/Project Manager* column (left of the Vivado GUI) and in the *Settings* dialog box that opens, unroll *IP* ▶ *Repository* and click on the `+` button. Navigate to the folder where we previously packaged the IP (c.f step **26**) i.e `ipecc/hw/ip/packaged` and click *Select*. Vivado informs you with a pop-up that the folder `packaged` was added to the IP repository and that one IP was detected inside (see fig-F.43). Click OK, then back in the *Settings* window click *Apply* and OK.

Click again on the `+` button of the *Diagram* toolbar we used before to add the Zynq PS system to the block design. The IPs are listed alphabetically so scroll down in the opened window until you find the IP named `ecc_v1_0` (fig-F.44). Alternatively type `"ecc"` in the *Search* bar and select `ecc_v1_0`. Double-click on the IP name, which creates an instance of IPECC in the Block Design. In the green frame, click on the *Run Connection Automation* blue link. Ignore the content of the *Run Connection Automation* dialog box (keep anything as is) and click OK. The Block Design is now enhanced with two new blocks implementing the AXI interconnect plus clocks and reset as well as signals and buses interconnecting the different parts together. Right-click somewhere in the white empty zone of the Block Design and choose *Regenerate Layout*. The Block Design view is reorganized with more visibility, see fig-F.45.

You can see that the `FCLK_CLK0` clock output of the Zynq PS block is now connected to the `s_axi_aclk` input clock of the IP. Now let's connect `FCLK_CLK1`: hover your mouse over `FCLK_CLK1` clock output of the Zynq PS block so as to get a pencil for pointer. Click and hold the mouse button down to pull a signal line from the port, and reach `clkmm` clock input port of instance `ecc_0` of our IP. Release the mouse button, thus establishing the new connection. Repeat the operation to connect `IRQ_F2P[0:0]` input port of the Zynq PS block to the `irq` output port of component `ecc_0`.

Right-click on port `busy` of the IP and in the drop-down menu select *Make External*. An output port is created and given the name `busy_0`. We will later assign this port to a package pin of the device driving an on-board LED, thus allowing us to visually monitor the activity of the IP. Keep the remaining ports (`irqo`, `dbgtrigger` and `dbghalted`) of the IP unconnected. These are outputs so it doesn't matter if they stay unconnected in the design, the synthesizer will just trim them out. The Block Design should now look like fig-F.46 (don't bother with the fact that connections are in orange on the figure, we just select them before making the screen capture to make them quickly identifiable).

Click the right button anywhere in the blank zone of the block design and choose *Validate Design*. This allows to quickly test if everything is consistent in the block design before actually running a synthesis step on it. Vivado should turn out not to be satisfied and display a critical warning (see fig-F.47). Indeed there are two remaining input ports of IPECC instance that need to be connected, the 8 bit data bus `dbgptdata` and its strobe signal `dbgptvalid`. As opposed to output ports, it's a sound requirement that input ports be rigorously connected.

For now we're not going to use the pseudo-TRNG feature of the IP, so we'll simply connect these two ports to the ground. For this, you need to add to the block design two instances of the Xilinx IP named *Constant*. Close the critical warning pop-up by clicking OK then click again on the `+` button to open up the IP catalog and search for the name *constant*. Double-click to add it to the block design. The instance will likely be given the name `xlconstant_0` by Vivado. Repeat the operation to add a second *Constant* to the design, which this time will be called `xlconstant_1`. Double-click on instance `xlconstant_0` and set its configuration parameters to 0 for `Const Val` and to 1 for `Const Width` then validate with OK. Proceed likewise to

[8] Alternatively, if you want to try higher frequencies for the Montgomery multipliers' clock, you may have to select a different PLL, because they don't all have the same capabilities. Anyway since the two clocks of the IP can be totally asynchronous to each other, any source for that clock will do.

configure instance `xlconstant_1` with configuration parameters set to 0 for `Const Val` and this time to **8** for `Const Width` (this is an 8-bit bus). Validate with OK. Back to the block design, connect output `dout[0:0]` of block `xlconstant_0` to input `dbgptvalid` of IPECC and output `dout[7:0]` of `xlconstant_1` to input bus `dbgptdata`, just as you previously did to establish connections between blocks.
Give the block design a little step of *Regenerate Layout* and you should now see something like in fig-F.48. Retry *Validate Design* and this time Vivado should be satistied.

29 **Synthesis and place-and-route** – In the *Block Design* view, click the *Sources* tab and select the *Hierarchy* view. Under *Design Sources* right-click `design_1 (design_1.bd)` and select *Create HDL Wrapper*. The *Create HDL Wrapper* dialog box opens. Select *Let Vivado manage wrapper and auto-update* and click OK. The operation takes a few seconds and creates a VHDL wrapper for the complete PS + PL Block Design which embeds all the structure and features of what was edited graphically. After a while this wrapper is elected as the top-level of the hierarchy, as indicated by its name being displayed in bold font (fig-F.49). Still in the *Sources* tab expand `design_1_wrapper`. Right-click the top-level block diagram which is named `design_1_i:design_1 (design_1.bd)` and select *Generate Output Products*. Keep default settings in the dialog box that opens and click *Generate*. The operation takes one or two minutes and then you're being informed that *"Out-of-context modules were launched for generating output product"* (whatever that means). Click OK.

In the *RTL ANALYSIS* section inside the *Flow Navigator* column click on *Open Elaborated Design* and simply click OK in the info box that opens. After approx. ten minutes the elaborated design is opened, which seems to be an abstraction level very close to a synthesized netlist (which probably explains why opening the elaborated design actually takes so much time . . .). Once the job's done go in the right-top corner of Vivado GUI and select *I/O planning* for layout instead of *Default Layout* (see fig-F.50)

This should change the bottom layout of Vivado, adding the two new tabs *Package Pins* and *I/O Ports*. Under the latter, unroll *Scalar Ports* to have `busy_0` port appear. In the column *I/O Std*, set `LVCMOS33` and in the *Package Pin* column, set placement `R14`. If you refer to the online Reference Manual of the Arty Z7 board[9] you can see in section "Basic I/O" that the `LD0` green led is mapped indeed on the `R14` physical pin of the Zynq package (see fig-F.51). What you should see now in the Vivado main window is illustrated in fig-F.52.

We must record this placement setting in a constraints file that Vivado will automatically add to the project. Type `Ctrl+S` (or click on the floppy button of Vivado main toolbar). The *Save Constraints* window opens, type `ecc-constraints` in the *File name* text field, keep `XDC` for *File Type* and *<Local to Project>* for *File location* (see fig-F.53). Click OK.

A new constraint file named ⟨`ecc-constraints.xdc`⟩ is created and added to the project, as you can see in the *Sources/Hierarchy* tab under the *Constraints* heading (see fig-F.54). Double-click on this file to display its content. As you can see it is a simple file containing two `set_property` Tcl commands (see fig-F.55) which are more or less human-readable.

In the *SYNTHESIS* section of the *Flow Navigator* column (left of the GUI) click on *Run Synthesis* and in the *Launch Runs* window click OK after setting the *Number of jobs* to the maximum (which should be the number of CPUs you chose when creating the VM).

After a few minutes the *Synthesis Completed* window opens (fig-F.56) to inform you that *"Synthesis successfully completed"*. Choose *Run Implementation* and click OK. Again click OK in the *Launch Runs* window to launch the place & route operations. This should take a moment, then an *Implementation Completed* window opens (fig-F.57). Select *Open Implemented Design* and click OK. If a pop-pup window appears inviting you to first close the Elaborated design naturally agree by clicking OK. Vivado opens the implemented design and displays the layout of the design now mapped in the PL part of the device (see *Device* tab).

It is highly probable that Vivado will display two critical messages windows (see fig-F.58 and fig-F.59). The first one is to inform you that the design violates the classical methodology expected from RTL designs, and the second one is to inform you that the timing requirements for the design were not met. Acknowledge both messages by clicking OK and let's investigate what this is about.

30 Let's first analyse what Vivado calls **methodology violations** (the timing requirements violation will be dealt with in next step **31**). As advised in the critical warning message, run the command `report_methodology` in the Tcl prompt at the bottom of Vivado. All the violations found in the design are listed, preceeded with a table of contents (see fig-F.60) giving the different rules that have been violated and for each of them a small description as well as the number of times the violation was found.

[9] `https://digilent.com/reference/programmable-logic/arty-z7/reference-manual`

Note: You can also find these informations in the *Methodology* tab (the bottom half of the Vivado GUI).

A total of six rules are listed among which five concern static timing analysis (the ones whose ID starts with "TIMING-"). The first two rules "TIMING-6" and "TIMING-7" are related, they simply state that there are two clock trees in the design whose logic domain exchange signals but without these two clocks having a common ancestor clock that Vivado could use to infer their timing relation. This is a false problem for us, because the two clocks in IPECC can be totally asynchronous by design assumption. Resynchronization registers are present in the Montgomery multipliers in both directions of the clock-domains crossing and on all control registers, guaranteeing with an extremely high probability that no error is encountered due to metastability. As for data (as opposed to control signals) we use synchronous dual-clock RAM blocks that allow producing data in one clock-domain and consuming them in another clock-domain, with the two clocks being strictly asynchronous to one another[10]. The sequence of operations inside the Montgomery multipliers guarantees that by the time the data written for instance in domain `clk_fpga_0` are read in the domain `clk_fpga_1` they will be stable for a sufficient time, hence producing no data uncertainty.

Notes:

1. The critical warnings discussed so far only concern the asynchronous configuration of the IP (i.e when `async = TRUE` in ⟨`ecc_customize.vhd`⟩). Obviously when the IP is configured with `async = FALSE` it uses only one clock and Vivado won't issue any warning related to clock-domain crossing.
2. The names `clk_fpga_0` and `clk_fpga_1` Vivado uses correspond respectively to the signals named `clk` and `clkmm` in the HDL of the IP. Remember from step **27** that we connected in the block design the input `clkmm` of the IP to the PS-to-PL clock named `FCLK_CLK1`. Thus `clk_fpga_1`, `clkmm` and `FCLK_CLK1` all designate the same signal here.

Besides the signals mentioned below, the Montgomery multipliers also make use of a few signals computed in the `clk_fpga_0` domain by the AXI interface (component `ecc_axi`). These are all the signals named `r.nndyn.*` in the HDL code of the AXI interface (⟨`ecc_axi.vhd`⟩). These signals hold precomputed values that directly depend on the main parameter `nn`. The AXI interface is expected to produce these signals each time a new value of `nn` is written by the software driver. However once computed these signals stay stable for a long period of time, and – more importantly – by the time the first Montgomery multiplication (REDC operation) is susceptible to happen, all these signals are guaranteed to be stabilized. Not only that, but the AXI interface also enforces the insertion of a small amount of waiting cycles after each recomputation of these signals before it releases the remaining logic of the IP. Therefore no uncertainty can happen from the direct sampling of these signals in the `clk_fpga_1` domain (not even without the remedy of resynchronizing registers).

Note:

1. The critical warnings we have just discussed only concern the *dynamic prime size* feature of the IP, that is when `nn_dynamic = TRUE` in ⟨`ecc_customize.vhd`⟩. Would the IP be configured instead with `nn_dynamic = FALSE`, then the values that need a computation based on the static value of `nn` would be computed statically in the HDL code, known and resolved statically by the synthesizer and hence would be «hardwired» to either ground or Vcc.

☞ As a conclusion we can **ignore caterogies TIMING-6 and TIMING-7 of critical warnings.**

Now the critical warnings named TIMING-17, TIMING-18 and TIMING-23 are also related together and we'll discuss them together hereafter. They are all related to the particular structure of the TRNG, which naturally makes a very specific use of the low-level FPGA primitives and how they are connected.

- The critical warnings of category TIMING-17 state that some **sequential cells have their clock-input port connected to a signal which is not a clock** as it's not physically routed along a clock propagation tree but rather propagates through general routing of the FPGA fabric. This is perfectly normal because the structure of ES-TRNG in FPGAs (see [YRG+18]) requires that some D-flip-flops have their clock driven to LUTs in order to implement unstable combinational paths, that is ring oscillators. Take for instance the first item of the list of TIMING-17-like warnings (you're likely to get a different one since this is not perfectly deterministic). In our case (see fig-F.61) it says that the input `C` of the DFF named `design_1_i/ecc_0/U0/t0/t0.t0/bn.bg[0].b/t0/bx2` *«is not reached by a timing clock»*. Now if you open the source file ⟨`es_trng_bit_series7.vhd`⟩ (remember that we've imported this file into our Vivado project in step **13**) you will see that this file is purely structural VHDL and indeed contains an instance of a Xilinx `FDCE`[11] primitive named `bx2`, whose input clock port is indeed connected to a signal named

[10] This is a guarantee provided by Xilinx on their dual-clock BlockRAMs.

[11] Xilinx devices of 7-series family contain four types of flip-flops named according to whether they have a Clear (aka Reset) or a Preset (aka Set) input, and according to this input being synchronous or asynchronous. The name `FDCE` here means that the register is of Asynchronous Clear type.

`ro2out` (see fig-F.62) which is indeed driven by the output `O` of another instance, named `ro2_0`, of a `LUT3` primitive (see fig-F.63). There are 20 warnings of type TIMING-17 (this number should be the same for you) and you can verify that they are all related to the exact same situation (meaning: a combinational signal clocking a flip-flip inside the TRNG)

- There is only one critical warning of category TIMING-18 and it concerns the `busy` output of the IP (that we have tied to the `busy_0` pin of the SoC in step **27**. Vivado ignores the purpose of this signal on the outside and by default can simply assume that is drives some other synchronous logic, in which case a timing constraint like a setup time would need to be assigned to it so as to make the signal switching delays in accordance with the timing requirements of that outside logic. Here we don't mind this at all because the `busy` signal is simply used to drive an indicator LED showing the coarse IP activity.

  In other words we don't mind if the `busy_0` output rises (and the LED lit up) 1 ns or rather 100 ns after the cycle of clock `clk` where a new computation actually started in the IP.
- There are 8 critical warnings of category TIMING-23 and they also all concern the TRNG. They state that there are combinational loops inside the TRNG part of the IP which is normal and is exactly what we want here as LUTs are cascaded in odd number to implement ring oscillators. Vivado is just being thorough and is listing all the signals which are concerned by these loops.

☞ As a conclusion we can **ignore caterogies TIMING-17, TIMING-18 and TIMING-23 of critical warnings.**

The only remaining critical warning is of type LMTCS-1 and states that our design *«uses 343 control sets»* and that *«exceeds the control set use guideline of 7.5 percent»*. Control sets are a combination of three identical clock, reset, and clock-enable signals. Two cells in the circuit that do not share exactly the same three control signals have different control-sets. Control-sets are important to the place-and-route tools because logic cells that use the same control set can be packed together into the same place/ressource inside the fabric, whereas it's not possible to do so when a sole one among the three signals of the control set (clock, reset or clock-enable) is not shared between the two cells.

Be that as it may, the RTL of the IP does not make explicit use of clock-enable signals except on memories (because this allows to save power). The HDL code of the IP is almost entirely behavorial hence using the clock-enable on flip-flops is on the synthesis algorithm.

As for reset signals, care was always taken when writing the HDL code to put a reset (or set) on only the registers for which this was functionaly absolutely necessary. You'll find a large number of comments in the code illustrating that. This warning is therefore probably simply a consequence of the fact that our design is starting to occupy much place in the FPGA (it occupies about the half of the logic ressources) and we will also ignore it. The warning was not issued when targeting a `xc7z020` device (which is twice as large). Furthermore, it's also not issued when the IP is configured with `debug` = `FALSE` (the debug mode adds a non-negligible amount of logic).

☞ As a conclusion we will **ignore the LMTCS-1 critical warning.**

31 Now let's quickly investigate the **timing requirement violations**. Take a look at the bottom part of the Vivado layout where messages and logs are displayed, and find the *Timing* tab. Inside this tab, click on *Design Timing Summary* to get an overview of the timing results. What you should see is illustrated on fig-F.64.

Of course you're not supposed to see exactly the same numbers as FPGA back-end operations like synthesis and place-and-route are not deterministic and hence different runs using the same HDL sources will lead to different placement and timing results. Here we can see that at least one path exhibits a negative slack (-314 ps) with a total accumulated negative slack of -1.033 ns. There are only 7 failing paths, but it only takes one to make the design fail nastily on the real hardware if it's not a false path. Also expand the *Inter-Clock Paths* heading, as the red dot on it suggests that the errors are reported here and again expand the `clk_fpga_0` to `clk_fpga_1` heading (see fig-F.65). These errors are related to the critical warnings we previously discussed about registers `r.nndyn.*`. As illustrated on fig-F.65 with highlighted areas all the failing paths are launched by the input clock port `C` of one of registers `r.nndyn.*` and arrive to the input data port `D` of a register inside one of the two Montgomery multipliers (`mm[0]` or `mm[1]`).

These errors are not important because they concern the `r.nndyn` group of signals (refer to ⟨`mm_ndsp.vhd`⟩) which indeed cross clock-domains, as they are generated in the `clk` clock-domain but used ("read") in the `clkmm` one. As explained before, when `nn_dynamic` option is set to `TRUE` in ⟨`ecc_customize.vhd`⟩, some signals are driven by `ecc_axi` component in the `clk` clock domain (corresponding to the clock signal named `clk_fpga_0` by Vivado here) which are then wired as input to `mm_ndsp` and read in the `clkmm` clock domain (corresponding to the clock domain named `clk_fpga_1` by Vivado here). But these signals only depend on the

value of prime number p transmitted by software driver and, once this number is set, not only can these signals be assumed to stay stable during scalar multiplication, but also we can be confident that they have already set stable by the time the first Montgomery multiplication will take place. In other words, in our context these paths are **false paths** which means their timing can be ignored by Vivado when running static timing analysis on the design. Declaring these paths as false paths will ensure that Vivado won't bother any more with them when checking for timing correctness of the design.

Open constraint file ⟨`ecc-constraints.xdc`⟩ we created earlier and add this line at the bottom of the file:

Add in the XDC file

```
set_clock_groups -asynchronous -group [get_clocks clk_fpga_0] -group [get_clocks clk_fpga_1]
```

The Tcl command to declare false paths is usually the `set_false_path` command but it has to be used once for each path to be declared as false. By using instead the `set_clock_groups` with the `-asynchronous` switch we can save a lot of time as it will do the same thing but at a whole clock-domain level.

Save the file and click on *Run Implementation* again (Vivado left column). Ignore the *Synthesis is Out-of-date* warning by clicking Yes. When Vivado has finished re-running the whole synthesis and implementation flow, choose again *Open Implemented Design* in the *Implementation Completed* window. This time timing requirements warning should not be issued. As for the critical warnings, we'll simply ignore them from now on. The layout of the design can be seen in the *Device* tab of Vivado right panel (see fig-F.66).

32 Before generating the bitstream and handing the design over to the software chain, we're going to look a little bit over the layout of the design. In the *Netlist* panel, expand the hierarchy several times starting from top-level `design_1_wrapper` until you can see the components inside `ecc_0` (see fig-F.67).

Right-click on `a0 (design_1_ecc_0_0_ecc_axi)` and in the drop-down menu scroll down to *Highlight Leaf Cells*. Choose a color for Vivado to display the logic cells associated to `ecc_axi` with. Proceed the same way for remaining components of the IP, using a different colors for each: instance `s0` of `ecc_scalar`, instance `c0` of `ecc_curve`, instance `f0` of `ecc_fp` and the two instances `mm[0]` and `mm[1]` of `mm_ndsp`. Please refer to fig-F.68 (whose colors are defined on fig-F.69). As you can see the two Montgomery multipliers occupy the largest portion of the implemented design (the red one) which is not surprising. Component `ecc_axi` also occupies quite an important area in regards to its function, but we set the IP in debug mode and also with the *dynamic prime size* feature on (`nn_dynamic = TRUE`) and both features carried extra logic to the design. Furthermore component `ecc_axi` is not a trivial AXI target interface, it also implements side-channel countermeasure by masking the value of the scalar when it's written by the software driver into the memory of large numbers of the IP.

33 **Bitstream generation and export**

Click on *Generate Bitstream* in the *PROGRAM AND DEBUG* section of the Vivado Flow Navigator (GUI left column). Accept the default settings of the *Launch Runs* window and click OK. After a short time the *Bitstream Generation Completed* window opens (fig-F.70). Don't choose to open the hardware manager yet as we're going to switch to the software part of the tutorial before actually programming the board FPGA. So choose *View Reports* instead and click OK.

Go to *File* ▶ *Export* ▶ *Export Hardware*. The *Export Hardware Platform* wizard opens (fig-F.71). Click *Next* then choose *Include bitstream*. Replace the default XSA file name `design_1_wrapper` (not a very evocative one) with the more readable `az7-ecc-axi`. Do not add the `.xsa` suffix as Vivado adds it implicitly, otherwise you'll end with a file having the suffix twice. Keep default export path (`~/ipecc/hw/soc/`) then click *Next* and *Finish*. Vivado performs the export quietly so don't be surprised not to get any sucessfull-like message. You can check anyway that file ⟨`az7-ecc-axi.xsa`⟩ has been actually created where expected.

☞ Here ends the hardware part of the tutorial. Next paragraph will give a demonstration of how to use the IP from standalone (aka bare metal) software.

If you're also interested in driving the IP on top of Linux, it's strongly advised that you make a small detour now to step **44** to prepare the compilation of Linux, as the first build from a clean repo takes several hours. Even if the PetaLinux build process requires the XSA file as mandatory input, most of the material that'll be «bit-baked» doesn't rely on the FPGA/PL part of the design.

It'll save you time to start the build now and let it run in the background, even if it means customizing the kernel and the rootfs afterwards to fit exactly the hardware (subsequent compilations will be much shorter).

Software part: driving IPECC from bare metal software

We will know give a quick demonstration on how to program and monitor the IP in standalone mode using the software driver provided in the repository. We will use *Vitis* (the Eclipse-based IDE from Xilinx) to cross-compile the software and then download and run it on the board. The C source files for the driver are located in folder `driver/` of the repo.

☞ The **standalone driver** and the **Linux driver** for the IP are based on exactly the **same API** and **same source files**. There are only two differences between the standalone mode and the Linux mode :

1. The standalone driver is obtained by compiling with `-DWITH_EC_HW_STANDALONE`. The Linux driver is obtained by compiling with either `-DWITH_EC_HW_UIO` or `-DWITH_EC_HW_DEVMEM`, depending on the interface you wish to go through to have the software communicate with the IP (the special device file `/dev/mem` in the latter case, the generic Userspace IO device driver layer, aka generic UIO, in the former). But the exact same API and functionality are provided through both the two interfaces.
2. The standalone demo and the Linux demo don't use the same top-level source file to call the driver API because they don't generate the test vectors in the same way. In both cases, the application aims at pushing test vectors extensively to the IP to test for correctness of the result or simply display it to user. But in the standalone mode test vectors are defined statically (at compilation time) directly from the C sources. While the Linux application expects test vectors to be received on its standard input, thus allowing the IP to be tested either on localhost, or over the network by using a simple tool like `netcat` (this will be shown in steps **54** - **55**).

34 Inside Vivado, project `az7-ecc-axi` being opened, launch Vitis software dev kit: Vivado main menu *Tools* ▶ *Launch Vitis IDE*. Keep default choice for the workspace (`~/workspace`) (fig-F.72).

35 Choose *Create Application Project* (fig-F.73). In the *New Application Project* window click *Next*. In the *Platform* section, select tab *Create a new platform from hardware (XSA)* and browse to select the XSA file we created earlier (step **33**) that is ⟨`~/ipecc/hw/soc/az7-ecc-axi.xsa`⟩ (fig-F.74). Leave the *Generate Boot Components* box checked, click *Next*. For the name of the application project choose `ecc-test-stdalone` (fig-F.75). Domain: nothing to do (leave defaults, click *Next*). Templates: keep default *Hello World*. Click *Finish*.

36 The main Vitis GUI opens with application project (fig-F.78). In the *Explorer* section (top left part of the window) unroll `src` folder of application `ecc-test-stdalone` and double-click on `helloworld.c` to edit the file. As you can see this is the very simple universal Hello World program as per the old *The C Programming Language* of D. Ritchie and B. Kernighan (1978).

Customize a bit the string message in the `printf` to ensure that what you'll see in the console at runtime is not a default or template already loaded on your board. For instance here we'll put instead that message: `"Hello World, this is IPECC test program.\n\r"` (see fig-F.79) and we'll also remove the second print.

☞ We're going to run this very simple Hello World program as is (without any IPECC related stuff for now) to first **test/establish the connectivity of the board with your guest machine.**

This might be a bit tricky because the procedure to follow depends on your virtual host system. The aim is to have your **virtual machine acquire the USB interface of the Arty board in place of the host** when it is plugged into your system.

The following instructions assume you're using VirtualBox (like we are) in which case it should be straightforward. The first thing to do is to check if the **VirtualBox Extension Pack** is already installed on your system (meaning the host). If not, you can follow steps **37** - **38** below to see how to do that. Otherwise you can skip these steps and directly go to step **39**, but before doing so also check that your guest has the USB interface configured in **USB3** (xHCI) mode[12]. You can check this in the settings of your guest machine, in the USB section. You should see something like on figure F.83. Otherwise you'll have to first shutdown your VM, switch the settings to USB3 and run it again.

[12] The point of installing the Extension Pack is precisely that it brings with it, among other things, the support for USB-3.

37 **Adding VirtualBox Extension Pack to get USB3 support**

Outside the virtual machine, browse to the VirtualBox Download page (hope the following link is still valid: `https://www.virtualbox.org/wiki/Downloads`, otherwise g**gle-in).

At the time this tutorial was written VirtualBox 7 was already released but we were still using version 6.1(.36). Note that **you must fetch the version of the Extension Pack that exactly matches your version of Virtualbox** to the third number (here the 36 in 6.1.36). You can find your version number of VirtualBox in the menu *Help* ▶ *About* of the VirtualBox manager window. The filename for the Extension Pack should be a `.vbox-extpack` file (fig-F.80). Navigate to your download local folder and double-click on the file (alternatively there is probably a way to open the file from the VirtualBox manager in case your host OS doesn't know what to do by default with the `.vbox-extpack` file extension). This should open a new VirtualBox window with a pop-up window inside it (fig-F.81). Click on the *Install* button of the latter. You'll be asked for a licence agreement and then your password if you're sudoer or equivalent, or the password of the administrator. Once these steps are done, VirtualBox should inform you that the Extension Pack has been successfully installed (fig-F.82).

38 Back in the guest virtual machine you will now close properly every running application, including Vitis and Vivado, and then power off the machine **but before doing so** and while you're at it, enter the two commands below in a terminal to add yourself to the `dialout` and `vboxsf` user groups:

From your Linux shell prompt

```
$ sudo adduser myself dialout   # Obviously replace 'myself' with your username
$ sudo adduser myself vboxsf    # Obviously replace 'myself' with your username
```

Troubleshooting. If the second command fails with a message saying that group `vboxsf` doesn't exit, this probably means that the Guest Addition CD image was not properly inserted in your guest. You can check about the Guest Addition insertion in the main VirtualBox window (hence on the host): in the left-panel listing all the VMs, find your guest and right-click on it, then choose *Show Log...* In the newly open window look for occurences of the string `Guest Additions` for information report.

Alternatively, if you have the feeling that the VB Guest Additions installation somehow failed, you can try to install them by hand: change dir (in the guest) to `/media/user/Vbox_GAs_x.y.z/` (where `x.y.z` should be the same version numbers as your VirtualBox client) and once there run script `./autorun.sh`.

Adding a user to a group requires rebooting the machine (I'm afraid simply logging out won't suffice) hence executing the above command now will have it taken into account once the VM is up again. Being in the `dialout` group happens to be necessary to communicate with the Arty board over USB.

Now power off the guest machine. Do not reboot, as we'll need the machine to be powered off for the config step we're now going to do. Once the machine is actually shutted down, go to the VirtualBox manager window and open the settings of the guest (meaning obviously the one that you're using for the tutorial). Take a look at the USB section: now the USB-2 and USB-3 options should be available (fig-F.83). Select option `USB 3.0 (xHCI) Controller` and click OK, then power on again the guest.

Once you have the guest running back and ready, quickly check that you've actually been added to `dialout`:

From your Linux shell prompt

```
$ groups
myself adm dialout cdrom sudo dip plugdev lpadmin lxd sambashare vboxsf
```

39 If you've reached this point it means that your virtual machine has the USB3 capability and that USB3 is the current active mode.

On the Arty Z7 board, check that jumper `JP4` is set as illustrated on the photograph of figure F.84. This is to allow the board to be configured in JTAG mode. Also check that no microSD card is present in `J9` slot of the Arty board (simply remove it if there is).

Now power on the board, which you can do in two different ways depending on the settings of jumper `JP5`:

- either you set it in the `REG` position (as in fig-F.132) in which case power can be delivered from your local machine using a USB cable plugged into the micro-USB `J14` connector (labelled **PROG UART**);

- or you set the jumper in the `USB` position (like in fig-F.133) in which case you'll need an AC-DC wall wart adapter (which is not shipped with the board) to provide a voltage in the range 7V-12V into the `J18` Jack input connector of the board.

In either way, connect the board to your host machine with a USB cable through the micro-USB `J14` connector, as we want to communicate with the board.

The Arty Z7 board is populated with an FTDI UART-over-USB device that allows host software to talk to the Zynq device using RS-232 protocol as if the board was connected to your PC using an old DB-9 serial cable instead of an USB one (the footprint of a micro-USB connector is much smaller than that of the old DB-9 connector from the 80's). On Linux systems, at least on Debian-based distribs like Ubuntu, the `ftdi_sio` module is usually already installed and capable to handle the USB-to-UART bridge of the board. It then creates two device files named `/dev/ttyUSB0` and `/dev/ttyUSB1`, only the first one of which will be of interest to us here.

Let's check that this is also the case for you:

From your Linux shell prompt **(Host)**

```
$ ls -1 /dev/ttyUSB*
/dev/ttyUSB0
/dev/ttyUSB1
```

Troubleshooting. If the devices are not listed after the `ls` command, try the following:

1. Check that module `ftdi_sio` is actually loaded on the host using `lsmod | grep -i ftdi`, otherwise load it using command `sudo modprobe -i ftdi_sio` (requires the root proivileges).
2. Use `sudo dmesg` to diagnose enumeration of the device's USB functions by the kernel (also requires the root privileges). You should see something like in figure F.85.
3. Check that USB vendor ID 0403 is handled by the udev rules on your system (`grep -R "0403"` in folder `/etc/udev/rules.d`).

40 Now let the guest machine snatch the USB-to-UART bridge to the host: in the menu bar of the virtual machine, go to *Devices* ▶ *USB*. In the roll-down menu you should see, under *USB Settings...* a list of peripheral among which there should be one named *Digilent Adept USB device*. Select this line, which means asking the VM to acquire the peripheral, as if the virtual machine was a real machine in which the board was actually hot-plugged.

Troubleshooting. If the Digilent device is not present in the list of peripherals check that you're actually a member of group `vboxusers` on the host. Use e.g the `groups` command (you should do that all the more if there's no peripheral listed in the menu at all). Otherwise add yourself to that group using command `sudo adduser myself vboxusers`.

Now the exact same sequence of system and software events is supposed to take place in the guest than it already happened in the host when you plugged the board, so typing `ls /dev/ttyUSB` in a terminal now **inside the guest** should yield the two device files `/dev/ttyUSB0` and `/dev/ttyUSB1` as seen before on the host. Otherwise refer to the troubleshooting list above.

You can configure the virtual machine to automatically acquire the board each time you plug it in by creating what VirtualBox calls a «filter». In the USB Settings for the guest (fig-F.86) click on button  to add a filter. Double-click on the filter name (that would be *New Filter 1* by default). For the `Vendor Id` enter value `0403` and for the `Product ID` enter value `6010` (fig-F.87). Leave other fields empty and validate with OK twice. If you unplug the board and plug it again, you should now observe that its USB interface will be captured automatically by the guest.

41 **Xilinx cable installation** – In a terminal (everything should again be typed inside the guest from now on) change dir to the folder where you installed Vivado in steps **8** - **10**, to wit `~/xilinx/` and change again dir to subfolder `Vivado/2022.1/Vivado/2022.1/xicom/cable_drivers/lin64/install_script/install_drivers`. Here run the script `install_drivers` as sudoer:

From your Linux shell prompt (Guest again)

```
$ cd ~/xilinx/
$ cd Vivado/2022.1/Vivado/2022.1/
$ cd xicom/cable_drivers/lin64/install_script/install_drivers
$ sudo ./install_drivers
[sudo] password:
```

The short log of that script run is given in fig-F.88. As precised on the last line of the log, it's better you unplug the board and plug it back again to have the configuration change to take effect.

We are now ready to drive the board from either Vivado (to program the FPGA with a bitstream) or Vitis (to download executable software to the PS).

42 Install the `screen` program and run it:

From your Linux shell prompt

```
$ sudo apt-get install screen
[sudo] password:
$ screen /dev/ttyUSB0 115200
```

This will run the `screen` console program and attach it to the pseudo device `/dev/ttyUSB0` as if it was talking to a true physical console with a 115 200 bit s^{-1} baud-rate. The terminal should turn blank, with `screen` waiting on any character sent from the board.

Troubleshooting. It's possible that you get a message from the `screen` program at the bottom of the terminal saying `Cannot exec '/dev/ttyUSB0': No such file or directory`. The error stays displayed a few seconds and then `screen` will leave. In this situation ignore the message and retry the same command but this time by replacing `/dev/ttyUSB0` with `/dev/ttyUSB1`.

Open Vitis (don't forget to source the necessary config file ⟨`settings64.sh`⟩ before if you're lauching the tools from command-line interface, as seen in step **11**) and right-click on project name `ecc-test-stdalone`. Choose *Build Project* (fig-F.89). Vitis/Eclipse builds the BSP and then the Hello World executable. Once this is done (fig-F.90), right-click again on project name `ecc-test-stdalone` in the *Explorer* tab and go to *Run As* ▶ *Run Configurations* (fig-F.91). In the *Run Configurations* window that opens click on *Single Application Debug* (fig-F.92) which should activate the top-left button  . Click on that button and in the new visual directly click on *Run* (fig-F.93). A pop-up window opens with a progress bar (fig-F.94) and after a few seconds you should see the message string from the `printf` appearing in the `screen` console (fig-F.95).

Troubleshooting:

1. The support of Vitis for hardware is not particularly stable, meaning that each time you download a program to run on the board, you might get a pop-up window such as the one on figure F.96 telling you that Vitis couldn't find the proper ARM device to talk to on the board. Simply ignore this message and retry the procedure. The error happens almost one out of two trials . . .
2. If you really can't manage to interact with the board despite the presence of the device files `/dev/ttyUSB[01]` in the virtual machine, try launching Vivado in parallel of Vitis with the following sequence of operations: *Open Hardware Manager* ▶ *Open Target* ▶ *Auto-connect* (to launch the Xilinx hardware server). If Vivado achieves to detect the hardware and you get something like on figure F.97, retry the operation on Vitis and hopefully it should work.

43 If you get the Hello World message, it means the Xilinx tools can interact properly with hardware, so now let's play a bit with IPECC.

Instead of fetching the sources for the IP software driver create links inside the `src/` folder of the Eclipse project. Type the following in a terminal:

From your Linux shell prompt

```
$ cd ~/workspace/                # or whatever place you chose for Vitis workspace
$ cd ecc-test-stdalone/src
$ ln -s ~/IPECC/driver/hw_accelerator_driver.h hw_accelerator_driver.h
$ ln -s ~/IPECC/driver/hw_accelerator_driver_ipecc.c hw_accelerator_driver_ipecc.c
$ ln -s ~/IPECC/driver/hw_accelerator_driver_ipecc_platform.c hw_accelerator_driver_ipecc_platform.c
$ ln -s ~/IPECC/driver/hw_accelerator_driver_ipecc_platform.h hw_accelerator_driver_ipecc_platform.h
$ ln -s ~/IPECC/driver/hw_accelerator_driver_socket_emul.c hw_accelerator_driver_socket_emul.c
$ ln -s ~/IPECC/driver/stdalone/ecc-test-stdl.c ecc-test-stdl.c
$ ln -s ~/IPECC/driver/stdalone/ecc-test-stdl.h ecc-test-stdl.h
$ ln -s ~/IPECC/hdl/common/ecc_curve_iram/ecc_addr.h ecc_addr.h
$ ln -s ~/IPECC/hdl/common/ecc_curve_iram/ecc_vars.h ecc_vars.h
$ ln -s ~/IPECC/hdl/common/ecc_curve_iram/ecc_states.h ecc_states.h
```

As you add along symbolic links one by one, you can see that Eclipse dynamically detects the presence of the new source files and automatically adds them to the project. In the end you should see the seven new files in here (fig-F.98).

At this step you can't compile the project yet because there are two C files containing the `main()` function and the linker won't like it. Remove the source ⟨`helloworld.c`⟩ by right-clicking on the filename and selecting *Delete* (confirm with OK).

Now right-click again on project name `ecc-test-stdalone` and this time go to *Properties*. In the *Properties* window that opens unroll *C/C++ Build* in the left column and select *Settings*. Under *ARM v7 gcc compiler* click on *Symbols* and, in the right section *Defined symbols (-D)* click on button  . In the pop-up window, enter value `WITH_EC_HW_ACCELERATOR` and click OK. Repeat the operation to also add the three preprocessor symbols `WITH_EC_HW_ACCELERATOR_WORD32`, `WITH_EC_HW_STANDALONE`, `WITH_EC_HW_STANDALONE_XILINX`. You can also add `WITH_EC_HW_DEBUG` if you want a higher verbosity level.

Click on button *Apply and Close*. Rebuild the application (*right-click* ▶ *Build Project*). Check that the `screen` application we opened in step **42** is still running (otherwise relaunch it). Right-click again on `ecc-test-stdalone` and select *Run As* ▶ *Run Configurations* to run the application on the hardware, but this time before clicking on the *Run* button, go to the *Target Setup* tab of the *Run Configurations* window and check that four boxes are checked as illustrated in F.99, in particular the *Program FPGA* one. When you ran the Hello World application, it wasn't important that the PL part of the FPGA be programmed with a bitstream because everything was done in software and run on the PS (processor). Now we need the PL to be programmed with our hardware design for the driver to have something to drive.

After a few seconds Vitis/Eclipse programs the FPGA, transfers the binary image of the application in DDR memory and branch the Cortex processor to its starting point. The application here consists in programming IPECC with a sequence of tests which are defined in the source file ⟨`ecc-test-stdl.h`⟩. If you take a look at this file, you'll see that it contains at the bottom the definition of an array named `ipecc_all_tests` of elements of type `ipecc_test` (which is defined at the top of the file). Two macros named `IPECC_TEST_VECTOR_NOQ` and `IPECC_TEST_VECTOR_Q` are defined that allow defining IP tests using a simple one line C statement. As stated earlier, the tests here are defined statically (at compilation time). You can see that several tests are defined but only three are uncommented by default, a test on a 24-bit curve on a point of order 2 (which is indicated by its Y-coordinate being null) a test on a 127-bit curve and a test on a 256-bit curve. The log you should get from the application as displayed on the `screen` console is shown on figure F.100.

Let's check the correctness of e.g the 256-bit $[k]\mathcal{P}$ result. Open an interactive instance of *Sage* by simply typing `sage` in a Linux terminal[13] (remember that we've already used the *Sage* tool in step **17**). In the `screen` console, copy all the parameters of the 256-bit test, from line `nn=256` to line `Pouty=0x...`, paste these under the Sage prompt and press Enter (you can do the copy-paste all at once with a single block copy using `Ctrl`+`Mouse`):

[13]Note that the default color scheme of *Sage* is not very compatible with the one of Ubuntu desktop as it makes some patterns like hexadecimal numbers almost unreadable. To fix this you can enter configuration command `%colors Linux` directly under the sage prompt, or add it to your config file ⟨`~/.sage/init.sage`⟩ so it'll be sourced automatically each time you run *Sage*).

From your *Sage* prompt

```
sage: nn=256
....: a=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
....: b=0xee353fca5428a9300d4aba754a44c00fdfec0c9ae4b1a1803075ed967b7bb73f
....: p=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c03
....: q=0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1
....: k=0xf1adb2506355162d0de14468748fb171f730bd40f6595fe1732651df00589fcf
....: Px=0xb6b3d4c356c139eb31183d4749d423958c27d2dcaf98b70164c97a2dd98f5cff
....: Py=0x6142e0f7c8b204911f9271f0f3ecef8c2701c307e8e4c9e183115a1554062cfb
....: Poutx=0x4c0338872b9f13aa0c01f74c7f504eeae9d947f54e9bb80dfce61c3f2e5d151d
....: Pouty=0xa9571e1e0447b7fb302eab3a57dae4866f238eac48f67993a37cc660a831879b
....:
sage:
```

Now enter the following commands one by one:

From your *Sage* prompt

```
sage: Fp=GF(p)
sage: EE=EllipticCurve(Fp, [a, b])
sage: P=EE(Px, Py)
sage: Q=k*P
sage: hex(Q[0])
'0x4c0338872b9f13aa0c01f74c7f504eeae9d947f54e9bb80dfce61c3f2e5d151d'
sage: hex(Q[1])
'0xa9571e1e0447b7fb302eab3a57dae4866f238eac48f67993a37cc660a831879b'
sage: Q[0] == Poutx
True
sage: Q[1] == Pouty
True
sage: Q == EE(Poutx, Pouty)
True
```

Instead of checking all hexadecimal digits one by one, the last boolean tests above asks *Sage* for a direct comparison between the coordinates of $[k]\mathcal{P}$ it has computed itself with the ones computed by IPECC. Also refer to figure F.101. You can notice furthermore on the `screen` console (fig-F.100) that the IP detects the possible nullity of the result point of the computations. In the case of $[k]\mathcal{P}$ computation, the result is stored in point $\mathcal{R}_1$. We can see for instance that the $[k]\mathcal{P}$ computation for `nn=32` is the null point, which is what should be expected from the fact that the input point is of order 2 (as shown by its Y-coordinate being 0) and that the scalar k for the scalar multiplication is even. When the IP returns a point as result of a curve computation, whatever point $\mathcal{R}_0$ or $\mathcal{R}_1$ is expected to store that result, software must check for the corresponding status-flag of that point in register `R_STATUS` to determine if the result is the null point (aka point at infinity). That flag takes precedence over the coordinates of the point. This means that when the point is actually null, data stored in the point coordinates are meaningless and should be ignored by software.

Back to the C source file ⟨`ecc-test-stdl.h`⟩, you can now uncomment more tests if you wish to test them or adding your own extra curve and point configurations very easily.

That's it! We've programmed the IPECC on the real hardware using a piece of standalone software running on the SoC ARM, asking to compute $[k]\mathcal{P}$ computations and displaying the result on the console.

Now let's put aside the quite dry bare metal environment. It's always more fun to play with a rich OS environment like Linux. Using the Ethernet capabilities, we'll drive the hardware directly from a host PC over the network.

Software part: driving IPECC on top of Linux

Make sure you have a working Internet connexion during PetaLinux config and build. PetaLinux tools download a lof of stuff, especially during first build.

44 **PetaLinux prerequisites install** – The first thing to do before installing PetaLinux is to set `bash` as the default shell, because it's the shell required by the tools while Ubuntu by default provides `dash`:

From your Linux shell prompt

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 août 4 15:33 /bin/sh -> dash*
```

Change this using command `dpkg-reconfigure`:

From your Linux shell prompt

```
$ sudo dpkg-reconfigure dash
```

You'll be prompted with a ncurse-like config terminal asking for confirmation to select `dash` (see fig-F.102) select `No` and press Enter. You can check afterwards that `/bin/sh` now points to `bash`:

From your Linux shell prompt

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 août 4 15:33 /bin/sh -> bash*
```

PetaLinux install program checks the presence on your system of prerequisite soft tools and libraries and leaves you with an error for each one that is missing so to avoid multiple trial-and-errors you should install required packages all at once:

From your Linux shell prompt

```
$ sudo apt-get update
$ sudo apt-get install gawk net-tools autoconf libtool gcc gcc-multilib zlib1g:i386
```

45 **PetaLinux download and install** – Now go to the Xilinx PetaLinux download web page and select the **PetaLinux Tools - Installer 2022.1 Full Product Installation.** You'll have to register to the Xilinx web site first, so create an account (for free) in case you don't already own one. Take a special caution to the **2022.1** version number. Also verify the MD5 checksum of the downloaded file (fig-F.103):

From your Linux shell prompt

```
$ cd ~/Downloads
$ md5sum petalinux-v2022.1-04191534-installer.run
5ea0aee3ab9d4c1b138119b0b6613a17 petalinux-v2022.1-04191534-installer.run
```

Move the `.run` file to the place where you want to install the PetaLinux building tools (alternatively you can use the `-d|--dir` option of the install program to select the target folder). In this tutorial we'll use path `~/petalinux`:

From your Linux shell prompt

```
$ mkdir ~/petalinux
$ mv ~/Downloads/petalinux-v2022.1-04191534-installer.run ~/petalinux
```

Add the exec permission to the file and run it:

From your Linux shell prompt

```
$ cd ~/petalinux
$ chmod +x petalinux-v2022.1-04191534-installer.run
$ ./petalinux-v2022.1-04191534-installer.run
INFO: Checking installation environment requirements...
WARNING: This is not a supported OS
INFO: Checking free disk space
INFO: Checking installed tools
INFO: Checking installed development libraries
INFO: Checking network and other services
WARNING: No tftp server found - please refer to "UG1144 PetaL...
INFO: Checking installer checksum...
INFO: Extracting PetaLinux installer...
```

Don't bother the two warnings about the unsupported OS (it's given whatever the OS) and the TFTP server (we won't need one).

Once you've accepted the licence agreements, open a new terminal and type (we'll have to repeat this each time you open a terminal in which you wish to issue PetaLinux related commands):

From your Linux shell prompt

```
$ source ~/petalinux/settings.sh
```

Once again ignore the warning messages. Go to folder `ipecc/sw/linux` we created in step **11** and create a project, named e.g `az7-ecc-axi` after the hardware design, using the command `petalinux-create`:

From your Linux shell prompt

```
$ cd ipecc/sw/linux
$ petalinux-create --type project --template zynq --name az7-ecc-axi
INFO: Create project:  az7-ecc-axi
INFO: New project successfully created in /home/.../ipecc/sw/linux/az7-ecc-axi
```

Enter the newly created project directory and import the XSA hardware description file (created in step **33**) using the `--get-hw-description` switch of the `petalinux-config` command:

From your Linux shell prompt

```
$ cd az7-ecc-axi
$ petalinux-config --get-hw-description  /ipecc/hw/soc/az7-ecc-axi.xsa
```

You'll be prompted with a menuconfig-like window (see fig-F.104). For now don't configure anything, simply choose ⟨`Exit`⟩ and then ⟨`Yes`⟩ to save a default configuration.

You can now start the building process:

From your Linux shell prompt

```
$ petalinux-build
[INFO] Sourcing buildtools
[INFO] Building project
[INFO] Sourcing build environment
[INFO] Generating workspace directory
INFO: bitbake petalinux-image-minimal
NOTE: Started PRServer with DBfile:  /home/myself/ipecc/sw/linux/az7-ecc-axi
...
```

☞ The complete build takes from two to three hours. If you ran this step as the detour that we advised in step **33**, you can now resume back the tutorial to the standalone driver part in page 139 (otherwise you'll have to be patient).

46 **Customizing Linux for our hardware** – In Linux the static description of hardware is implemented using the **device tree** infrastructure: all components are described in a `.dts` (device tree source) file that describes the whole hardware in the form of a hierarchical tree. This file is usually composed by aggregating multiple include `.dtsi` files. The `.dts` file is compiled using the `dtc` (device tree compiler) tool into a `.dtb` (binary version) file which is passed to the kernel at runtime, usually by a second stage boot loader such as Uboot. The kernel dynamically parses the `.dtb` to discover one by one every piece of the hardware (CPUs, memories, disks, peripherals, etc) and for each one the interface, protocol or standard it belongs to, its location inside the hierarchical structure, and **which driver it's compatible with.** This allows the kernel to dynamically load all the necessary modules for a proper support of the hardware.

On Zynq devices the PetaLinux framework (based on the Yocto open-source project) allows to quickly insert any IP of the PL into the device tree and declare a compatible module (driver) for it. The hardware tree description

includes the PL as a peripheral that in turn contains other peripherals, among which the IP. Not only that, but the PL itself is mapped as a device (`/dev/fpga0`) with its own firmware, viz. the bitstream, that you can dynamically update using the command-line tool `fpgautil` (exactly as many peripheral microcontrollers have their internal firmware written or updated by their software driver).

Let's configure the kernel to add the support for the **generic UIO driver** layer (on top of which is based the IPECC driver, see later step **54**). The command lines below assume that you're still in the PetaLinux project folder (otherwise open a terminal, change dir to `ipecc/sw/linux/az7-ecc-axi` and don't forget to source the PetaLinux settings file with `source ~/petalinux/settings.sh`):

From your Linux shell prompt

```
$ petalinux-config -c kernel      # Be patient as Yocto parses its recipes
...
```

In the menuconfig-like window that opens (fig-F.105) go to *Device Drivers* ▶ *Userspace I/O drivers* and press the space bar (only once not to modularize the feature) to get the `⟨*⟩` static selection token (see fig-F.106, note that the token might already by present) then enter the *Userspace I/O drivers* menu itself, go down to *Userspace I/O platform driver with generic IRQ handling* and make it a module by pressing twice the space bar to get the module selection token `⟨M⟩`) (see fig-F.107, feature might already be set as module). Exit, save, and wait for the Yocto black-magic to terminate.

By default Linux cannot be aware that our IPECC software driver relies on the generic UIO layer. When we imported the `.xsa` file and built the kernel in step **45**, PetaLinux/Yocto created a default `.dts` file for the PL with an IPECC instance in it, along with its address mapping (sampled from the `.xsa` file) but the `compatible` field, used to associate drivers to peripheral devices, was left to a default meaningless value. We're going to see that by decompiling the `.dtb` file that was built at that time. For this install first the device tree compiler:

From your Linux shell prompt

```
$ sudo apt-get update
$ sudo apt-get install device-tree-compiler
```

Now let's go to the place where all binaries where produced (`images/linux`) and run `dtc` to get an equivalent `.dts` file back from the compiled `.dtb`:

From your Linux shell prompt

```
$ cd images/linux
$ dtc -I dtb -O dts -o system.dts system.dtb
```

Ignore the many warnings. Open the file `⟨system.dts⟩` that was just generated in a text editor and search for the first occurence of string `ecc@` (see fig-F.108). You should recognize value `0x40000000` assigned to the base address of the IP. This matches the value that was set by Vivado at the time we edited the Design Block for the complete hardware system. If you open project `az7-ecc-axi` back in Vivado, open the Block Design and go to the *Address Editor* tab (see fig-F.109) you'll see that the AXI response port of hardware instance `ecc_0` of the IP was given base address `0x40000000` in the AXI system bus address space, along with a size of 4KB that corresponds to the usual page size on 32-bit systems.
More importantly than the address, you can see in the `.dts` file that `compatible` field is set with a value that is obviously a dymmy one: `xlnx,ecc-1.0`. We're going to modify this by editing a `.dtsi` file that Xilinx provides precisely for customization of the user hardware. Back to the PetaLinux root folder (`az7-ecc-axi/`) change dir to path `project-spec/meta-user/recipes-bsp/device-tree/files` and edit here the file named `⟨system-user.dtsi⟩`. The file simply contains an include of another file named `⟨system-conf.dtsi⟩`. At the bottom of the file add the following lines:

Add at the end of `system-user.dtsi` file

```
/* IPECC */
&ecc_0 {
   compatible = "generic-uio";
};
```

Now rebuild PetaLinux:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-build
```

When the build is over, decompile again file ⟨`system.dtb`⟩ generated in folder `images/linux` using the same `dtc` command as above and take a look again at file ⟨`system.dts`⟩. The `compatible` field should now be set to `generic-uio` (see fig-F.110) instead of the previous dummy value `xlnx,ecc-1.0`.

A last step is required now to properly configure the kernel. We need to make the system be aware that the filesystem we'll use at runtime is a filesystem which resides on a real non-volatile storage device, not simply the memory-live small initramfs that Linux embeds with its kernel to spawn the system at boot time. To do that run the `petalinux-config` command, again from folder `images/linux`:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-config
```

In the menuconfig window that opens (fig-F.104) browse to *Image Packaging Configuration* ▶ *Root filesystem type* and select `EXT4 (SD/eMMC/SATA/USB)` (see fig-F.111)

Build again the whole system:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-build
```

47 **Customizing rootfs** – We're now going to generate the root filesystem by adding the following components/programs to our small embedded distribution:

- a real shell like `bash`
- an SSH server so as to be able to log onto the Arty board through the network
- The `netcat` program, a small multi-usage networking tool which is a bit like a swiss-army knife for network applications.

Run the `petalinux-config` command again, this time with the `-c rootfs` switch to select the `rootfs` component of PetaLinux:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-config -c rootfs
```

In the menuconfig window that opens, go to *Filesystem Packages* ▶ *base* ▶ *shell* ▶ *bash* and select `bash` (fig-F.114). Similarly navigate to *Filesystem Packages* ▶ *net* ▶ *netcat* and select `netcat` (fig-F.115).
For the SSH server go likewise to *Filesystem Packages* ▶ *misc* ▶ *packagegroup-core-ssh-dropbear* and select `packagegroup-core-ssh-dropbear` (fig-F.116). Also add the `coreutils` package: go to *Filesystem Packages* ▶ *misc* ▶ *coreutils* and select `coreutils`. Exit with saving and run again `petalinux-build`, this time with the `-c rootfs` switch (to restrict the build to the root filesystem).

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-build -c rootfs
```

When the build is over, the root filesystem should be in a tarball format in folder `images/linux` along with all other binary files (FSBL, U-boot, kernel, etc).

48 **Customizing bootargs of the kernel** – We now need to modify the list of arguments that will be passed to the kernel at boot time, in particular to ask for the loading of the generic UIO driver. Run again the configuration of PetaLinux:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-config
```

In the menuconfig window that opens, go to *DTG Settings* ▶ *Kernel Bootargs* (DTG stands for Device Tree Generation). By default the *generate boot args automatically* option is probably selected (with token ⟨*⟩) so unselect it. Scroll down to the line below and press Enter to edit the list of arguments. This one probably looks like the following:

```
console=ttyPS0,115200 earlycon root=/dev/ram0 rw
```

So replace it with this (see fig-F.112 and fig-F.113):

```
earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk0p2
rw rootwait uio_pdrv_genirq.of_id=generic-uio
```

The last argument is to tell the kernel to load the `generic-uio` driver. You can also notice that the root file system is not anymore an init RAM filesystem (`/dev/ram0`) but instead the partition (`/dev/mmcblk0p2`) of the microSD card.

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-build
```

49 Now the command `petalinux-package` will wrap for us everything we've previously built (kernel, root filesystem, U-boot, first stage boot loader and bitstream for the PL) in a format that can be exported to the microSD card for a proper boot of the Arty board:

From your Linux shell prompt

```
$ cd ~/ipecc/sw/linux
$ petalinux-package --force --boot --fsbl --u-boot --fpga images/linux/system.bit
```

Ignore the two warnings you'll probably get about overlapped partitions range (`:-.`) The files we need are ⟨`BOOT.BIN`⟩, ⟨`boot.scr`⟩, ⟨`image.ub`⟩ and ⟨`rootfs.tar.gz`⟩ and they are all located in `images/linux` (in case you're wondering where the bitstream is, it's been wrapped into file ⟨`BOOT.BIN`⟩).

50 **Creating a bootable SD card** – The Arty Z7 board doesn't come shipped with a microSD card alas, so you'll need to get one on your own. A simple 4 GB card will suffice which only costs a few dollars. Insert the microSD card onto your host machine. Depending on whether your PC is populated with an SD card slot, you may have to use an SDcard-to-USB adapter (fig-F.117) and possibly a microSD card to SD card adapter (fig-F.118). If you use that one, make sure the tiny mechanical *lock* switch on the adapter is released (see fig-F.119) otherwise you won't be able to format nor modify the content of the card. If the system on your host is configured with the automount feature, the partition(s) on your card will be automatically mounted on the system but we don't want that, as we're going to format the microSD card. So first you must ensure that whatever the number of partition(s) is present on the microSD card, they are all unmounted. For instance on Ubuntu you can do this by right-clicking on the disk icon representing each of the partitions of the microSD card in the left-panel of the desktop and choose *Unmount* (not *Eject*!), see fig-F.120. Alternatively you can use the good old command `sudo umount` on the device files corresponding to the partitions of the microSD card, the ones you'll easily identify with command `lsblk` in a terminal (the partitions will most likely appear as `/dev/mmcblk0p1`, `/dev/mmcblk0p2`, etc).

Now that you've unmounted the SD card, launch the *GParted* program on your host. Obviously you'll need the root privileges here:

From your Linux shell prompt **(Host)**

```
$ sudo apt-get install gparted
...
$ sudo gparted
```

In the main GUI of the program, select, in the roll-down menu of the top right corner, the device file `/dev/mmcblk0` (see fig-F.121, that screen capture shows an SD card formatted with three partitions).

Take extreme caution when selecting the device in *GParted*. Taking actions on the wrong device will make you lose data permanently, with no possible step back. Keep in mind that you're running *GParted* as super-user.

In the *Device* menu of *GParted*, select *Create Partition Table...*. Keep default `msdos` partition type (fig-F.122) and click *Apply*. The partitions are erased from the SD card (fig-F.123). In the *Partition* menu select *New*. Set the configuration as shown below:

- `New Size`: 500 MiB
- `File system`: `fat32`
- `Label:` `BOOT` (or whatever name you'll find more appropriate)

Leave all other settings as default. Proceed identically to create a second partition using this time the following settings:

- `New Size`: [*GParted* automatically computes the remaining size of the SD card following the first partition and will set the result here, keep that value].
- `File system`: `ext4`
- `Label:` `ROOTFS` (or whatever name you'll find more appropriate)

Click on the *Apply All Operations* button  of *GParted*, confirm your action and wait for the operation to complete (fig-F.124). Right-click on the first `fat32` partition, select *Manage Flags* and check the `boot` box (fig-F.125) then close. You should get what is shown on figure F.126. You can quit *GParted*.

Now we need to transfer the different binary files required for the boot into the microSD card. There are three ways to do that:

1. You can create a shared folder between the guest VM and the host, transfer the files from the former to the latter, then with the microSD card mounted on your host, simply copy the files to the card.
2. You can use the network to transfer the files (e.g by mail or through an online drive) from the guest VM to the host.
3. You can mount the microSD card on the guest and copy the files into it.

The last solution is a bit tricky which is why we'll show you how to do it. For a disk to be mounted on a guest, VirtualBox first requires that a `.vmdk` file be created for it. With the microSD card still in your host machine, ensure the two partitions have been kept unmounted (some systems dynamically automount block peripherals after their repartitioning).

Troubleshooting. In case you can't find the microSD card partitions `/dev/mmcblk0p1` and `/dev/mmcblk0p2` in the log of `lsblk`, this means that they were not only unmounted but also removed by your system after the partitioning. This is very unlikely to happen but in this case simply physically remove the microSD card and reinsert it again. This should have the system to enumerate again the partition table of the card. If again the system automounts the partitions, simply unmount them (without removing/ejecting them ...)

We're going to use the `VBoxManage` command (which is part of the VirtualBox installation) to create a file named ⟨`sdcard.vmdk`⟩ (obviously you can choose whatever filename you'll find more suitable here). Also note that the command hereafter assumes that the microSD Card has been enumerated by your system under device file `/dev/mmcblk0` which is most likely to be the case). You can create the `.vmdk` file wherever you want, however it's a good practice to do this in the folder where VirtualBox keeps the storage-file and the metadata for your guest VM, e.g in ~/`VirtualboxVM/ubuntu 22.04.2 LTS`:

From your Linux shell prompt **(Host)**

```
$ cd ~/VirtualboxVM/ubuntu 22.04.2 LTS
$ VBoxManage internalcommands createrawvmdk -filename sdcard.vmdk -rawdisk /dev/mmcblk0
RAW host disk access VMDK file sdcard.vmdk created successfully.
$
```

Troubleshooting. If the command fails, check your membership to the `disk` group (on the host system mind you, not the virtual one). If you're not a member you'll need to execute command `sudo adduser myself disk` and reboot the host machine (and therefore also the virtual one before) to have your membership to groups updated.

Make sure that the file was actually created:

From your Linux shell prompt **(Host)**

```
$ ls -1
Logs/
sdcard.vmdk
'ubuntu 22.04.2 LTS.vbox'
'ubuntu 22.04.2 LTS.vdi'
...
```

Open the *Settings* window for your guest VM and select *Storage*. In the *Storage Devices* panel, click on *Controller: SATA*. The two small buttons  should appear, click on the rightmost one. In the *Hard Disk Selector* window that opens, click on the *Add* (leftmost) button. In the pop-up window navigate the filesystem to select the file ⟨`sdcard.vmdk`⟩ you previously created (it's likely you'll be presented directly with the proper folder). Click OK. Fig-F.127 shows an example with an 8 GB microSD card. Select the `.vmdk` file and click on the *Choose* button. Back in the *Settings* window, the file should now be listed under the *Controller:SATA* heading (see fig-F.128). Close the window with OK. The guest system should automatically mount the two partitions of the card.

Troubleshooting. It's possible that VirtualBox won't allow you to hotplug the virtual disk - you'll know that from the two buttons mentioned above being shaded. It is puzzling as why the feature is enabled or not (it was an erratic behaviour on our part). Anyway if the buttons are not actionable you'll have to shutdown the guest, add the `.vmdk` according to the procedure we've just described, then restart the virtual machine.

Troubleshooting. If the guest VM does not automatically mount the microSD card partitions, simply mount them by hand using `sudo mount /dev/mmcblk0p1 /media/myself/BOOT` and `sudo mount /dev/mmcblk0p2 /media/myself/ROOTFS`. Before doing so however, check that the device files corresponding to the two partitions are indeed `/dev/mmcblk0p1` and `/dev/mmcblk0p2`, and of course create the two mount points `/media/myself/BOOT` and `/media/myself/ROOTFS` (where `myself` designates your user login) using `mkdir` as sudoer.

The next step will assume that the two partitions of the microSD card are mounted in the guest machine.

51 **Finalizing the bootable SD card** – Now that the two microSD card partitions are mounted in the guest machine, we can transfer there everything that is expected on the Zynq platform to run a proper boot. As it's often the case in Linux embedded systems, two different things are expected in two different places: U-boot and the Kernel image are expected in the first boot partition (the `FAT` one), and the root filesystem is expected in the second `EXT` one. The kernel will start its execution using a RAM-live minimal filesystem and then later on mount the `EXT` filesystem and change root («chroot») to it.

Let's populate the boot (`FAT`) partition. Change dir to folder `images/linux` where we've seen the kernel image has been built by PetaLinux and copy the three files ⟨`BOOT.BIN`⟩, ⟨`image.ub`⟩ and ⟨`boot.scr`⟩ into the boot partition:

From your Linux shell prompt

```
$ cd images/linux   # Remember this is the dir.  where PetaLinux binaries where dropped down
$ cp BOOT.BIN image.ub boot.scr /media/myself/BOOT/
$ sync              # To flush-write all files onto the SD card
```

Now copy the root filesystem:

From your Linux shell prompt

```
$ sudo tar -C /media/myself/ROOTFS -xzf rootfs.tar.gz
$ sync
```

While we're at it, we're going to modify the config file of Bash so as to get a friendlier command line interface. Open file ⟨`.bashrc`⟩ which is in folder `/media/myself/ROOTFS/home/petalinux` (`petalinux` is the name of the user that PetaLinux creates by default in the root filesystem) and uncomment the line 7 and lines 9 to 11. Also modify value of the shell prompt variable `PS1`, making it the weird cryptic character string `'\[\033[01;33m\]\h\$\[\033[00m\] '` (this is just ASCII text mind you, the colors are simply here to emphasize that the text consists in escape control sequences for the shell). See fig-F.129 for look of file ⟨`.bashrc`⟩. Note that you can also set in the file any other configuration you'll like according to your preferences, but mind that this is an embedded system with only a tiny set of software tools available in the filesystem.

Troubleshooting. Be cautious with the SD card when its partitions are being mounted in the virtual machine. VirtualBox seems not quite stable with dynamically mounted storage devices, so don't try to mount the partitions on the host while they are already mounted in the guest, nor the other way round, and try not to inadvertently eject the microSD card from the hardware while it is mounted in the guest. You may otherwise experiment hanging of the VM.

Troubleshooting. Also note that if later on you try to restart the VM after having shut it down, VirtualBox may refuse to boot it *if the SD card is no longer physically present in the host.* This is because VirtualBox refuses to start a virtual machine when the current hardware configuration doesn't match what is described in the config files for that VM, typically if a storage device should be here according to the VM config files but is not actually visible in the hardware. In that case, and assuming you won't need to mount the SD card in the guest system anymore, simply remove the `.vmdk` file from the VM configuration (using menu *Settings*>*Storage* for that VM, and of course while the VM is shut down) and try again.

This is it. The SD card is ready to boot. Ensure a safe removal of it (i.e tell the guest OS to unmount it before tearing it off).

52 **Booting Linux** – Plug the microSD card into slot `J9` of the Arty board (fig-F.130) and change position of jumper `JP4` as illustrated on fig-F.131 to now have the board to boot from SD card.

Depending on the position of jumper `JP5` the board can be powered from either a USB cable or from a wall wart with an output DC voltage anywhere within the range 7 V to 15 V. The Arty Z7 board doesn't come shipped with such an adpater but it's easy to find one as many small electrical appliances of the everyday life use this kind of voltage adapter. If you get to grab one then set the jumper `JP5` on position `REG` (fig-F.132) otherwise set it on the other position `USB`.

Plug the USB cable supplied with the board into your host machine on its type-A end () and into the `J14` connector of the Arty board on its micro-type-B end ().

With the guest still running, and given the presets we made in steps **37**-**38**, the VM should immediately capture the two interfaces `/dev/ttyUSB0` and `/dev/ttyUSB1`, in which case you can run in a new terminal:

From your Linux shell prompt (Guest)

```
$ screen /dev/ttyUSB1 115200
```

If the guest didn't capture the UART interfaces, try and run the same command on the host side[14].

As soon as the board is powered-up the Xilinx FSBL configures the FPGA very fast, as you can see from the `DONE` green led switching on. You can see U-boot catching and executing the ⟨`boot.scr`⟩ script, then the kernel is loaded and starts its boot process (fig-F.134). After a few seconds the login prompt is displayed. Log in using `petalinux` as user name. Since this is the first time you are logged in and no password fingerprint has yet been defined in `/etc/shadow`, you'll need to enter a new password and to confirm it (choose a strong password here, like `foo` or `bar ?-.`). See figure F.135.

53 **Connect the board to your local network** – The PetaLinux distribution is configured by default to use a DHCP client, so:

- either you have access to a local wired network, in which case you'll simply need to use an RJ45 cable to connect the board to it (but for this to work you may need to contact your network administrator)
- or you don't, in which case you'll have to create a point-to-point connexion between your board and your guest machine (for which you'll need the administrator privileges).
  You can quickly set a point-to-point connexion like this:
  - on the board side: with PetaLinux up and running, enter following command into a shell:

    From your Linux shell prompt (Guest)
    ```
    $ sudo ip addr add 192.168.111.38/24 dev eth0
    ```
  - on the guest side: go to Ubuntu settings ▶ Network ▶ Wired ▶ click on button ⚙ ▶ IPv4 tab ▶ check *Manual* box ▶ Set `Address` field to `192.168.111.x`, where `x` is any byte value different from 38.

We won't go into further details on these aspects as they're too specific of local network configuration. We'll consider in the remainder of the tutorial that your board is granted an IP address on the same local network as your guest machine.
Alternatively, if don't have access to network, you can use the microSD card as a poor but nonetheless functional solution to transfer the test-vectors file generated by Sage (see step **55** below) from your guest to the board.

54 **Building IPECC driver** – The software driver for IPECC is a user-space driver. As already mentioned in step **46** it uses the generic UIO layer of Linux (`https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html`) which is a subsystem of the kernel that allows driving a hardware device directly from a user application whenever a few memory-mapped I/O read/writes are enough to properly instruct the driver of what it should do. The UIO feature also supports asynchronous interrupts but for this tutorial we'll restrict ourselves to talk to the hardware using state polling.

On the guest machine, install the ARM GCC cross-compiler:

From your Linux shell prompt
```
$ sudo apt-get update
$ sudo apt-get install gcc-arm-linux-gnueabihf
...
```

Copy source folder `driver` (contains sources for both the driver the IP test application):

From your Linux shell prompt
```
$ cp -Rf ${IPECC}/driver ~/ipecc/sw/linux/driver
$ cd ~/ipecc/sw/linux/driver
```

At this stage trying to compile the sources wouldn't work as a slight modification of the local `./Makefile` is required. Edit `./Makefile` and modify its line 3 containing empty definition of variable `VHD_DIR` so as to

[14] At least if it's a Unix brand. If you're using a Windows or a MacOS then try any other terminal program such as *Putty* or other, there are many ones freely available on the Internet. The only important thing is to hook your console on the proper UART interface and with the proper baud-rate (115 200 bit s^{-1}).

make it to point to the directory where you built the IP microcode in step 5. If you rigorously followed the tutorial, this should be `/home/myself/IPECC/hdl/common/ecc_curve_iram/`.
Do not use a relative path when setting the variable! Only absolute paths are supported by `make`.
Once you have edited and saved file `./Makefile` (fig F.136) try again building the driver and the IP test application with `make`:

From your Linux shell prompt (Guest)

```
$ make ecc-test-linux-uio
arm-linux-gnueabihf-gcc -Wall -Wextra -Wpedantic -O3 -g3
...
```

Alternatively, you may also run `make` using an inline definition of variable `VHD_DIR` like this:

From your Linux shell prompt (Guest)

```
$ make VHD_DIR=/home/myself/IPECC/hdl/common/ecc_curve_iram ecc-test-linux-uio
arm-linux-gnueabihf-gcc -Wall -Wextra -Wpedantic -O3 -g3
...
```

Note that the compilation is a bit slow as we set the `-O3` optimization switch to `gcc`. If you happen to modify the sources of the driver and wish to gain some time you may remove this in the Makefile.

The executable is named ⟨`ecc-test-linux-uio`⟩ and is built in place:

From your Linux shell prompt (Guest)

```
$ ls -1
ecc-test-linux-uio
hw_accelerator_driver.h
hw_accelerator_driver_ipecc.c
hw_accelerator_driver_ipecc_platform.c
hw_accelerator_driver_ipecc_platform.h
hw_accelerator_driver_socket_emul.c
linux
Makefile
stdalone
```

We need to transfer this file onto the board using e.g `scp` (the OpenSSH secure copy tool) but for that we need the network IP address the board obtained from DHCP:

From the `screen` console (Arty board)

```
az7-ecc-axi$ ip a
...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
   link/ether ...
   inet 192.168.111.38/24 brd 192.168.111.255 scope global eth0
...
```

In the example below the board got the address `192.168.111.38`, which is what will be assumed in the remainder of the tutorial (of course adapt this to your own setting).

Copy the executable file to the board with `scp` as user `petalinux`, authenticating yourself with the password you chose in step 52:

From your Linux shell prompt (Guest)

```
$ scp ecc-test-linux-uio petalinux@192.168.111.38:.
petalinux@192.168.111.38's password:
```

Now use the same IP address to open a secure connexion to the board through `ssh`, with the same credentials:

From your Linux shell prompt (Guest)

```
$ ssh petalinux@192.168.111.38
petalinux@192.168.111.38's password:
```

Troubleshooting. In case `ssh` aborts telling you that *«the remote host identification has changed blabla»*, simply remove your config file `~/.ssh/known_hosts` and retry.

You can put aside now the `screen` console. Keep it opened if you want but everything from here and in the remainder of the tutorial will be made through the `ssh` session.

On the Arty board, change permissions of the device file `/dev/uio0` (this is the file identifying the IPECC instance in the Linux UIO layer - there's only one peripheral using the UIO driver, so it's been logically assigned number #0):

From your ssh prompt (Arty board)

```
az7-ecc-axi$ ls -l /dev/uio0
crw------- 1 root   root   246,   0 Jan 1  1970 /dev/uio0
az7-ecc-axi$ sudo chmod 666 /dev/uio0
Password:
az7-ecc-axi$ ls -l /dev/uio0
crw-rw-rw- 1 root   root   246,   0 Jan 1  1970 /dev/uio0
```

Troubleshooting. If file `/dev/uio0` doesn't exist, this means that the UIO module was not properly loaded by the kernel. Check this using the `lsmod` command: its log should feature a line with `uio_pdrv_genirq`. Otherwise you can try to load the module by hand using `sudo modprobe uio_pdrv_genirq`. Also check that you didn't forget to edit the command line of the kernel when you configured PetaLinux (see step **48**) using e.g `cat /proc/cmdline`: the command line should include the argument string `'clk_ignore_unused uio_pdrv_genirq.of_id=generic-uio'`.

Troubleshooting. If you're targeting another board than the ArtyZ7, take care that there might be other hardware devices also enumerated as UIO devices in the system, in which case IPECC might not necessarily be attached to `/dev/uio0` but perhaps to subsequent devices (`uio1` etc).

Run the IPECC test application program:

From your ssh prompt (Arty board)

```
az7-ecc-axi$ stdbuf -oL nc -l -p 20000 | ./ecc-test-linux-uio
Driver in UIO mode
IP in debug mode (HW version 1.2.25)
```

The test application starts by opening device file `/dev/uio0` and then asks for an `mmap` system call on that file to make the register map of the IP to be accessible in its address space. The program detects that this version of the IP was synthesized in `debug` mode (remember from step **26**).

The version numbers displayed are the ones available from the IP register `HW_VERSION` (major, minor and patch). The program `nc` designates `netcat` (c.f step **47**) that acts here as a simple network pass-through (`-l` means the listen/server mode) allowing us to virtually connect the IP test program with the one that is going to send test vectors to it just as if the two programs were connected with a pipe on the same physical machine (this means that you can also generate tests into a simple text file, transfer this file to the board, and «`cat`» it into the program standard input through a pipe). We use the port `20000` but this is a totally arbitrary choice.

The test application then stays idle, waiting for test-vectors to be pushed to it in the exact same format as the one we used when we simulated the IP in steps **16**-**25**.

55 **Testing the IP over the network** – Back on the guest machine, open a new terminal and change dir to `sage` folder:

From your Linux shell prompt (Guest)

```
$ cd ~/ipecc/sw/sage
```

Edit file ⟨`generate-tests.sage`⟩. We've already met this file in step **17** when we generated a few test-vectors to push into the IP simulation testbench. As you can see there is a large text frame at the begining of the file containing definitions of variables. These variables define the way test-vectors are generated. Each variable comes with an explanation that you may take the time to read if you want, however the modifications that we'll do here are very simple.

The script is intended as some kind of fuzzing tool testing numerous random configurations for all the operations provided by the hardware API (remember we already met it in step **17** to generate test-vectors for the IP simu testbench). It was helpful in identifying a few side effects and corner cases in the RTL during the IP development and you might also consider to use, modify or adapt it for your own application if you wish to add modifications to the hardware.
The script consists in a simple loop that randomly generates elliptic curves of a randomly generated size, and for each of these curves randomly generates a given number of test vectors involving random points, random scalars along with random exception cases (e.g small order points).
At any time the parameter `nn` used to generate the underlying field of the curve is drawn from within the range `[nnmin, nnmax]`, where `nnmin` and `nnmax` are parameters that each start at a predefined value and are periodically increased until they hit a predefined plateau value. This way of doing allows to start testing the IP (and the possible modifications you might want to bring in yourself) very fast using very small values for `nn`. Numerous tests, each performed in a very short time, will allow you to quickly detect possible anomalies in your RTL updates. Now with `nn` values increasing, the tests will soon reach values more reflecting of a cryptographic usage of the IP, also allowing you to assess performance of the IP.

Now search for the following parameters below in the script and set each one to the value that is given alongside:

```
nnmaxabsolute = 256    # no curve/test will be of field size nn > 256
nnminmax = 256         # after a while only 256-bit curves will be generated
NNMINMOD = 10          # nnmin will increase every round of 10 curves...
NNMININCR = 32         # ...and it will be increased by 32
NNMAXMOD = 5           # nnmax will increase every round of 5 curves...
NNMAXINCR = 32         # ...and it will be increased by 48
nn_constant = 0        # we don't want a constant value of nn
NBCURV = 0             # the script will iterate indefinitly
NBKP = 100             # 100 scalar-multiplication tests generated per curve
NBADD = 10             # 10 point-addition tests generated per curve
NBDBL = 10             # 10 point-doubling tests generated per curve
NBNEG = 10             # 10 point-negate (−P) tests generated per curve
NBCHK = 10             # 10 boolean "is P on curve?" tests generated per curve
NBEQU = 10             # 10 boolean "are points equal?" tests per curve
NBOPP = 10             # 10 boolean "are points opposite?" tests per curve
NO_EXCEPTION = False   # the tests will include exception cases
```

Whatever the values you set for `NNMINMOD` and `NNMAXMOD`, keep `NNMAXMOD < NNMINMOD`. Otherwise the script will undoubtedly fail at some point as `nnmin` and `nnmax` turn out to be inverted.

Save and exit, then run the script with `sage`, piping the command into `nc`, this time in client mode (no `-l` switch) and targeting the IP address of the board and on the same `20000` port:

From your Linux shell prompt (Guest)

```
$ sage generate-tests.sage | tee /tmp/arty-z7.txt | stdbuf -oL nc 192.168.111.38 20000
Generating curves from nn = 32 to 48
Generating curves from nn = 32 to 80
Generating curves from nn = 64 to 112
...
```

In the example code above we also intertwined a `tee` process, so as to intercept everything sent to the board and dump it in the temporary file ⟨`/tmp/arty-z7.txt`⟩.

In the SSH console on the board you should start seeing the IP test program displaying statistics on the tests received (fig F.137). At the begining tests received are on small curves (`nn = 32`). Cryptographic sizes (e.g `nn = 256`) are reached after a few minutes. The point operations performed by the IP are organized in colums, from left to right:

1. Scalar multiplication (computes $[k]\mathcal{P}$) in column labelled `[k]P`
2. Point addition (computes $\mathcal{P}+\mathcal{Q}$) in column labeled `P+Q`
3. Point doubling (computes $[2]\mathcal{P}$) in column labeled `[2]P`
4. Point negation (computes $-\mathcal{P}$) in column labeled `-P`

and the three logical tests:

5. Are the two points equal? (tests if $\mathcal{P} = \mathcal{Q}$) in column labeled `P==Q`
6. Are the two points opposite? (tests if $\mathcal{P} = -\mathcal{Q}$) in column labeled `P==-Q`
7. Does this point belong to the curve? (tests if $\mathcal{P} \in \mathcal{C}$) in column labeled `PonC`.

For each operation the array shows the number of tests submitted to the IP, the number of correct tests (in green) and the number of errors (in red). The default behaviour of the test program is to break execution as soon as an error is encountered.

☞ Needless to say no error is expected to occur! **The IP has been thoroughly tested both in simulation and on real hardware FPGA platforms.** Many errors, corner cases and side effects were found and fixed, and the RTL is now **highly stable**. At time of version 1.2.25 there remains two known bugs of minor importance which are described in §?.

Should you encounter a new error case anyway, please don't hesitate to submit a bug report to the authors.

If you observe the content of file ⟨`/tmp/arty-z7.txt`⟩ while the test vectors are being transfered from the host machine to the board, e.g by using command `tail -f /tmp/arty-z7.txt` in a new terminal, you may notice that the value of `nn` seems to increase faster in the file than what it seems on the board according to the average value of `nn` that is being displayed. This is because the board computes $[k]\mathcal{P}$ much slower than the host machine does in *Sage*. The FPGA on the board runs the two Montgomery multipliers inside the IP at 250 MHz which is one magnitude of order less than let say the 2 GHz which clocks a x86 multicore processor these days. The FPGA being the bottleneck, test-vectors are buffered both by the embedded Linux and the host/guest machines, which is why they'll be dumped in file `/tmp/arty-z7.txt` sometimes several minutes before being actually processed by the IP. You can assess this by hitting `Ctrl-C` in the host/guest terminal running the *Sage* script (not on the board), the IP will continue processing test vectors for several minutes before quitting.

Troubleshooting. You might sometimes experience a stall on the network link, as if netcat was sending no data on the host side, or as if no incoming data was seen on the board side. In this case simply break both TX and RX applications with `Ctrl-C` and start again. This is not a very satisfactory solution but it will work eventually.

Alternatively try another TCP port (than the anyhow arbitrary 20000 we used in the example above).

You can also try to completely remove buffering on the transmission path between the two programs by using the set of options `-i0 -o0 -e0` (in place of `-oL`) of `stdbuf` on both sides.

You can interrupt the IP test program by hitting `Ctrl-C` in the board console, which will also interrupt the sending process on the host machine as result of TCP socket termination.

That's it! This tutorial is over. You have simulated the IP, synthesized its RTL targeting a Xilinx FPGA device and tested it on real hardware using a driver on top of an embedded Linux.

Next paragraph will show you how to update the PL bitstream at runtime, without having to reboot the board nor to rebuild a complete root filesystem.

56 Updating PL bitstream at runtime

PetaLinux sees the PL (the logic fabric) as a peripheral it can write to using the special device file ⟨`/dev/fpga0`⟩.

A program called `fpgautil`, provided by default in the PetaLinux distrib, allows to reprogram the PL at runtime. This is very useful to update the hardware portion of the design without having to rebuild Linux and the rootfs

again each time. The program `fpgautil` expects the bitstream to be in a specific binary format, so a conversion step is required that only takes a few seconds. In Vivado, with project `az7-ecc-axi` opened, type the following in the Tcl console (bottom of GUI):

In the Vivado Tcl console

```
write_cfgmem -force -format BIN -interface smapx32 -disablebitswap -loadbit "up
0x0 /home/myself/ipecc/hw/soc/az7-ecc-axi.runs/impl_1/design_1_wrapper.bit"
/home/myself/ipecc/hw/soc/az7-ecc-axi.runs/impl_1/design_1_wrapper.bin
...
write_cfgmem completed successfully
```

Note that both files ⟨`design_1_wrapper.bit`⟩ and ⟨`design_1_wrapper.bin`⟩ are named after the basename of the design block file `design_1` we created in step **27**, so you may have to adapt the command to your own settings obviously (same for the paths in the command).

You can then use e.g the `scp` command to transfer the new `.bin` file to the Arty board:

From your Linux shell prompt (Guest)

```
$ cd /home/myself/ipecc/hw/soc/az7-ecc-axi.runs/impl_1/
$ scp design_1_wrapper.bin petalinux@192.168.111.38:.
petalinux@192.168.111.38's password:
```

Now to update the PL with the new bitstream execute the simple command below on the board, as sudoer:

From your `ssh` prompt (Arty board)

```
az7-ecc-axi$ sudo fpgautil -b ~/design_1_wrapper.bin
Password:
fpga_manager fpga0:  writing design_1_wrapper.bin to Xilinx Zynq FPGA Manager
Time taken to load BIN is 43.000000 Milli Seconds
BIN FILE loaded through FPGA manager successfully
az7-ecc-axi$
```

Obviously try not to do this while the hardware in the PL is being adressed by software (e.g when the IPECC driver is issuing commands to the IP) or the CPU might stall due to a broken AXI transaction.

F.3 Figures

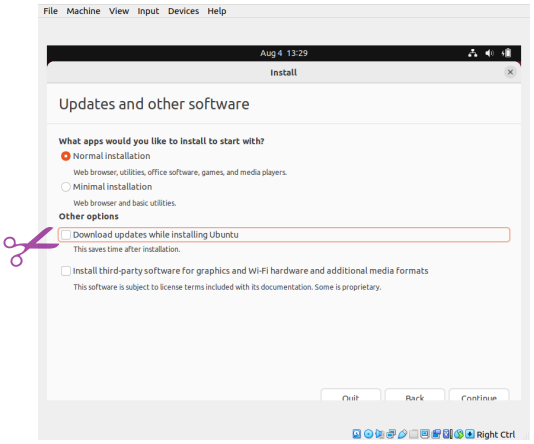


Figure F.2

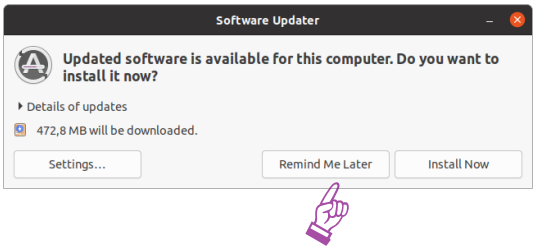


Figure F.3

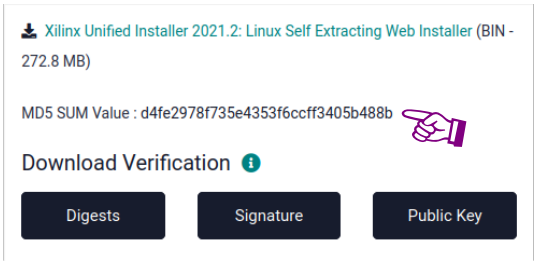


Figure F.4

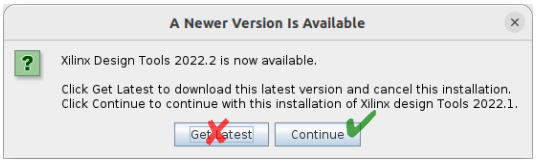


Figure F.5

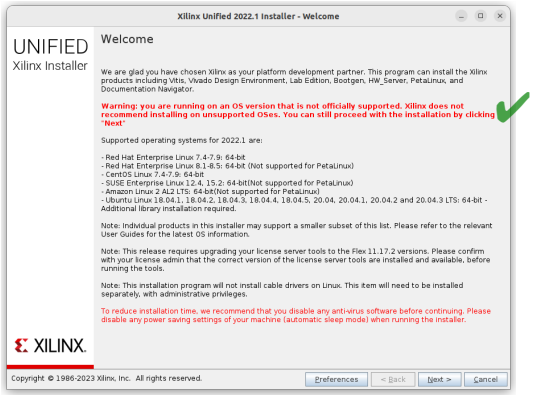


Figure F.6

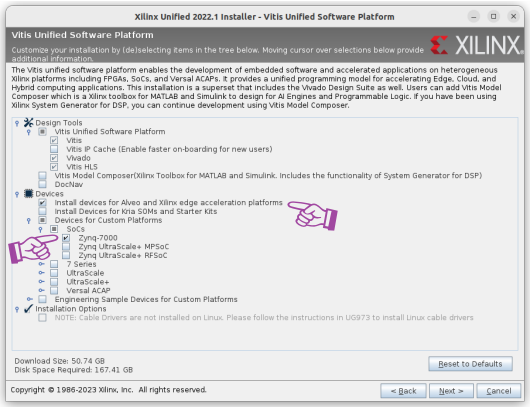


Figure F.7

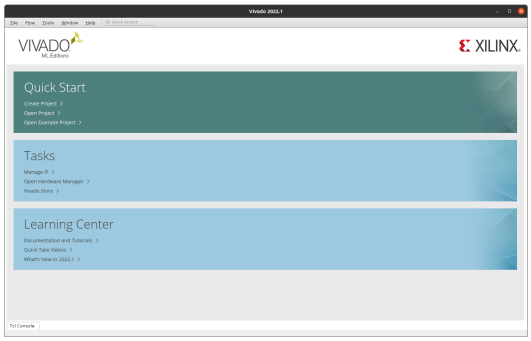


Figure F.8

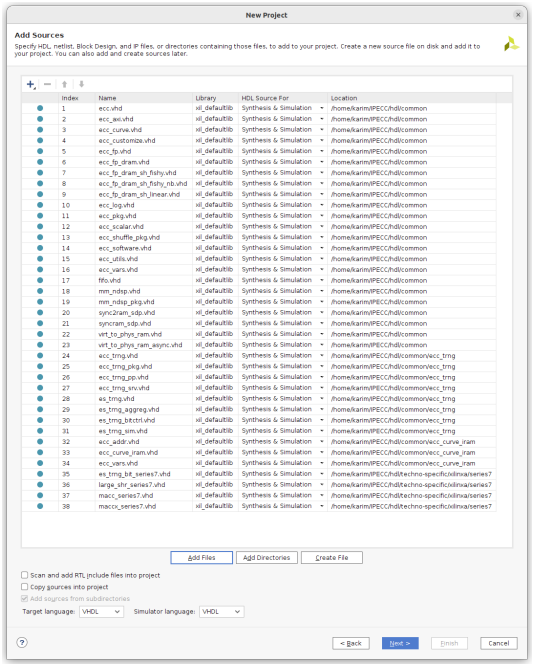


Figure F.9

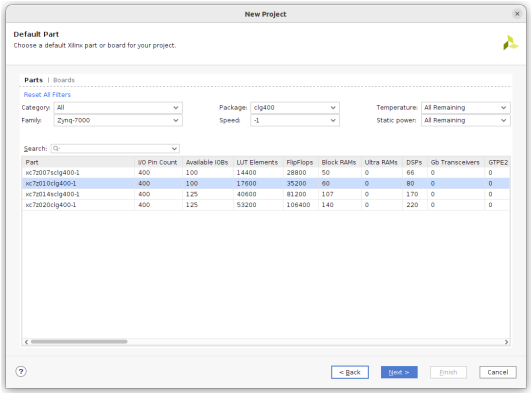


Figure F.10

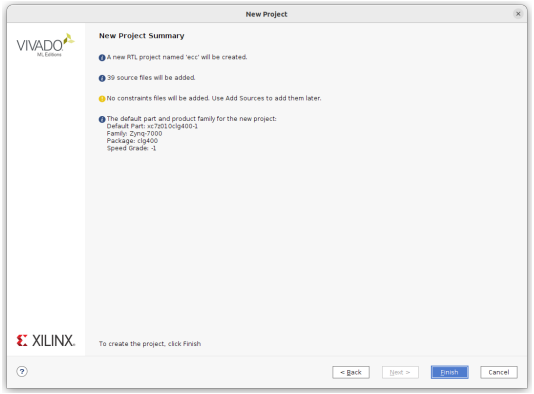


Figure F.11

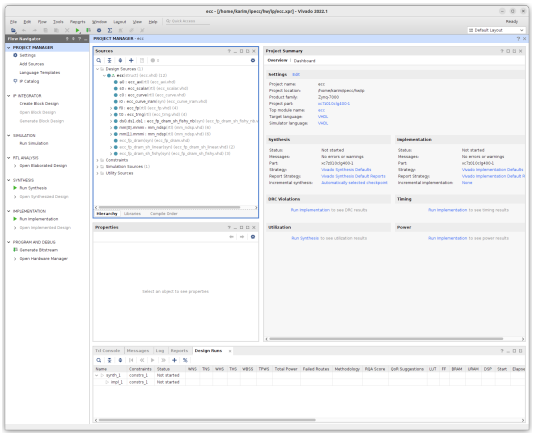


Figure F.12

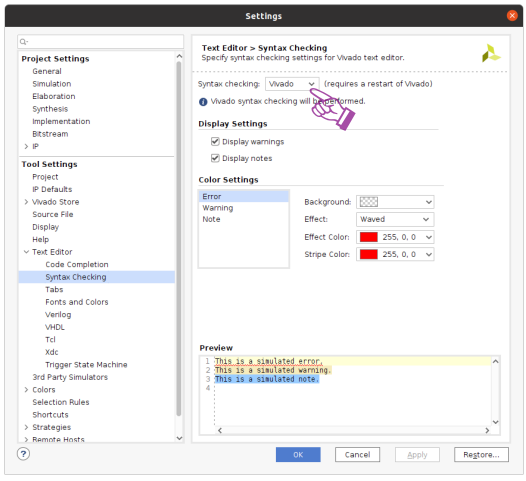


Figure F.13

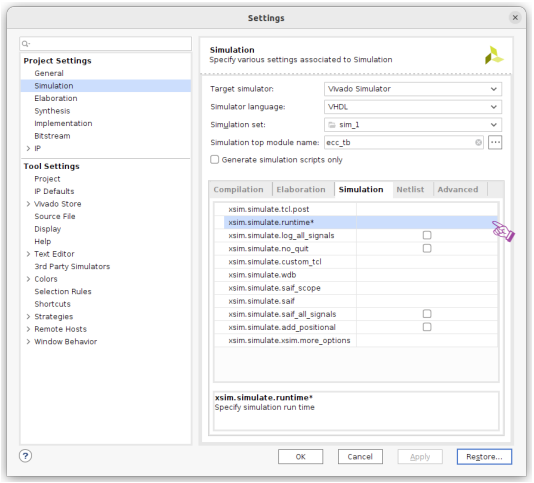


Figure F.14

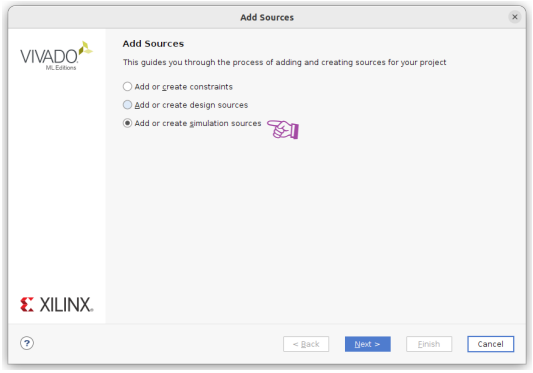


Figure F.15

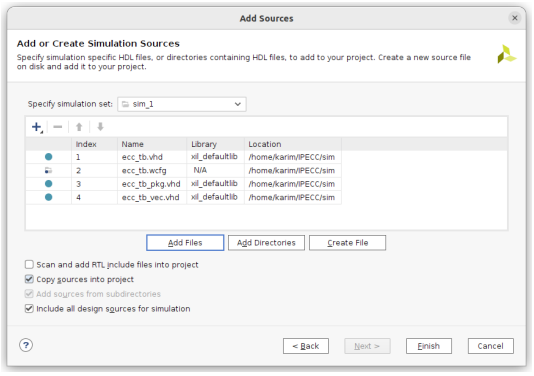


Figure F.16

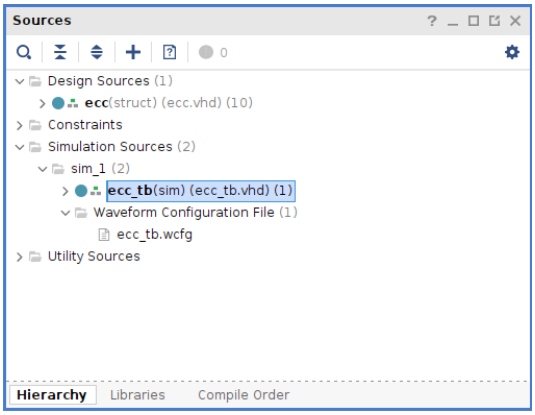


Figure F.17

```
21  -- *********************************
22  -- Start of: user-editable parameters
23  -- *********************************
24  -- Please refer to the in-file documentation of parameters below (after the
25  -- package specification), where parameters are described in the same order
26  -- as they appear in the code hereafter.
27  constant nn : positive := 256;
28  constant nn_dynamic : boolean := TRUE;
29  type techno_type is (spartan6, virtex6, series7, ultrascale, ialtera, asic);
30  constant techno : techno_type := series7; -- set a 'techno_type' value
31  -- -------------------------------
32  -- Performance related parameters
33  -- -------------------------------
34  -- multwidth is only used if 'techno' = 'asic'
35  -- (otherwise its value has no meaning and can be ignored)
36  constant multwidth : positive := 32; -- 32 seems fair for an ASIC default
37  constant nbmult : positive range 1 to 2 := 2;
38  -- parameter nbdsp below is range-constrained because it must be >=2
39  constant nbdsp : positive range 2 to positive'high := 6;
40  constant sramlat : positive range 1 to 2 := 1;
41  constant async : boolean := TRUE;
42  -- ------------------------------------------------
43  -- Side-channel countermeasures related parameters
44  -- ------------------------------------------------
45  constant debug : boolean := FALSE; -- FALSE = highly secure, TRUE = highly not
46  constant blinding : integer := 0;  -- 96 seems fair for size of blinding rnd
47  constant shuffle : boolean := TRUE; -- memory shuffling
48  type shuftype is (none, linear, permute_lgnb, permute_limbs);
49  constant shuffle_type : shuftype := permute_lgnb; -- set a 'shuftype' value
50  constant zremask : integer := 4; -- quite arbitrary
51  -- ------------------------
52  -- TRNG related parameters
53  -- ------------------------
54  constant notrng : boolean := TRUE; -- set to TRUE for simu, to FALSE for syn
55  constant nbtrng : positive := 4;
56  constant trngta : natural range 1 to 4095 := 32;
57  constant trng_ramsz_raw : positive := 4; -- in kB
58  constant trng_ramsz_axi : positive := 4; -- in kB
59  constant trng_ramsz_fpr : positive := 4; -- in kB
60  constant trng_ramsz_crv : positive := 4; -- in kB
61  constant trng_ramsz_shf : positive := 16; -- in kB
62  -- -------------
63  -- Miscellaneous
64  -- -------------
65  constant axi32or64 : natural := 32; -- 32 or 64 only allowed values
66  constant nblargenb : positive := 32;  -- change these two parameters only if
67  constant nbopcodes : positive := 512; -- you really know what you're doing
68  -- ---------------------------
69  -- Simulation-only parameters
70  -- ---------------------------
71  constant simvecfile : string := "/tmp/ecc_vec_in.txt"
72  constant simkb : natural range 0 to natural'high := 0; -- if 0 then ignored
73  constant simlogfile : string := "/tmp/ecc.log";
74  constant simtrngfile : string := "/tmp/random.txt";
75  -- *********************************
76  -- End of: user-editable parameters
77  -- *********************************
```

Figure F.18

```
1 == NEW CURVE #0
2 nn=21
3 p=0x1ce54b
4 a=0x0ec20f
5 b=0x1bb973
6 q=0x1ce256
7 == TEST [k]P #0.0
8 Px=0x00851a
9 Py=0x0a0e0f
10 k=0x1c0ac1
11 nbbld=14
12 kPx=0x0acc93
13 kPy=0x0e007f
14 == TEST P+Q #0.1
15 Px=0x0e58a7
16 Py=0x0b0eb7
17 Qx=0x07136d
18 Qy=0x032a06
19 PplusQx=0x0c11ee
20 PplusQy=0x07c882
21 == TEST [2]P #0.2
22 Px=0x0adc61
23 Py=0x14fae0
24 twoPx=0x1462de
25 twoPy=0x0cfb3a
26 == TEST -P #0.3
27 Px=0x0a3f63
28 Py=0x16043d
29 negPx=0x0a3f63
30 negPy=0x06e10e
31 == TEST isPoncurve #0.4
32 Px=0x041730
33 Py=0x19233e
34 false
35 == TEST isP==Q #0.5
36 Px=0x01abef
37 Py=0x040c8d
38 Qx=0x0e2427
39 Qy=0x037216
40 false
41 == TEST isP==-Q #0.6
42 Px=0x123a44
43 Py=0x18f264
44 Qx=0x123a44
45 Qy=0x03f2e7
46 true
```

Figure F.19

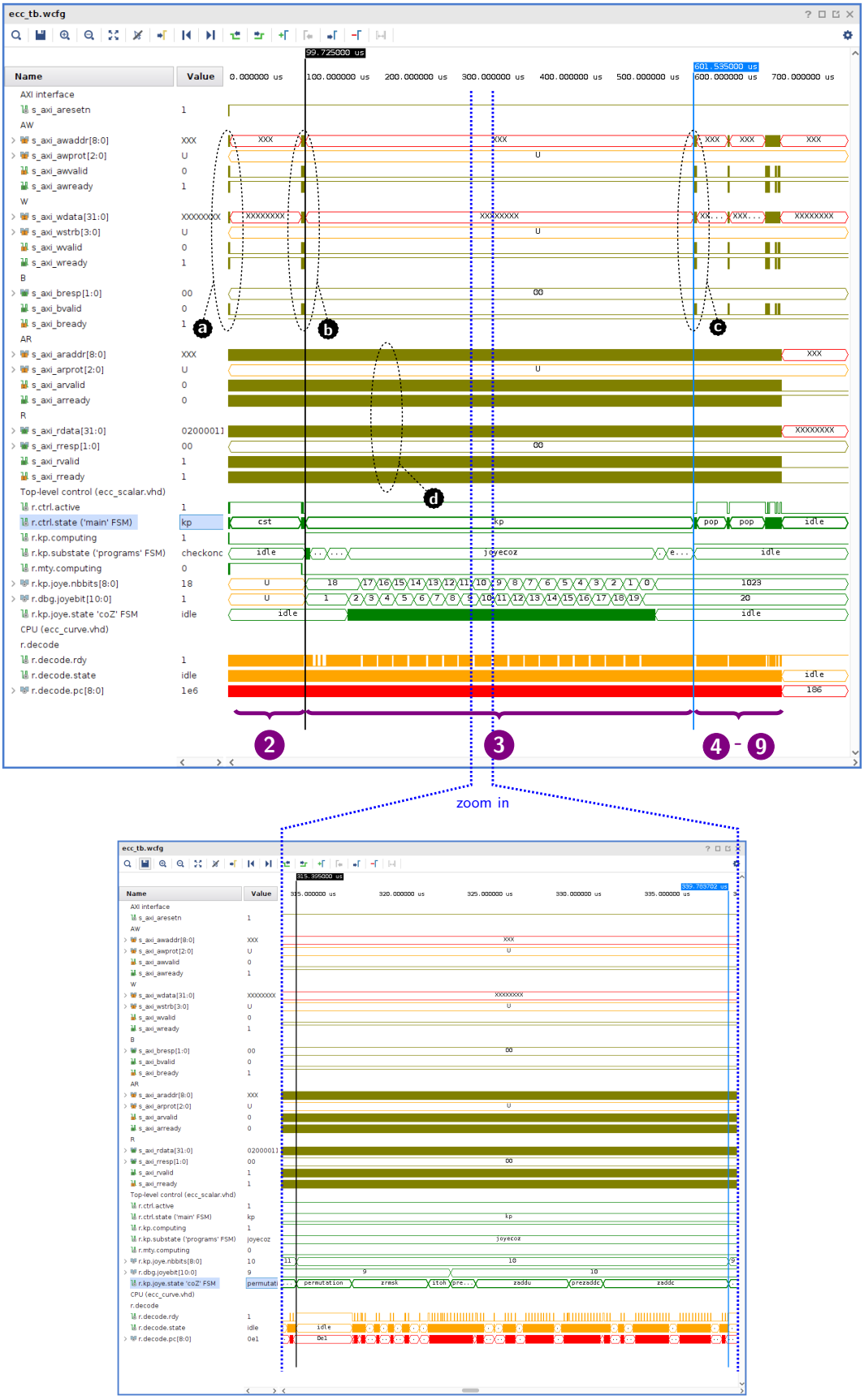


Figure F.20: Simulation waveform of our input test-vectors file. Lower half of the figure shows a detailed view of the processing of one bit of the scalar (the bit of weight 10).

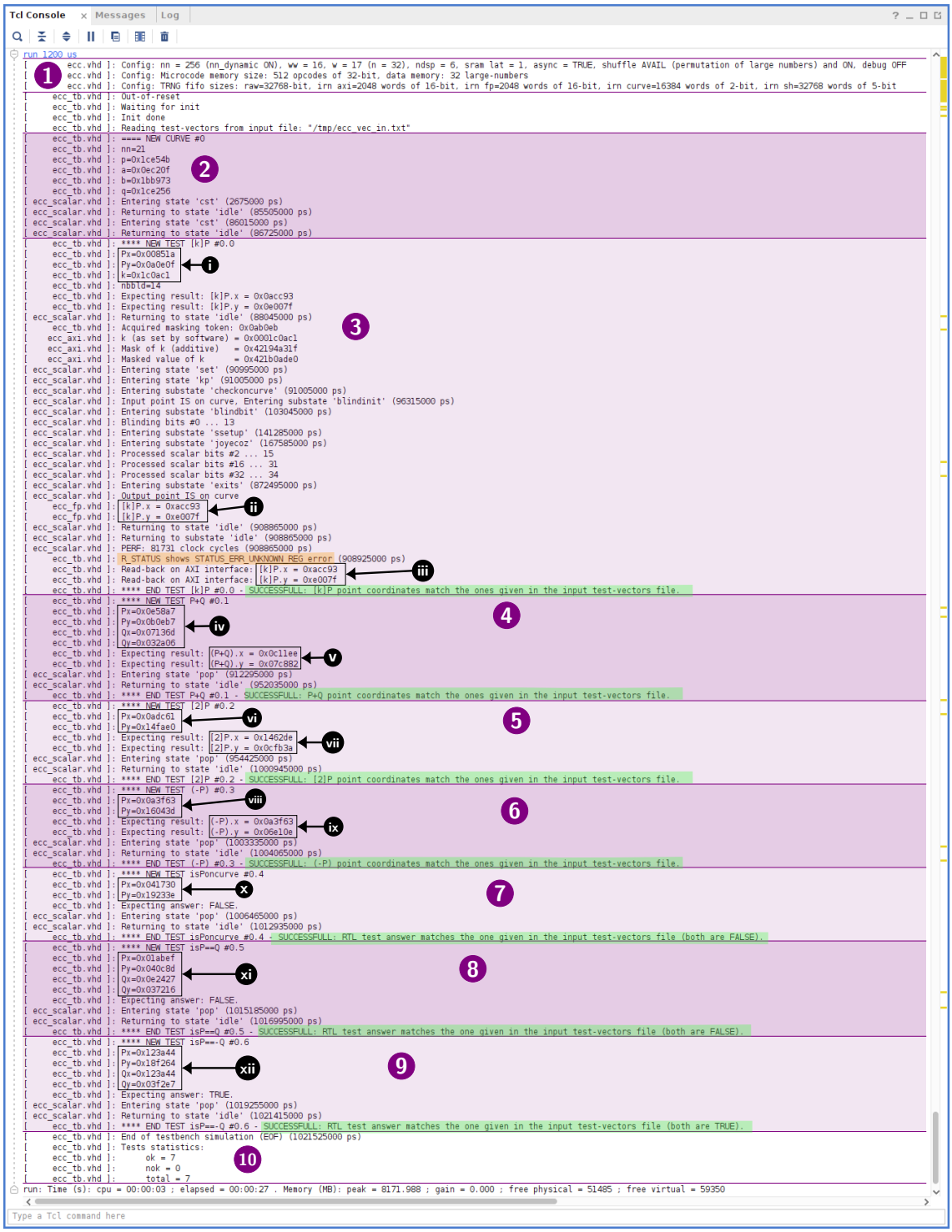


Figure F.21: Simulation console log of our input test-vectors file.

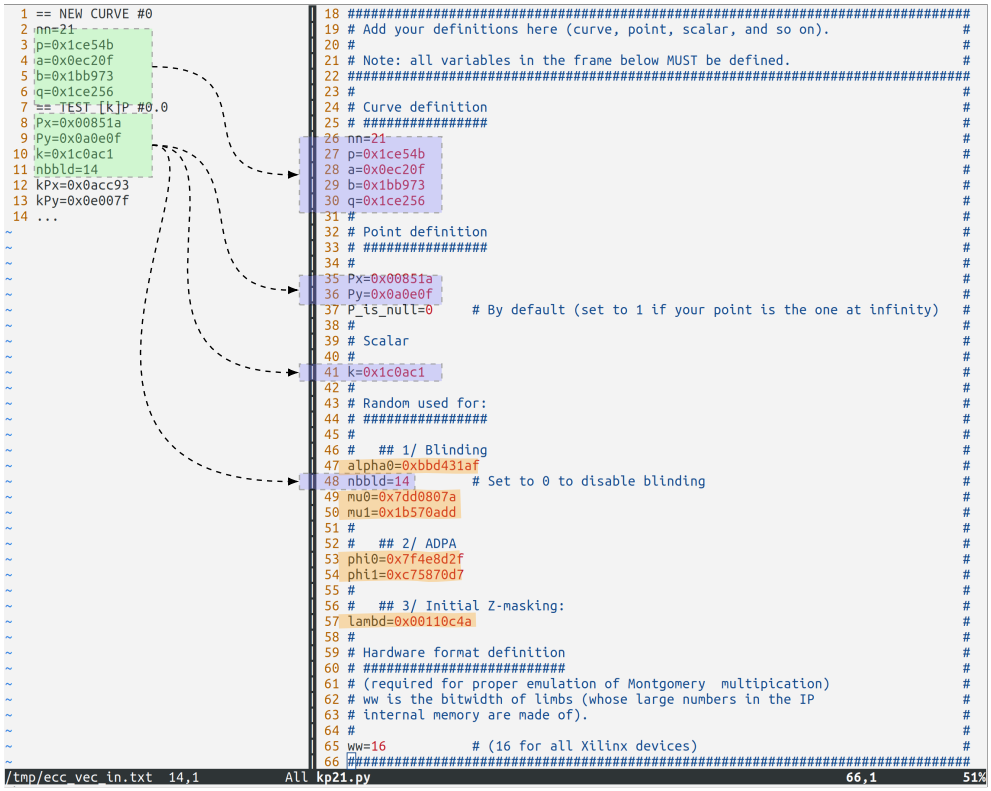


Figure F.22: Copy-paste cryptographic values for curve and points from the test-vector file (in green) into the Python script (in blue). Remaining parameters (in orange) are random values we can quickly extract from the RTL log file (`/tmp/ecc.log`) as seen on fig-F.23 just below.

From your Linux shell prompt

```
$ grep -A 3 --color "\.random_.*L" /tmp/ecc.log
.random_alphaL [0x031]
[0x031]    NNRND 0xbbd431af  (12 <-  random  )  [96705000 ps]
[0x032]    NNADD 0x001ce256  (08 <- 03  +  31)  [96875000 ps]
[0x033]    NNADD 0x00000000  (09 <- 31  +  31)  [97045000 ps]
--
.random_muL [0x041]
[0x041]    NNRND 0x7dd0807a  (26 <-  random  )  [117645000 ps]
[0x042]    TESTPAR           (26 is   even   )  [117755000 ps]
[0x043]    NNRND 0x1b570add  (27 <-  random  )  [117885000 ps]
--
.random_phiL [0x047]
[0x047]    NNRND 0x7f4e8d2f  (10 <-  random  )  [118585000 ps]
[0x048]    NNRND 0xc75870d7  (11 <-  random  )  [118715000 ps]
[0x049]    TESTPAR           (04 is    odd   )  [118825000 ps]
--
.random_lambdaL [0x071]
[0x071]    NNRND 0x00110c4a  (21 <-  random  )  [141345000 ps]
[0x072]    NNSUB 0xfff426ff  (22 <- 21  -  00)  [141515000 ps]
[0x073]    NNADD 0x00110c4a  (21 <- 22  +  00)  [141685000 ps]
```

Figure F.23: Grepping the regexp `"\.random_.*L"` in the RTL simulation log file allows you to quickly retrieve the values of random variables drawn by the IP using the `NNRND` instruction during a $[k]\mathcal{P}$ execution. The lines highlighted in orange match the highlighted ones in fig-F.22 just above.

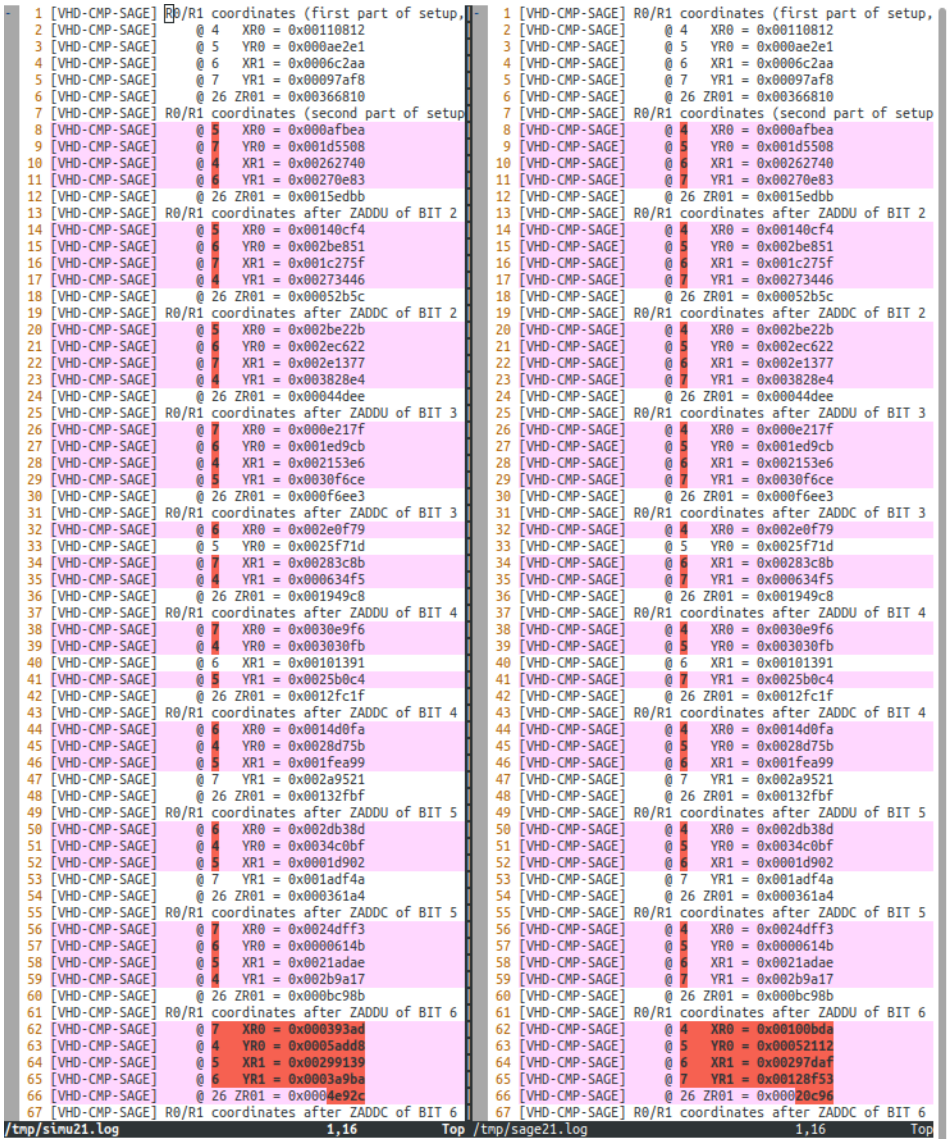


Figure F.24: Differences between the RTL simu log (on the left) and the *Sage* script (on the right) for $[k]\mathcal{P}$ operation. Coordinates X,Y,Z of $\mathcal{R}_0$ and $\mathcal{R}_1$ are identical from the start of computation until the bit of weight 6 of the scalar is hit (the «Z-remask» countermeasure starts to take effect) Furthermore, from the begining of computation the physical addresses of the four coords. differ: they are fixed and constant on the right while they are randomized on the left (effect of the «XY-shuffling» countermeasure).

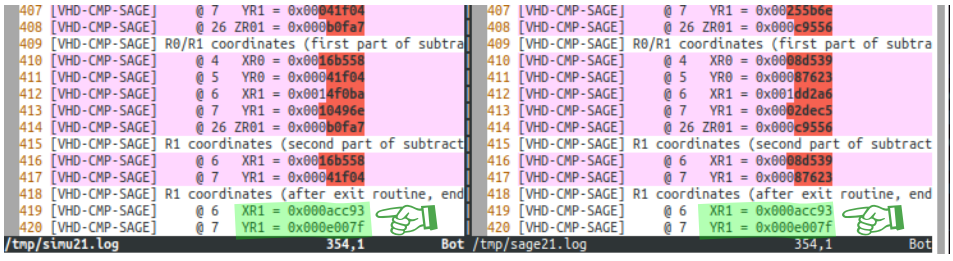


Figure F.25: Despite the randomizations taking place all along the computation (see fig-F.24 just above) the final coordinates of $\mathcal{R}_1 = [k]\mathcal{P}$ naturally match.

```
42  -- ------------------------------------------------
43  -- Side-channel countermeasures related parameters
44  -- ------------------------------------------------
45  constant debug : boolean := TRUE;
46  constant blinding : integer := 0;
47  constant shuffle : boolean := TRUE;
```

<ecc_customize.vhd>

Figure F.26: Editing file ⟨ecc_customize.vhd⟩ to switch the IP into debug (unsecure) mode.

```
744         -- Choice of the TRNG random source.
745         debug_trng_use_real(s_axi_aclk, axi0, axo0);
746
747         -- Enable the post-processing unit from reading raw random bytes.
748         debug_trng_pp_start_pulling_raw(s_axi_aclk, axi0, axo0);
749
750         -- Disable Z-remasking
751         configure_zremasking(s_axi_aclk, axi0, axo0, FALSE, 0);
752
753         -- End of IP initialization & config
754
755         -- ------------------------------------------------------------------
756         -- Main infinite loop, getting lines from input file 'simvecfile'
```

<ecc_tb.vhd>

Figure F.27: Adding these two lines to ⟨ecc_tb.vhd⟩ will have the testbench disable the «Z-remask» countermeasure by accessing the W_ZREMASK register, just as a software driver would do at runtime. See also fig-F.28 below.

```
744         -- Choice of the TRNG random source.
745         debug_trng_use_real(s_axi_aclk, axi0, axo0);
746
747         -- Enable the post-processing unit from reading raw random bytes.
748         debug_trng_pp_start_pulling_raw(s_axi_aclk, axi0, axo0);
749
750         -- Disable Z-remasking
751         configure_zremasking(s_axi_aclk, axi0, axo0, FALSE, 0);
752
753         -- Disable XY-shuffling
754         debug_disable_xyshuf(s_axi_aclk, axi0, axo0);
755
756         -- End of IP initialization & config
757
758         -- ------------------------------------------------------------------
759         -- Main infinite loop, getting lines from input file 'simvecfile'
```

<ecc_tb.vhd>

Figure F.28: Adding these two lines to ⟨ecc_tb.vhd⟩ will have the testbench disable the «XY-shuffling» countermeasure by accessing the W_DBG_CFG_XYSHUF register. This can only be done with an IP configured in debug (unsecure) mode, or the IP will triggger an STATUS_ERR_UNKNOWN_REG error.

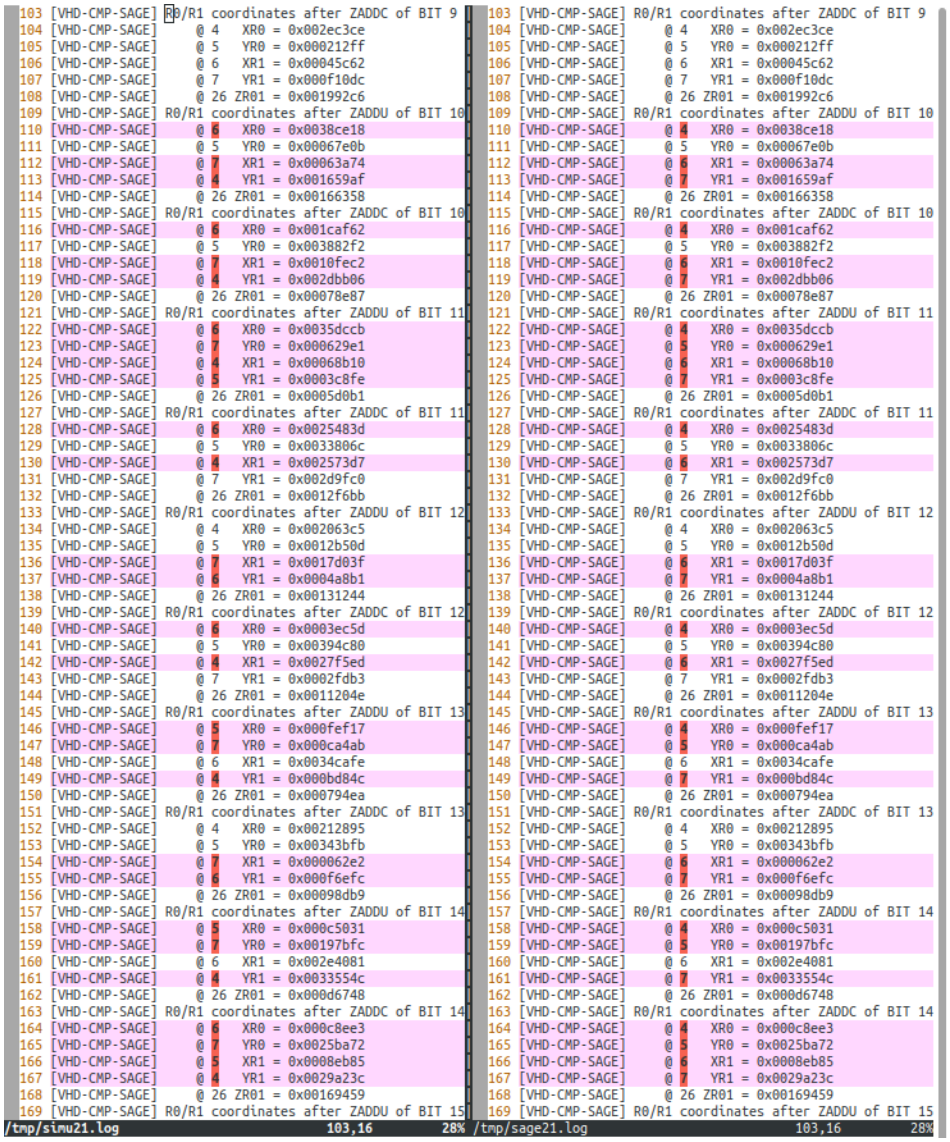


Figure F.29: Differences between the RTL simu log (on the left) and the *Sage* script (on the right) for $[k]\mathcal{P}$ operation after the «Z-remask» countermeasure has been disabled. The coordinates match all along between the two logs, only physical addresses of coordinates of points $\mathcal{R}_0$ and $\mathcal{R}_1$ stay randomized during the RTL simulation.

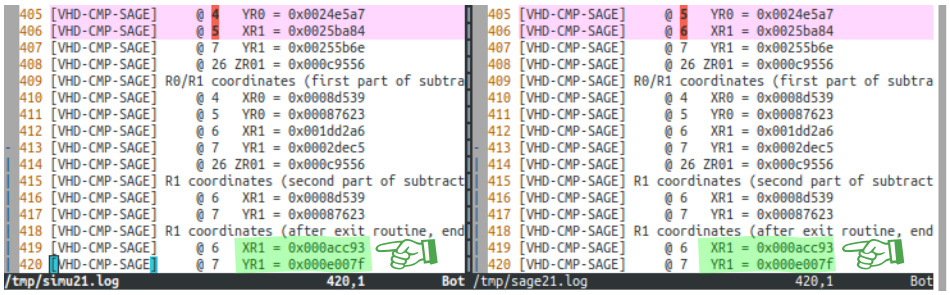


Figure F.30: Once the scalar parsing loop is complete and before entering conditional subtraction of the base point $\mathcal{P}$ the physical addresses of coordinates of $\mathcal{R}_0$ and $\mathcal{R}_1$ and restored back to their static deterministic values. For explanation on the final conditional subtraction at the end of $[k]\mathcal{P}$ computation, please refer to §4.1.2, in particular algorithm 9 and its related discussion.

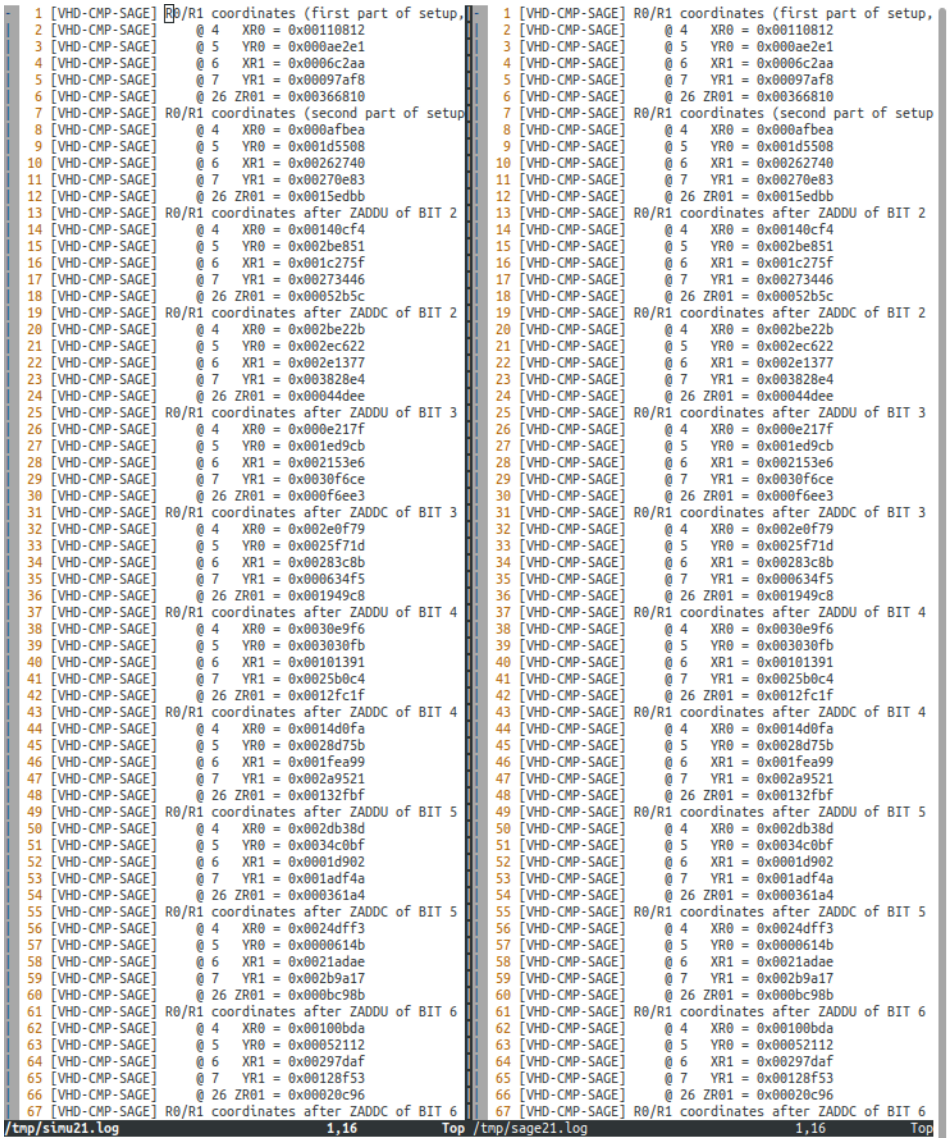


Figure F.31: Once both countermeasures «Z-remask» and «XY-shuffling» have been removed, content of RTL simu log (on the left) and *Sage* script log (on the right) fully match.

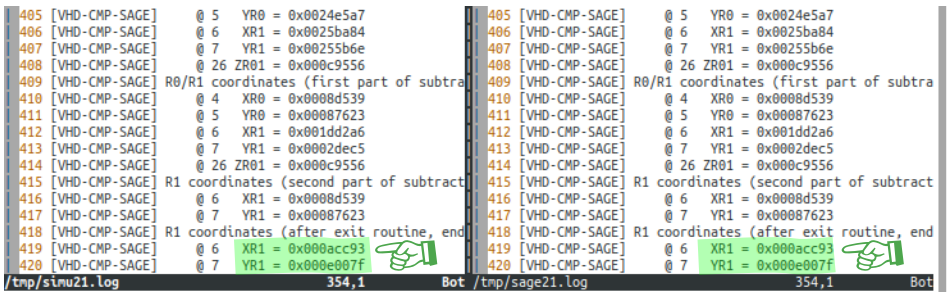


Figure F.32

```
51          -- ------------------------
52          -- TRNG related parameters
53          -- ------------------------
54          constant notrng : boolean := FALSE;
55          constant nbtrng : positive := 4;
56          constant trngta : natural range 1 to 4095 := 32;
```
<ecc_customize.vhd>

Figure F.33: Setting `notrng = FALSE` in ⟨`ecc_customize.vhd`⟩ will force the structural VHDL of top-level TRNG entity ⟨`ecc_trng.vhd`⟩ to instanciate the synthesizable model of the TRNG rather than the non-synthesizable one that's reading randomness from a testbench input file. See also fig-F.34 just below.

```
201 t0: if notrng = FALSE generate
202         -- 'ES-TRNG': real physical entropy source described structurally with
203         -- Xilinx low-level primitives (LUTs & DFFs) and combinational loops.
204         t0: es_trng
205                 port map(
206                         clk => clk,
207                         rstn => rstn,
208                         swrst => swrst,
209                         data_t => data_t,
210                         valid_t => valid_t,
211                         rdy_t => rdy_t,
212                         dbgtrngta => dbgtrngta,
213                         dbgtrngrawreset => dbgtrngrawreset,
214                         dbgtrngrawfull => dbgtrngrawfull,
215                         dbgtrngrawwaddr => dbgtrngrawwaddr,
216                         dbgtrngrawraddr => dbgtrngrawraddr,
217                         dbgtrngrawdata => dbgtrngrawdata,
218                         dbgtrngrawfiforeaddis => dbgtrngrawfiforeaddis,
219                         dbgtrngrawduration => dbgtrngrawduration,
220                         dbgtrngvonneuman => dbgtrngvonneuman,
221                         dbgtrngidletime => dbgtrngidletime
222                 );
223 end generate;
224
225 -- pragma translate_off
226 t1: if notrng = TRUE generate
227         -- 'es_trng_sim' reads randomness from local file
228         -- and provides them to 'ecc_trng_pp'
229         t0: es_trng_sim
230                 port map(
231                         clk => clk,
232                         rstn => rstn,
233                         swrst => swrst,
234                         data_t => data_t,
235                         valid_t => valid_t,
236                         rdy_t => rdy_t,
237                         dbgtrngta => dbgtrngta,
238                         dbgtrngrawreset => dbgtrngrawreset,
239                         dbgtrngrawfull => dbgtrngrawfull,
240                         dbgtrngrawwaddr => dbgtrngrawwaddr,
241                         dbgtrngrawraddr => dbgtrngrawraddr,
242                         dbgtrngrawdata => dbgtrngrawdata,
243                         dbgtrngrawfiforeaddis => dbgtrngrawfiforeaddis,
244                         dbgtrngrawduration => dbgtrngrawduration,
245                         dbgtrngvonneuman => dbgtrngvonneuman,
246                         dbgtrngidletime => dbgtrngidletime
247                 );
248 end generate;
249 -- pragma translate_on
```
<ecc_trng/ecc_trng.vhd>

Figure F.34: VHDL component `es_trng.vhd` corresponds to the true physical random generator used in IPECC. It is named after the design ES–TRNG ([YRG+18]) originally published by KU-Leuven.

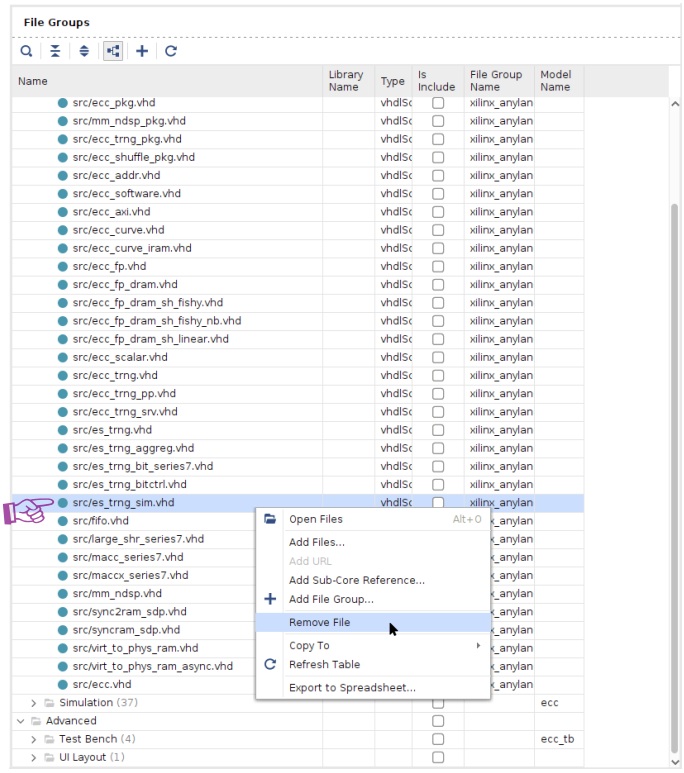


Figure F.35: Exported files for the packaging of the IP must not include the simulation model `ecc_trng_sim.vhd`.

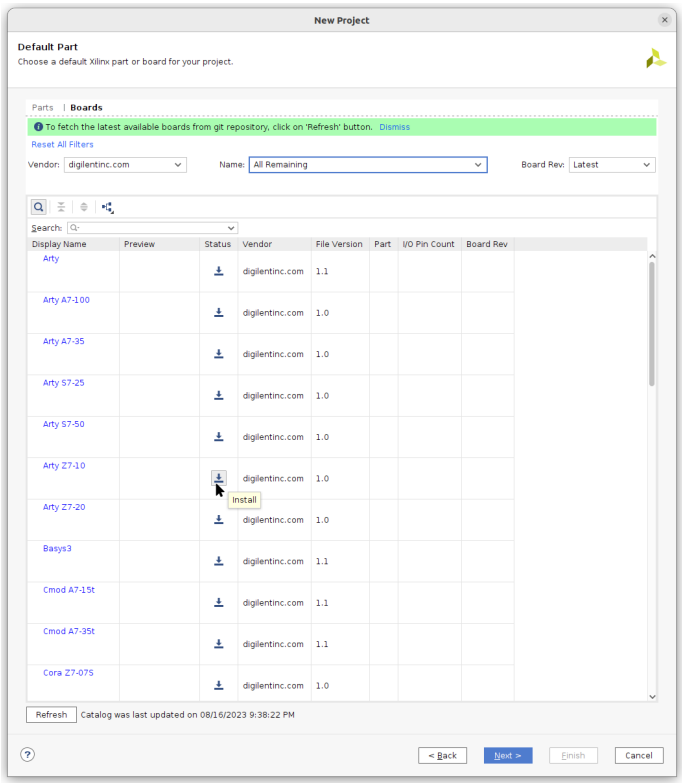


Figure F.36

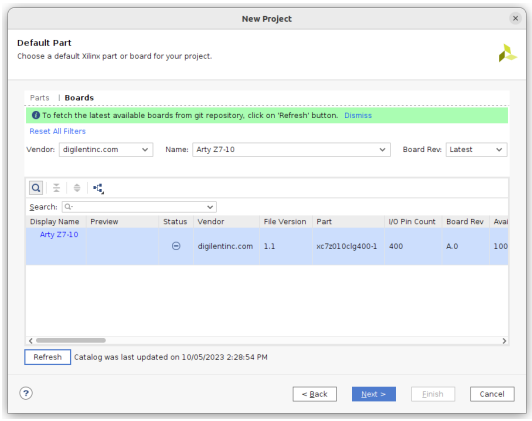


Figure F.37: Make sure the board Arty Z7 10 is selected! (its line must be highlighted)

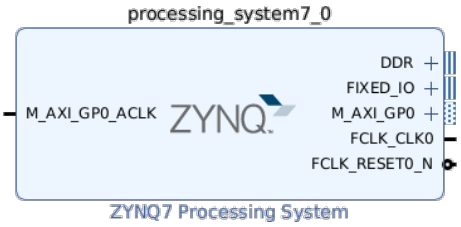


Figure F.38

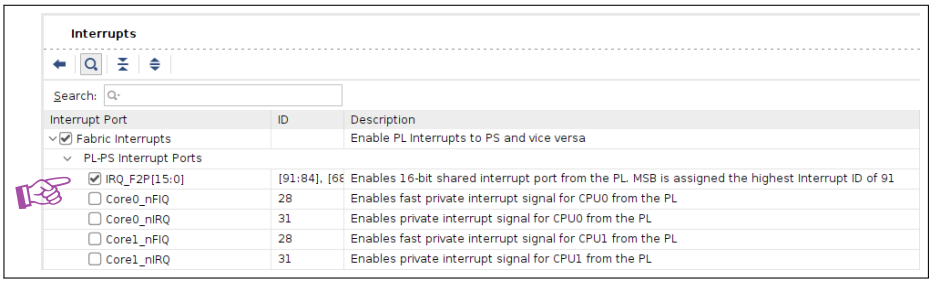


Figure F.39

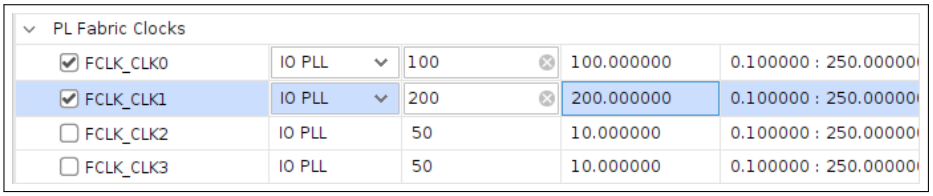


Figure F.40

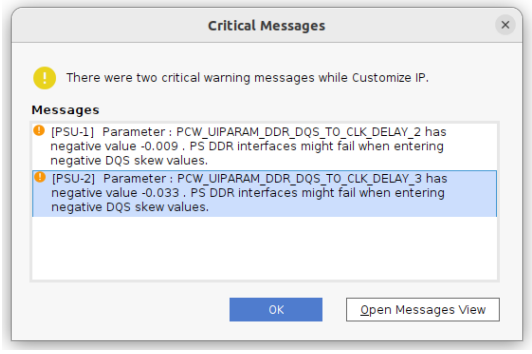


Figure F.41

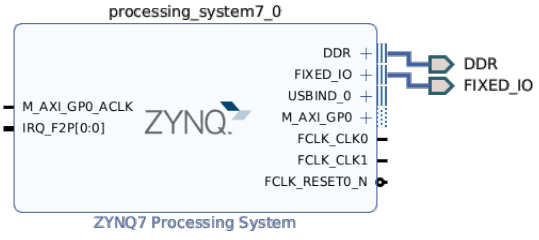


Figure F.42

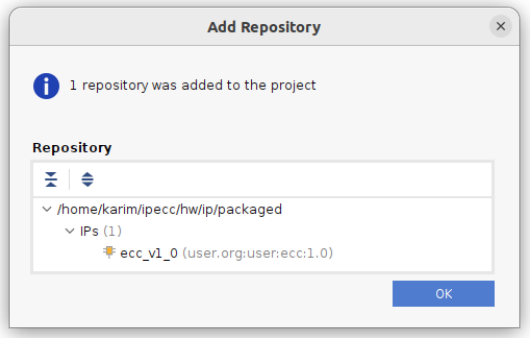


Figure F.43

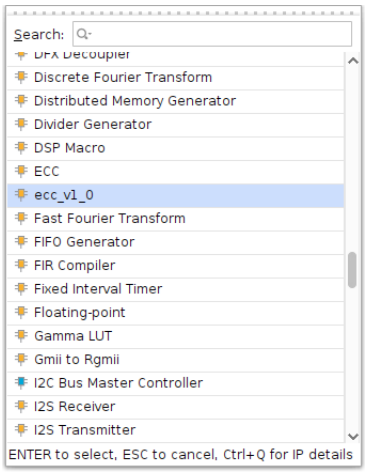


Figure F.44

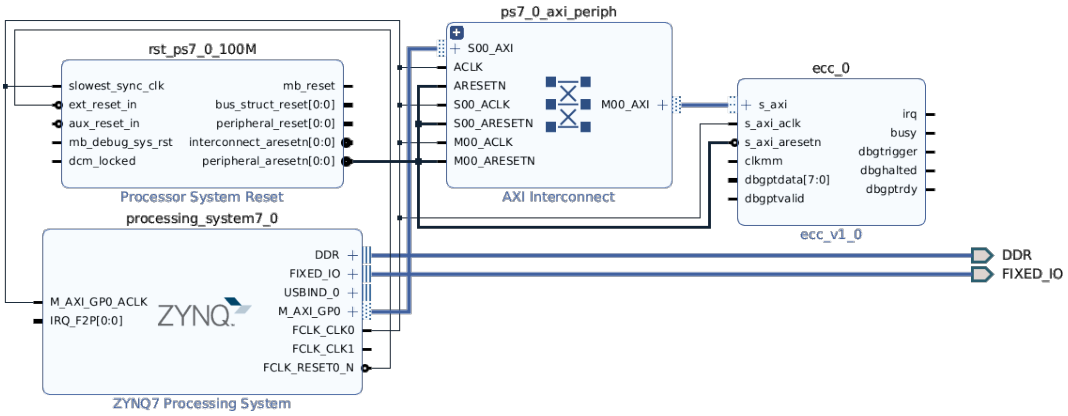


Figure F.45

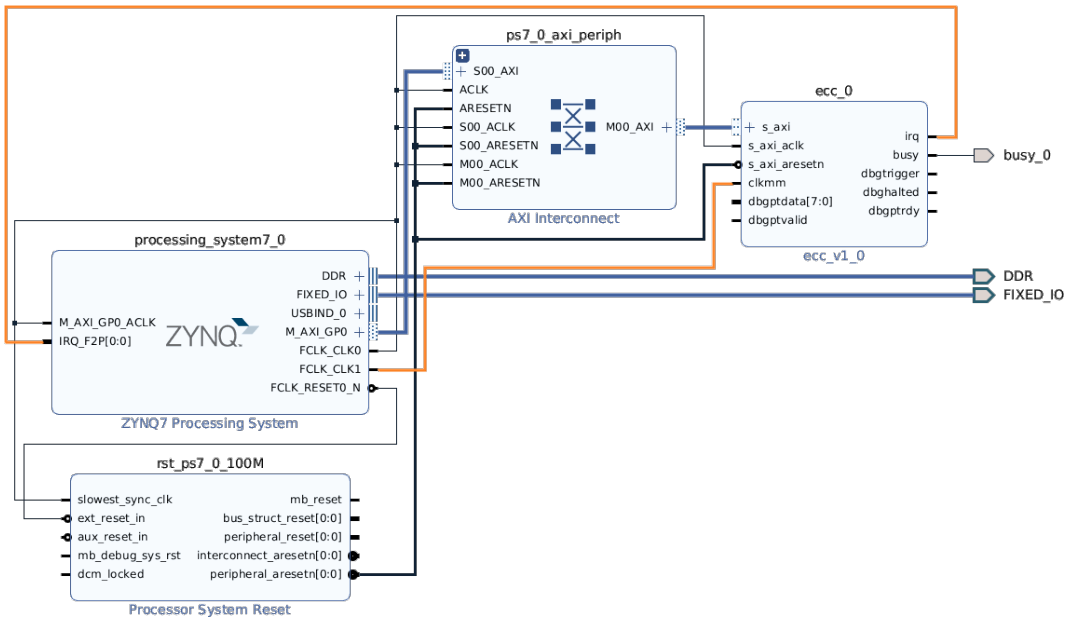


Figure F.46

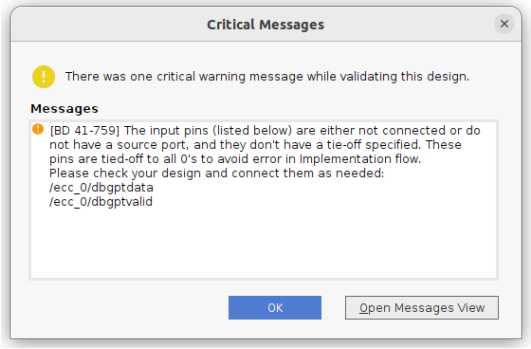


Figure F.47

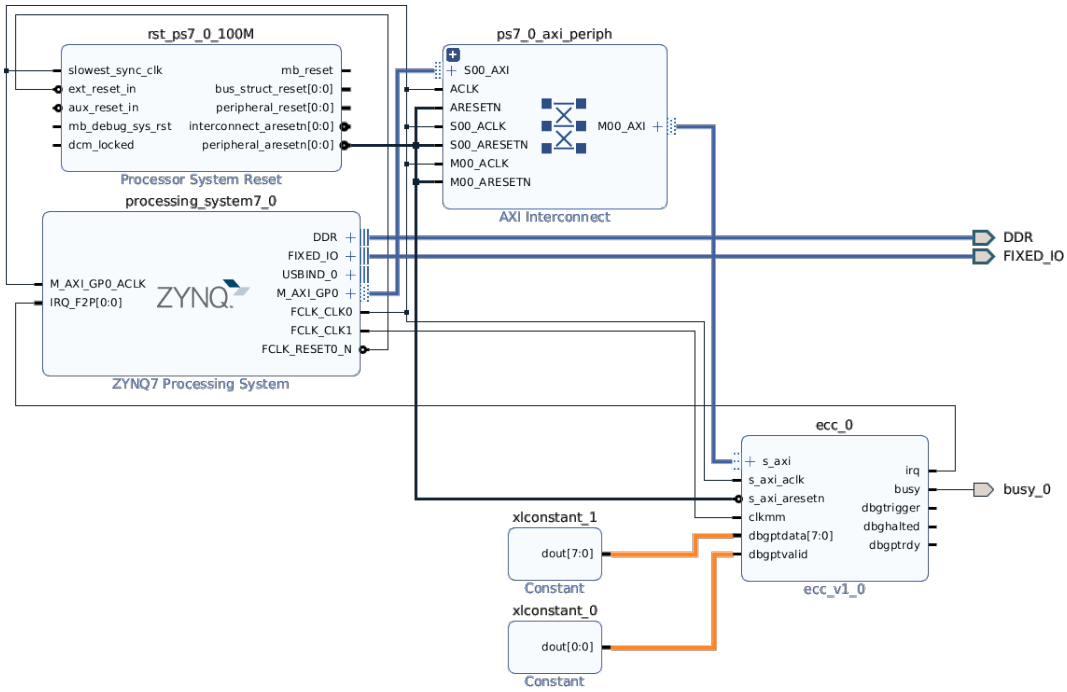


Figure F.48

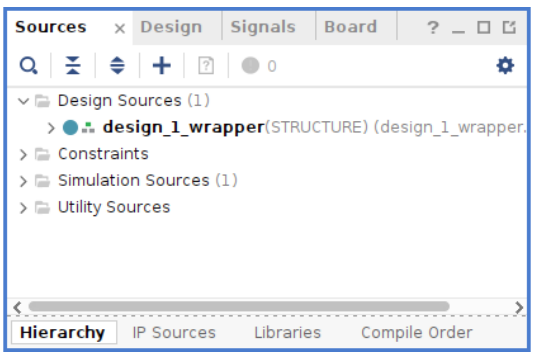


Figure F.49

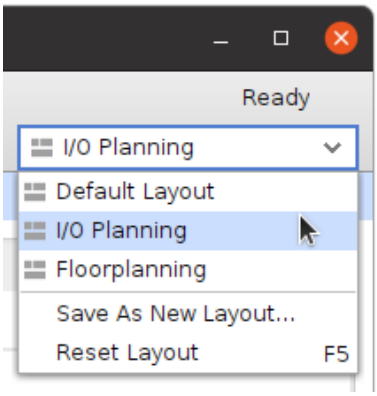


Figure F.50

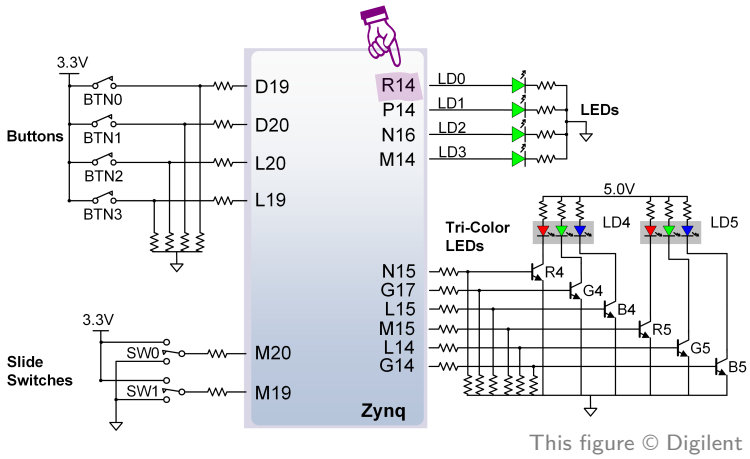


Figure F.51

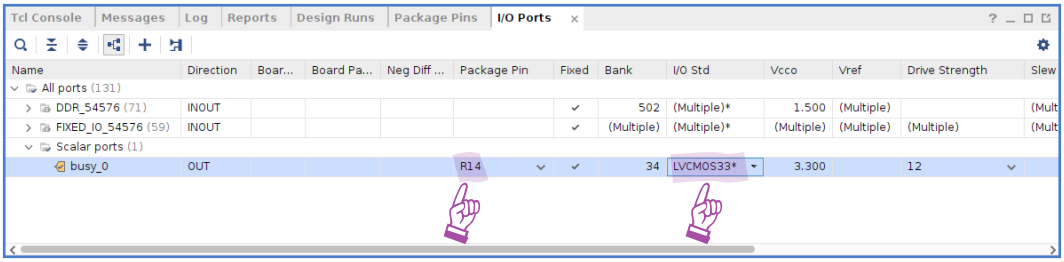


Figure F.52

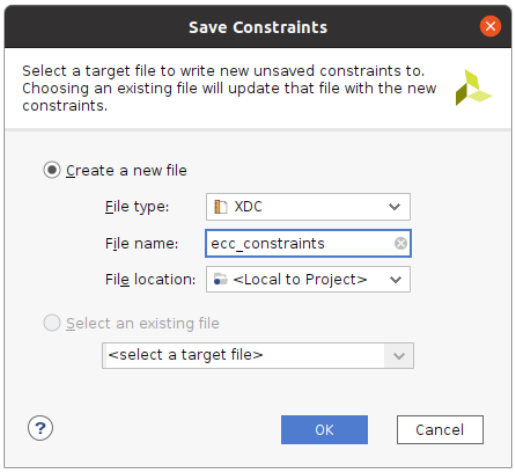


Figure F.53

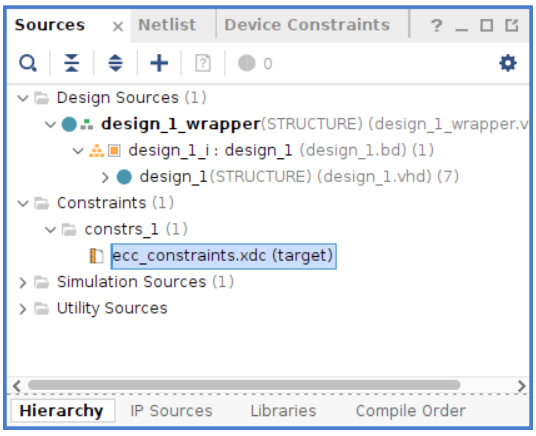


Figure F.54

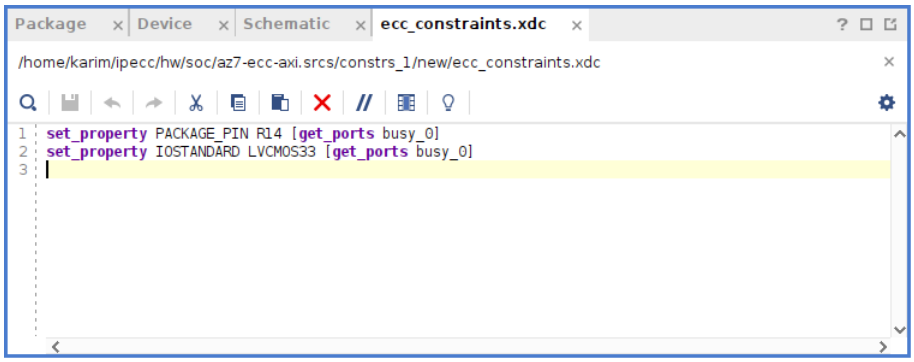


Figure F.55

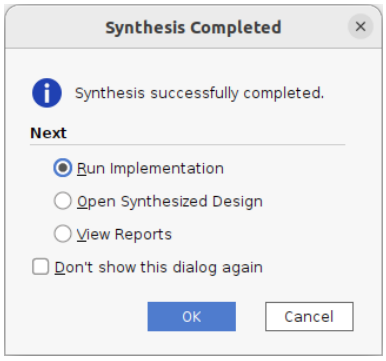


Figure F.56

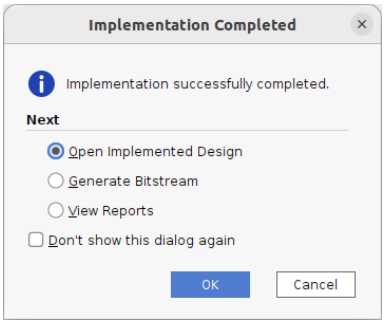


Figure F.57

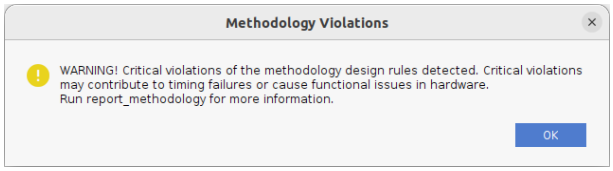


Figure F.58

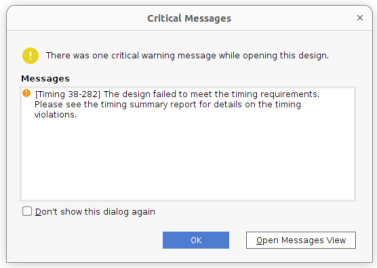


Figure F.59

```
+-----------+------------------+-------------------------------------------------+-----------+
| Rule      | Severity         | Description                                     | Violations
+-----------+------------------+-------------------------------------------------+-----------
| TIMING-6  | Critical Warning | No common primary clock between related clocks | 2
| TIMING-7  | Critical Warning | No common node between related clocks          | 2
| TIMING-17 | Critical Warning | Non-clocked sequential cell                    | 20
| TIMING-18 | Warning          | Missing input or output delay                  | 1
| TIMING-23 | Warning          | Combinational loop found                       | 8
| ULMTCS-1  | Warning          | Control Sets use limits recommend reduction    | 1
+-----------+------------------+-------------------------------------------------+-----------
```

Figure F.60: Vivado summary list of critical warnings, as yielded by a `report_methodology` command.

```
TIMING-17#1 Critical Warning
Non-clocked sequential cell
The clock pin design_1_i/ecc_0/U0/t0/t0.t0/bn.bg[0].b/t0/bx2/C is not reached by a timing clock
Related violations: <none>
```

Figure F.61

```
202   -- bit 'raw'
203   bx2: FDCE
204     generic map(
205       INIT => '0'
206     ) port map (
207       Q => raw_q,
208       C => ro2out, -- clock'd by RO2 output
209       CE => vcc,
210       CLR => ro1en_n,
211       D => raw_d
212     );                                    <es_trng_bit-series7.vhd>
```

Figure F.62

```
96    -- ------------------------
97    -- RO2 oscillator  1 x LUT2
98    -- ------------------------
99    ro2_0: LUT2
100     generic map(
101       INIT => x"4" -- O = I1.!I0
102     ) port map(
103       I0 => ro2s1,
104       I1 => ro2en,
105       O => ro2out
106     );                                    <es_trng_bit-series7.vhd>
```

Figure F.63

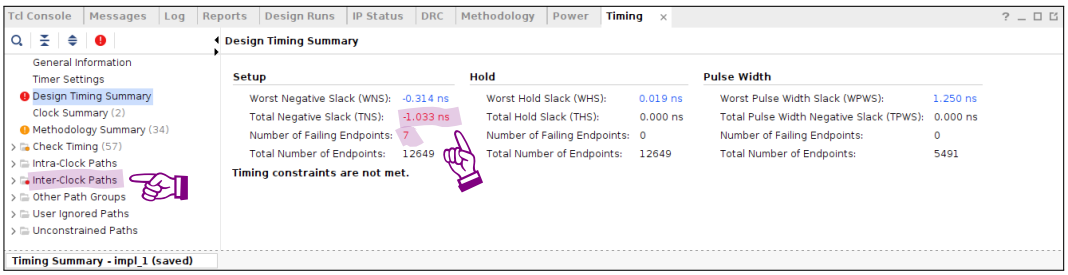


Figure F.64

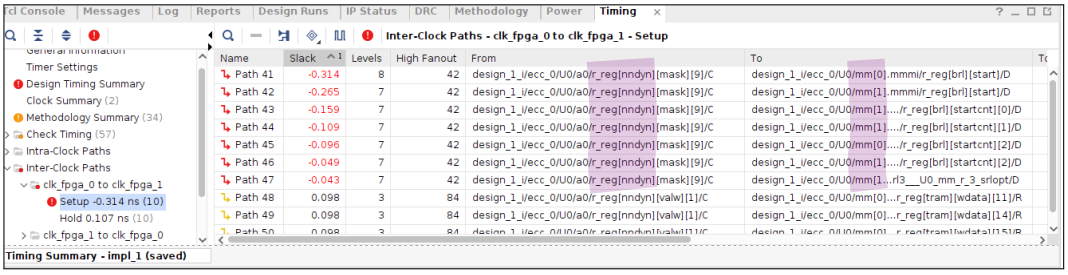


Figure F.65

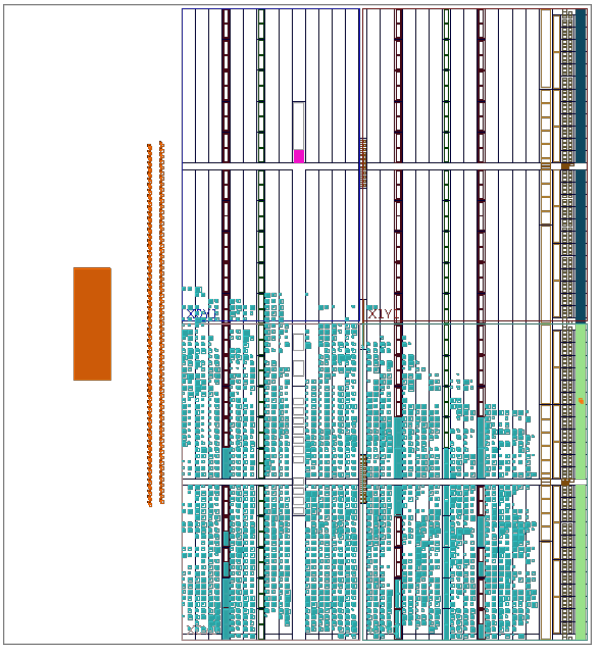


Figure F.66: Layout of the design. Cells colored in blue indicate occupied logic ressources.

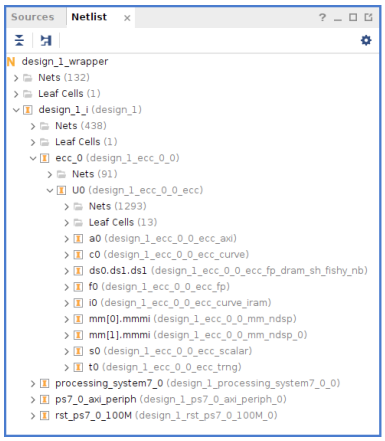


Figure F.67

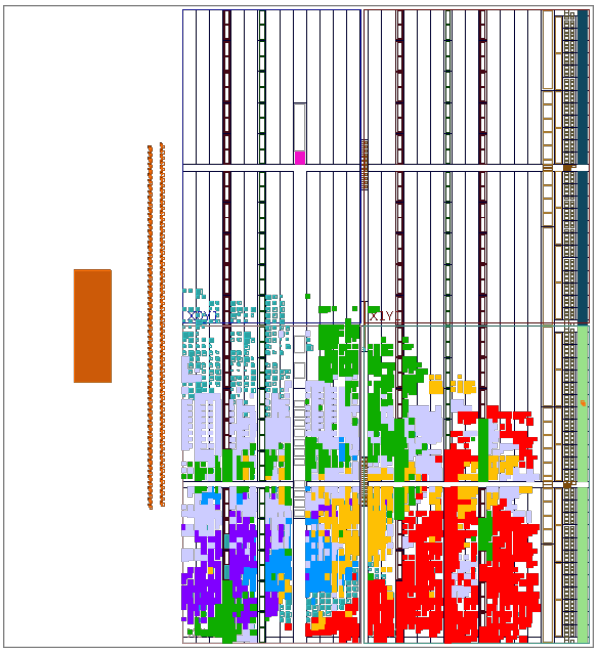


Figure F.68

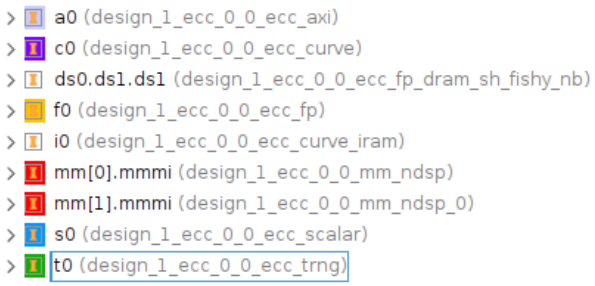


Figure F.69: Color legend for figure F.68.

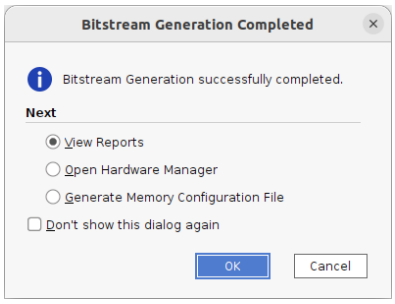


Figure F.70

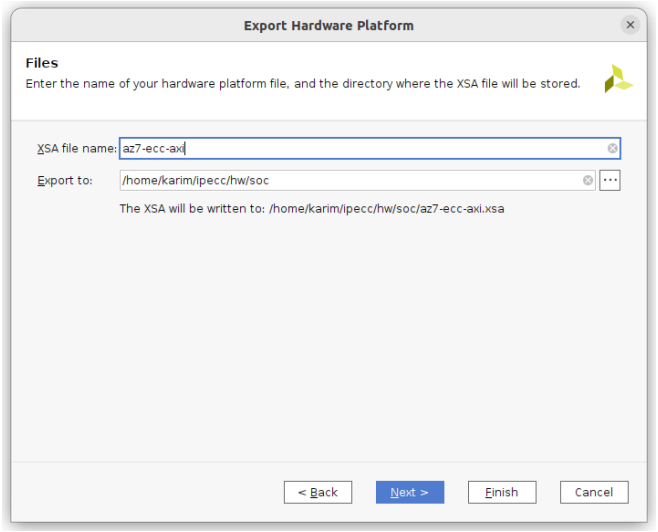


Figure F.71

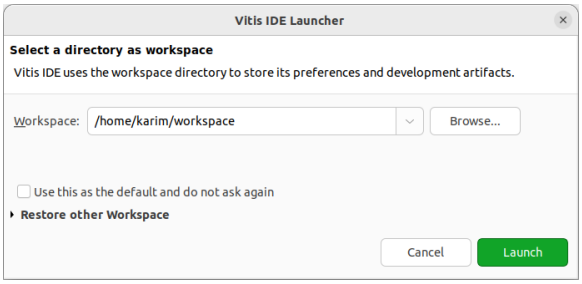


Figure F.72

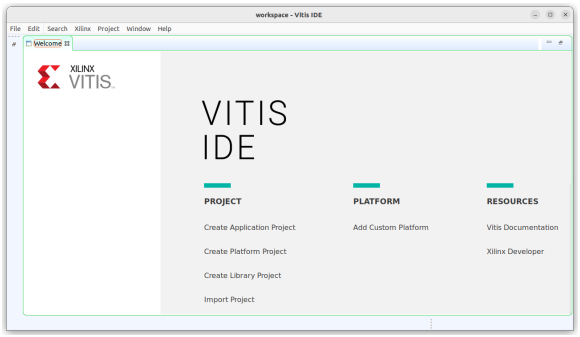


Figure F.73

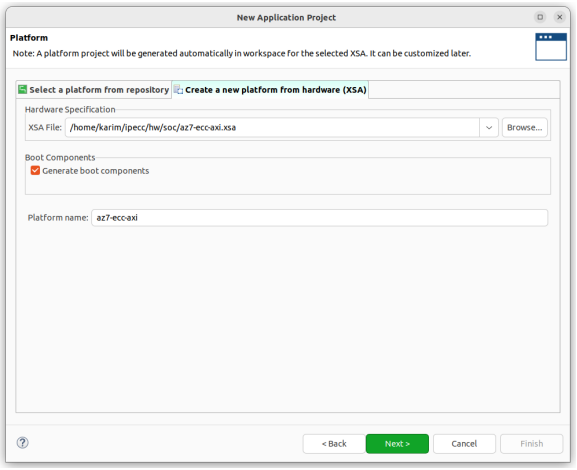


Figure F.74

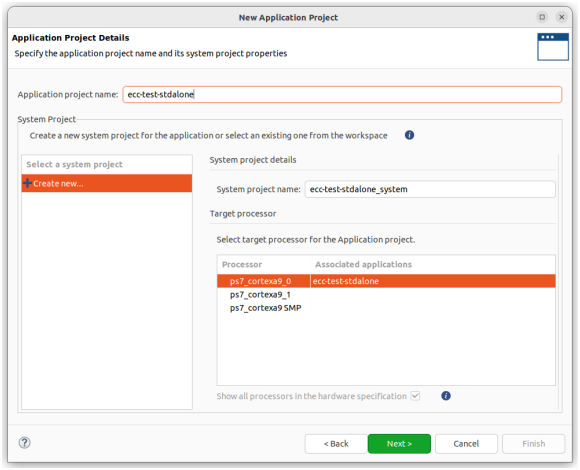


Figure F.75

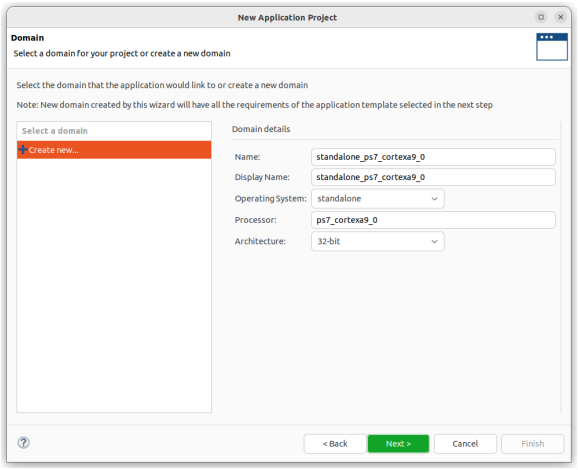


Figure F.76

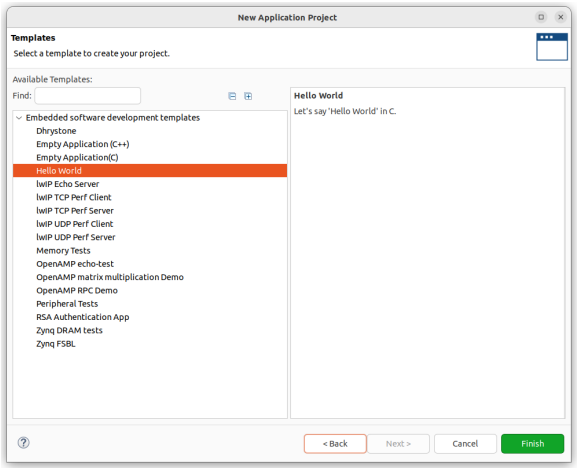


Figure F.77

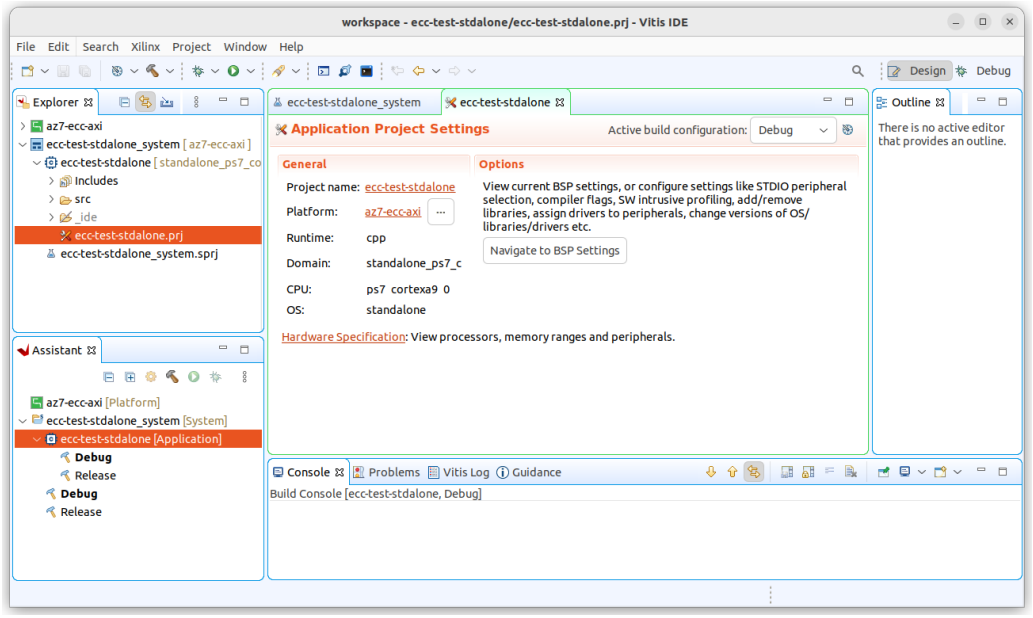


Figure F.78

```
47
48 #include <stdio.h>
49 #include "platform.h"
50 #include "xil_printf.h"
51
52
53 int main()
54 {
55     init_platform();
56
57     print("Hello World, this is IPECC test program.\n\r");
58     cleanup_platform();
59     return 0;
60 }
61
```

Figure F.79

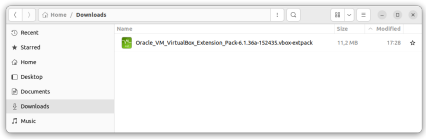


Figure F.80

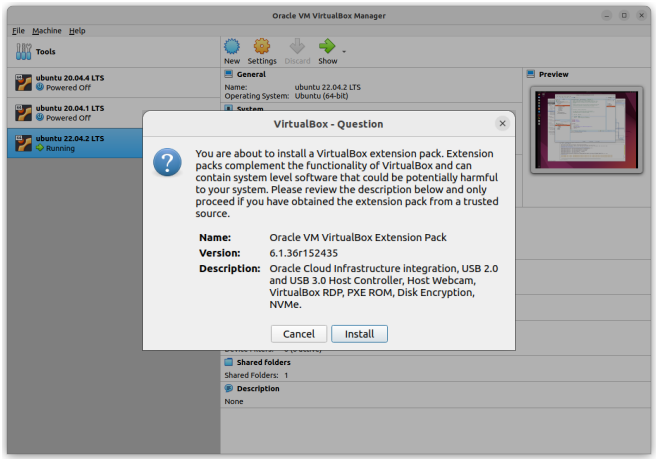


Figure F.81

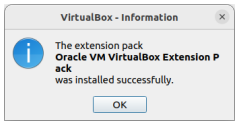


Figure F.82

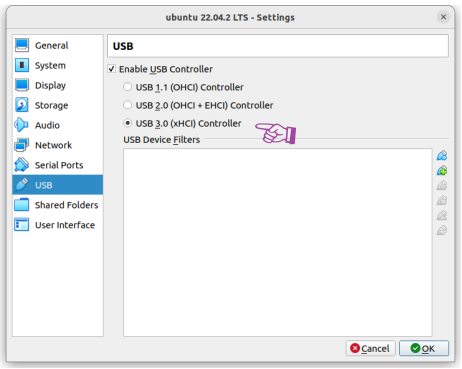


Figure F.83

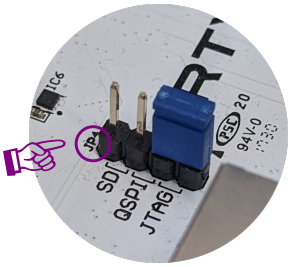


Figure F.84

```
[ 2204.536430] usb 1-4.2: new high-speed USB device number 8 using xhci_hcd
[ 2204.654766] usb 1-4.2: New USB device found, idVendor=0403, idProduct=6010, bcdDevice= 7.
[ 2204.654784] usb 1-4.2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 2204.654791] usb 1-4.2: Product: Digilent Adept USB Device
[ 2204.654797] usb 1-4.2: Manufacturer: Digilent
[ 2204.654802] usb 1-4.2: SerialNumber: 003017A702A6
[ 2204.660192] ftdi_sio 1-4.2:1.0: FTDI USB Serial Device converter detected
[ 2204.660272] usb 1-4.2: Detected FT2232H
[ 2204.660777] usb 1-4.2: FTDI USB Serial Device converter now attached to ttyUSB0
[ 2204.663982] ftdi_sio 1-4.2:1.1: FTDI USB Serial Device converter detected
[ 2204.664067] usb 1-4.2: Detected FT2232H
[ 2204.664388] usb 1-4.2: FTDI USB Serial Device converter now attached to ttyUSB1
```

Figure F.85

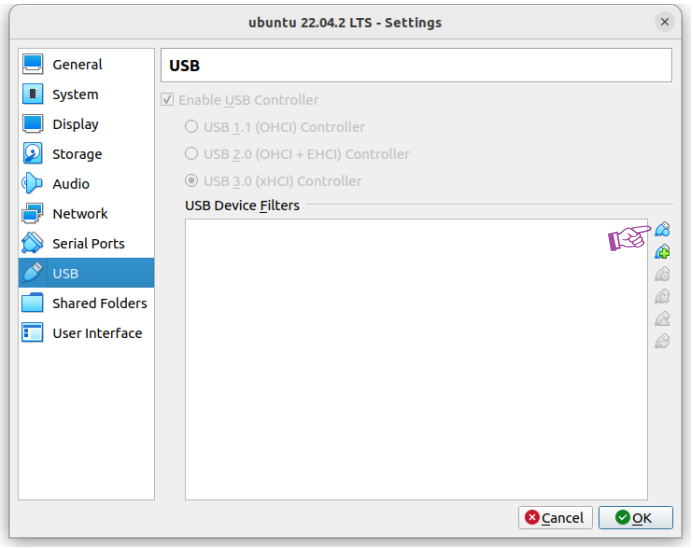


Figure F.86

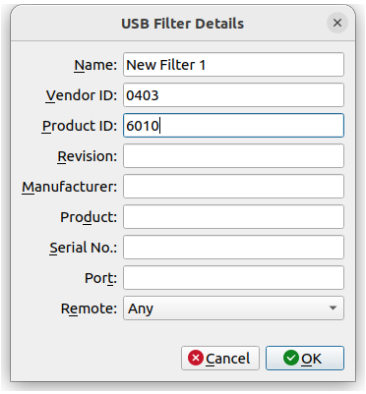


Figure F.87

```
$ sudo ./install_drivers
[sudo] password for myself:
INFO: Installing cable drivers.
INFO: Script name = ./install_drivers
INFO: HostName = vm
INFO: Current working dir = /home/myself/work/xilinx/Vivado/2022.1/Vivado/2022.1/data/xicom/cable_drivers/lin64/install_script/install_drivers
INFO: Kernel version = 6.2.0-31-generic.
INFO: Arch = x86_64.
Successfully installed Digilent Cable Drivers
--File /etc/udev/rules.d/52-xilinx-ftdi-usb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-ftdi-usb.rules = 0000.
--Updating rules file.
--File /etc/udev/rules.d/52-xilinx-pcusb.rules does not exist.
--File version of /etc/udev/rules.d/52-xilinx-pcusb.rules = 0000.
--Updating rules file.

INFO: Digilent Return code = 0
INFO: Xilinx Return code = 0
INFO: Xilinx FTDI Return code = 0
INFO: Return code = 0
INFO: Driver installation successful.
CRITICAL WARNING: Cable(s) on the system must be unplugged then plugged back in order for the driver scripts to update the cables.
```

Figure F.88

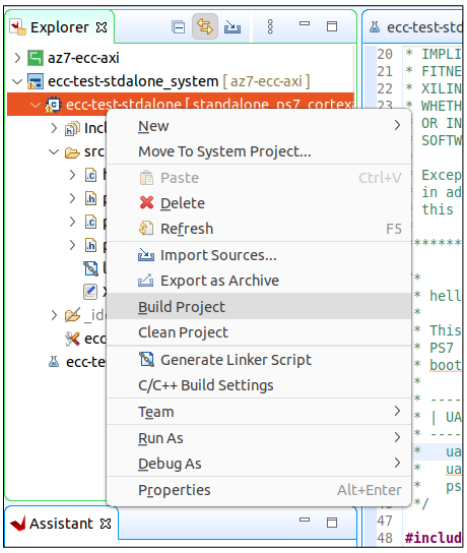


Figure F.89

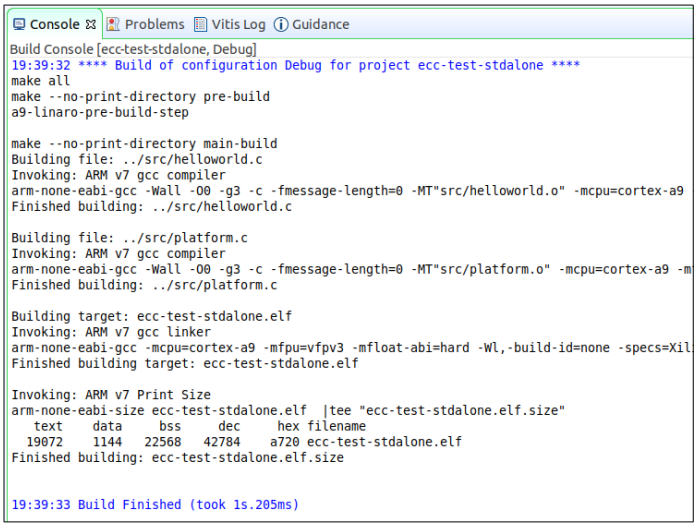


Figure F.90

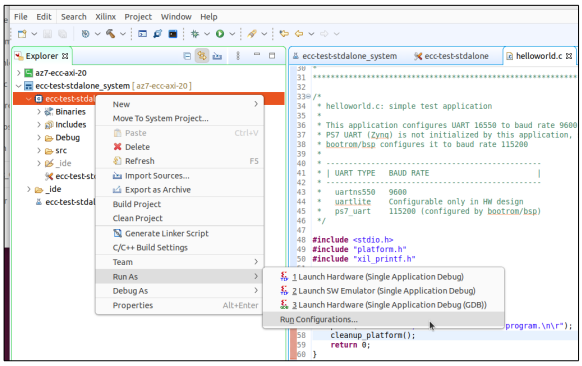


Figure F.91

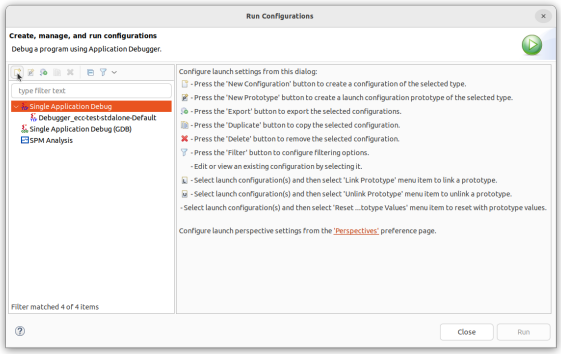


Figure F.92

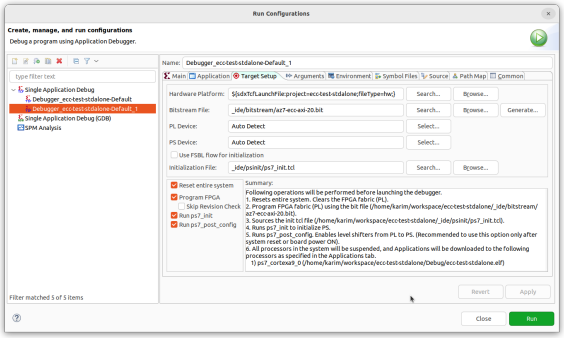


Figure F.93

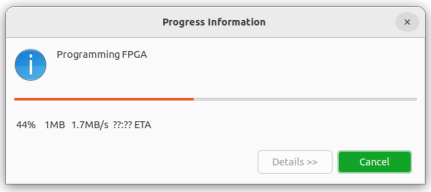


Figure F.94

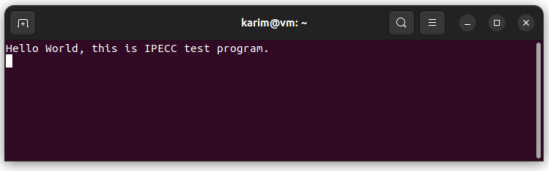


Figure F.95

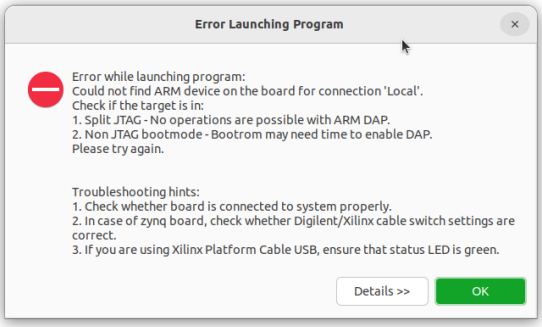


Figure F.96

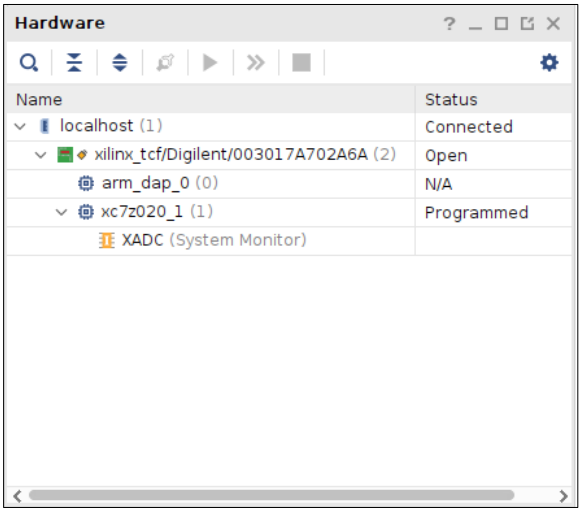


Figure F.97

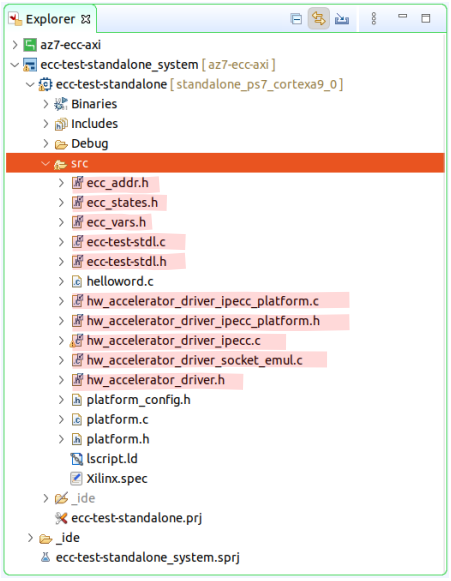


Figure F.98

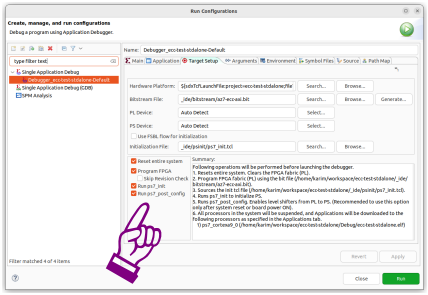


Figure F.99

```
Welcome to the standalone driver test!
============================================================
== Test 24-bit test: 2-torsion point * with an even scalar
============================================================
Entering in hw_driver_setup.
hw_driver_setup in standalone mode
*base_addr_p = 0x40000000
*pseudotrng_base_addr_p = 0x00000000
OK, loaded IP @40000000
nn=32
a=0x007fffd8
b=0x004729ea
p=0x007fffdb
q=0x00800600
k=0x007fd31e
Px=0x0078c1bc
Py=0x00000000
Poutx=0x00000000
Pouty=0x00000000
Iszero R0: 0, Iszero R1: 1
================================================
== Test 127-bit test: k = non-0 value (nominal)
================================================
nn=128
a=0x404eabbfca15879f989ee6aeec36ee11
b=0x15f3438cd61fb845350454814dc44e3c
p=0x5d60a0d84dcea66622279b301db4afcb
q=0x5d60a0d84dcea6658b745b3ce5d559de
k=0x4d2c5a8d9f4634e4b9163b1a42621286
Px=0x12e1d2159454ae289e3faf69d04dfc4e
Py=0x3e9c8186ab0b17ecc03c021af14ac288
Poutx=0x22f7840827566ccfa681fa4175f689fa
Pouty=0x13ac181c43b4b16bffaef137c6fd6ceb
Iszero R0: 0, Iszero R1: 0
================================================
== Test 256-bit test: k = non-0 value (nominal)
================================================
nn=256
a=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
b=0xee353fca5428a9300d4aba754a44c00fdfec0c9ae4b1a1803075ed967b7bb73f
p=0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c03
q=0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1
k=0xf1adb2506355162d0de14468748fb171f730bd40f6595fe1732651df00589fcf
Px=0xb6b3d4c356c139eb31183d4749d423958c27d2dcaf98b70164c97a2dd98f5cff
Py=0x6142e0f7c8b204911f9271f0f3ecef8c2701c307e8e4c9e183115a1554062cfb
Poutx=0x4c0338872b9f13aa0c01f74c7f504eeae9d947f54e9bb80dfce61c3f2e5d151d
Pouty=0xa9571e1e0447b7fb302eab3a57dae4866f238eac48f67993a37cc660a831879b
Iszero R0: 0, Iszero R1: 0
```

Figure F.100

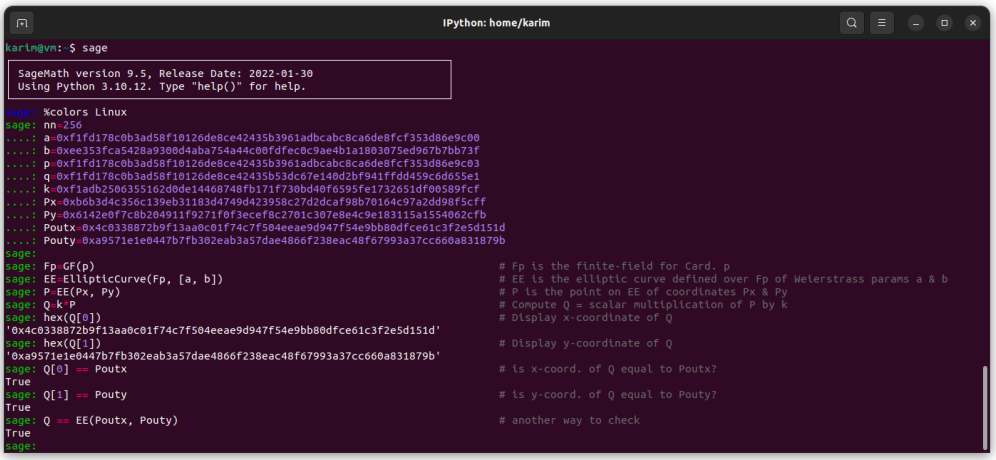


Figure F.101

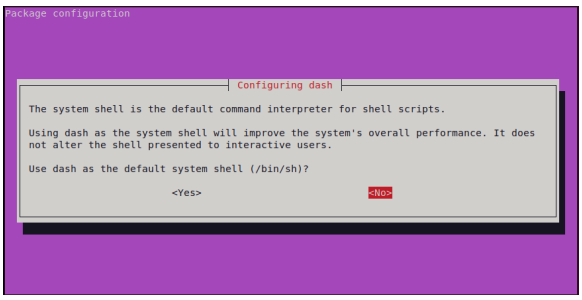


Figure F.102

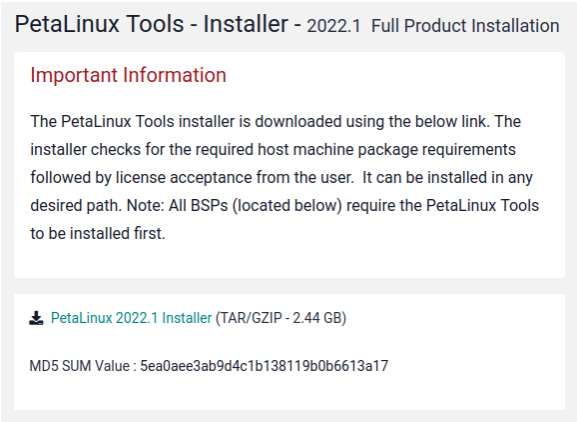


Figure F.103

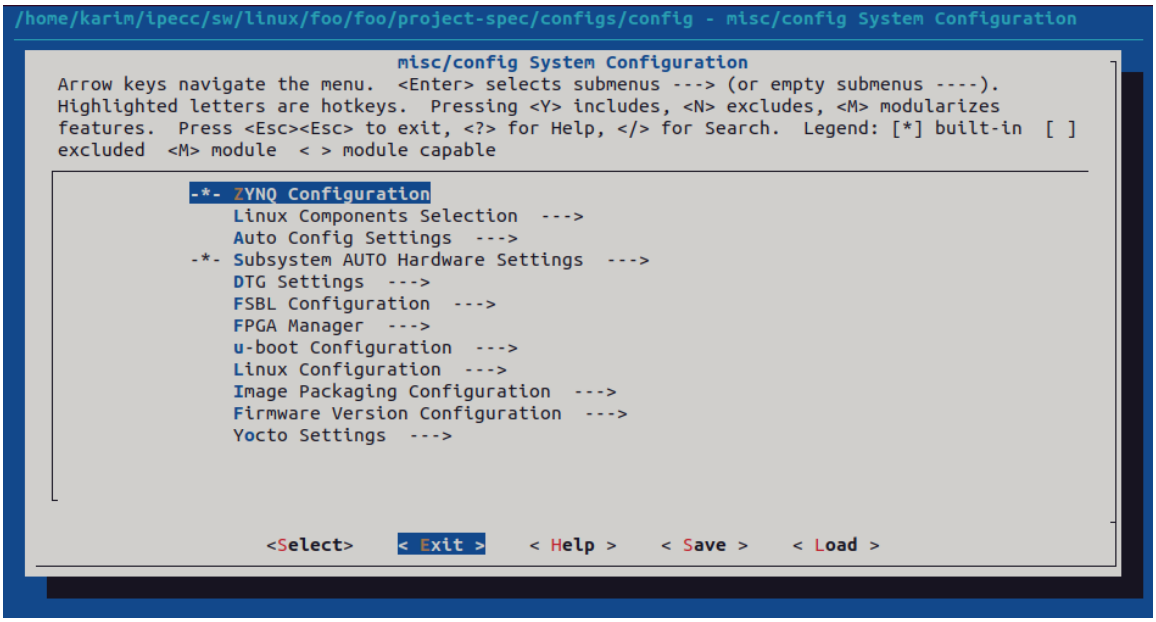


Figure F.104

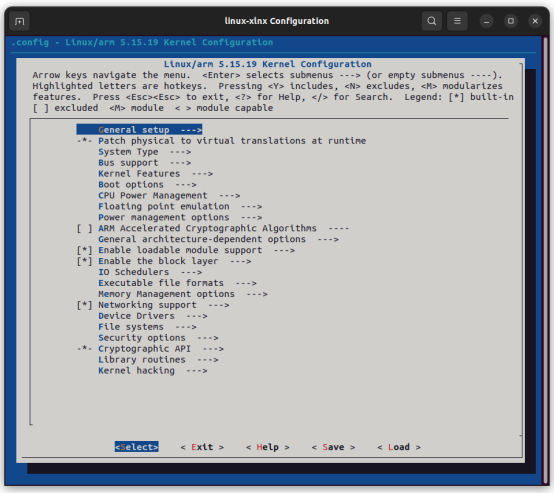


Figure F.105

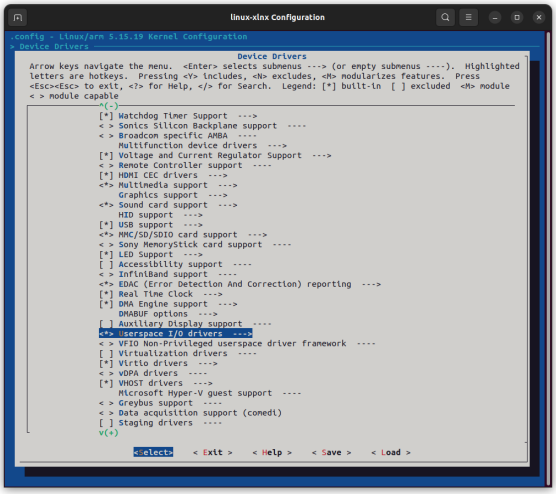


Figure F.106

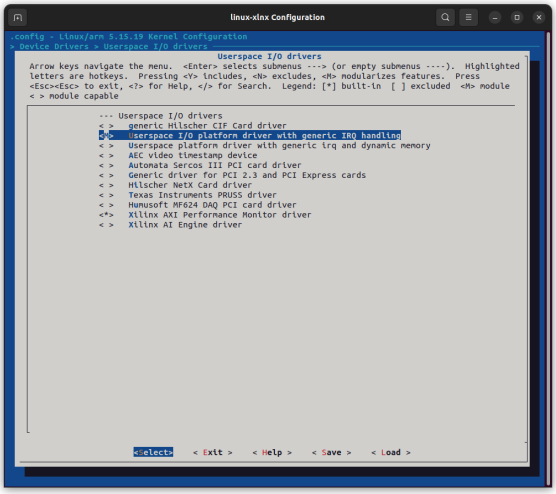


Figure F.107

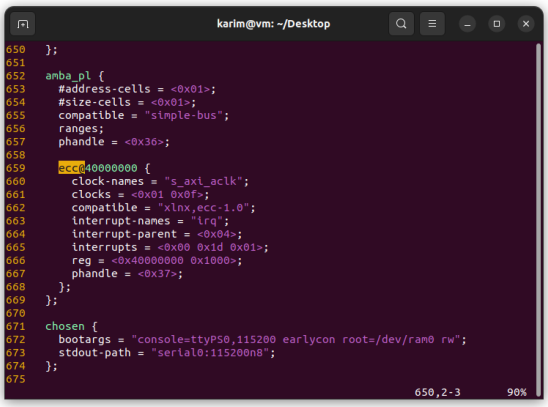


Figure F.108

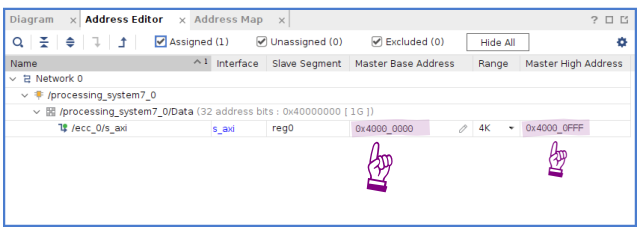


Figure F.109

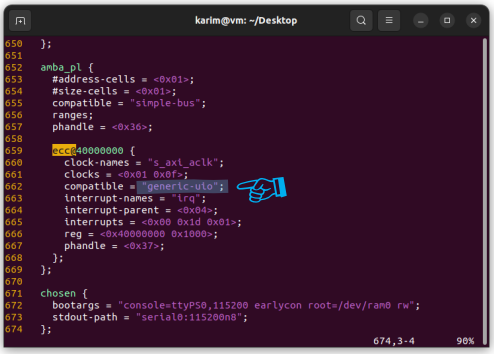


Figure F.110

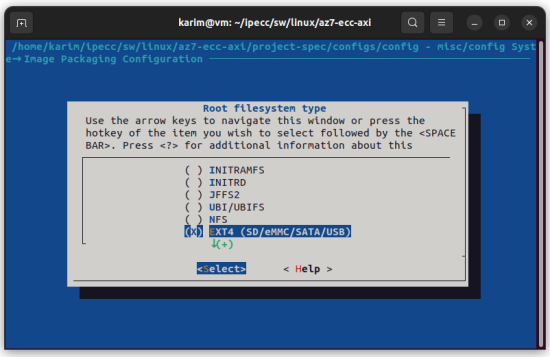


Figure F.111

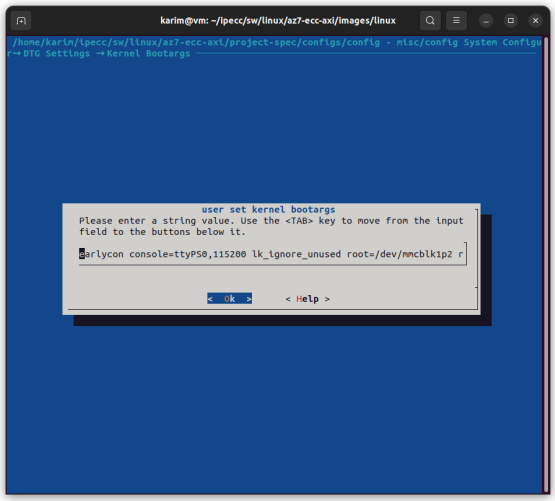


Figure F.112

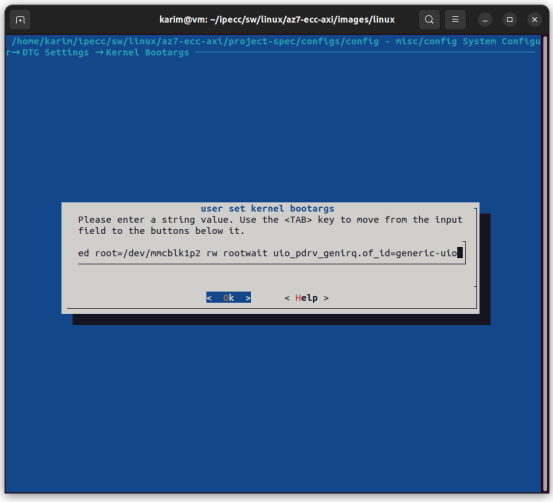


Figure F.113

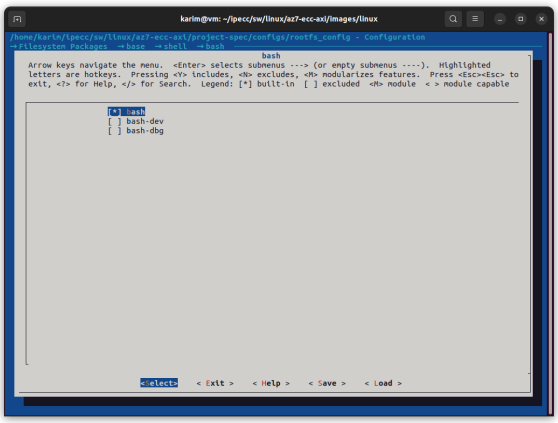


Figure F.114

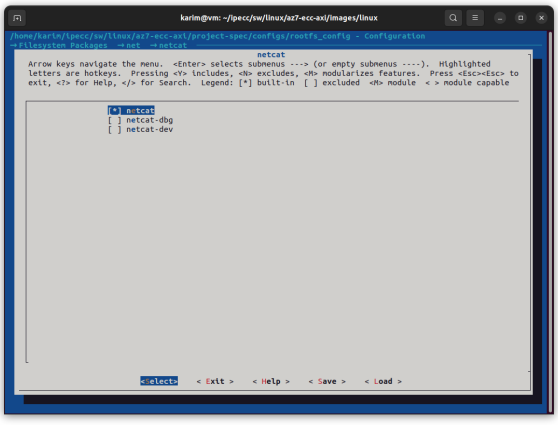


Figure F.115

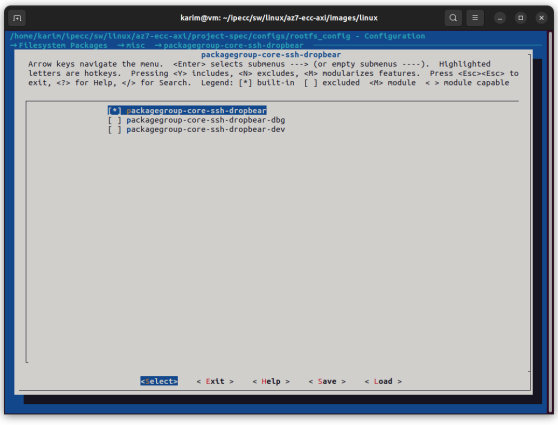


Figure F.116

Figure F.117

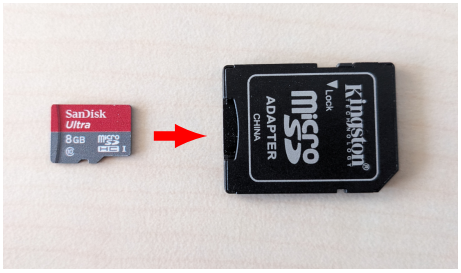


Figure F.118

Figure F.119

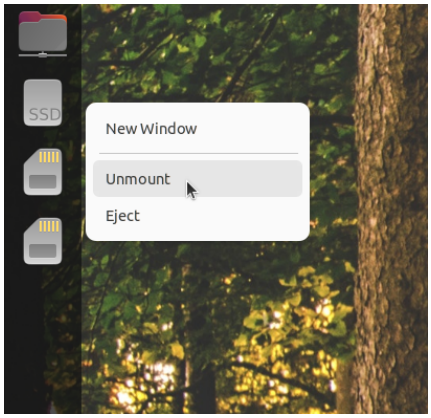


Figure F.120

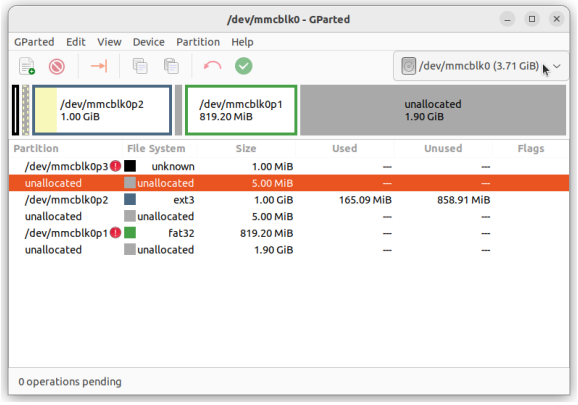


Figure F.121

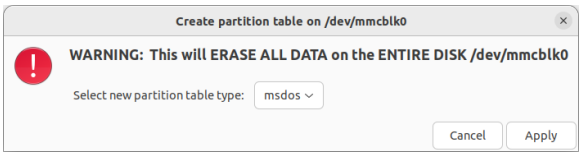


Figure F.122

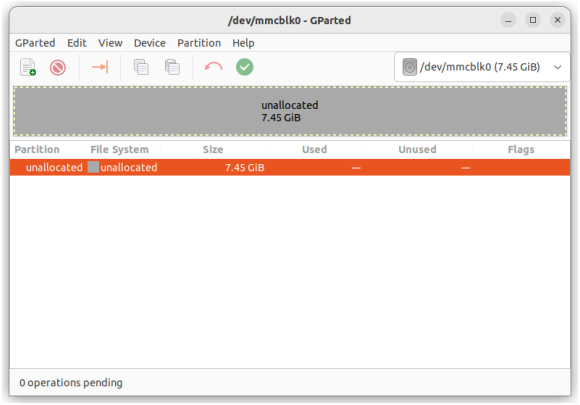


Figure F.123

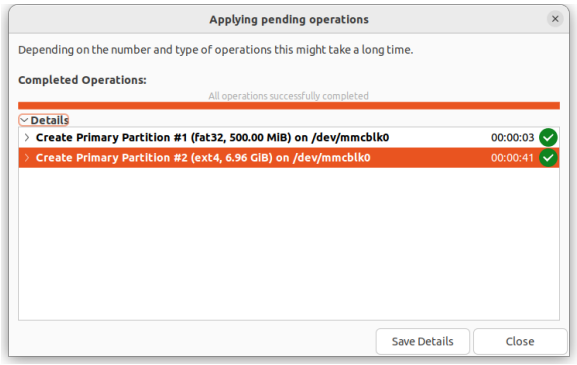


Figure F.124

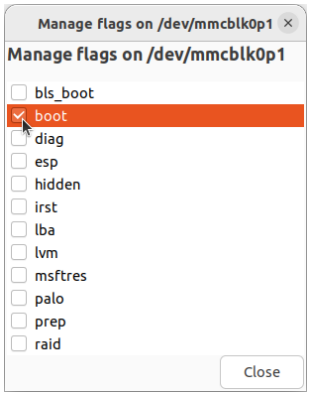


Figure F.125

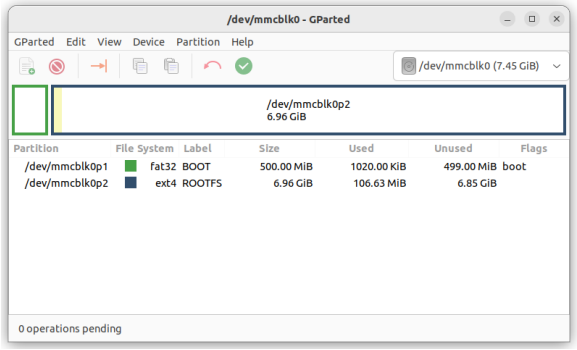


Figure F.126

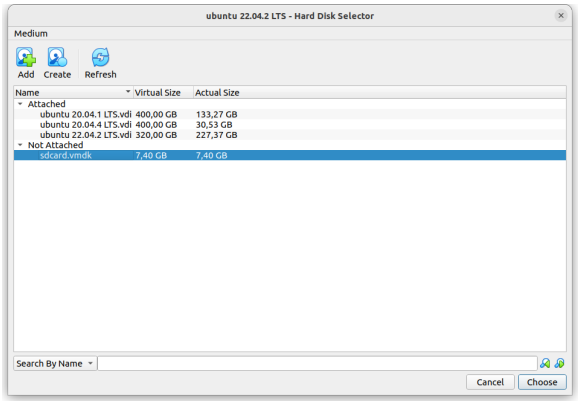


Figure F.127

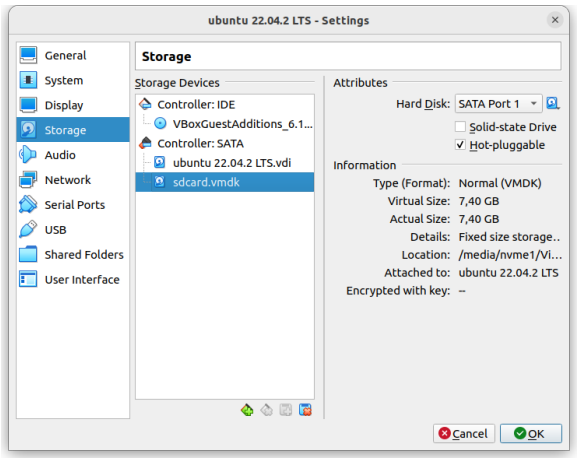


Figure F.128

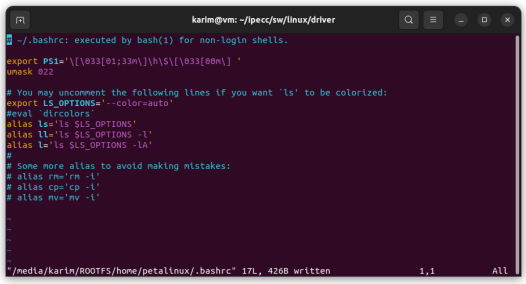


Figure F.129

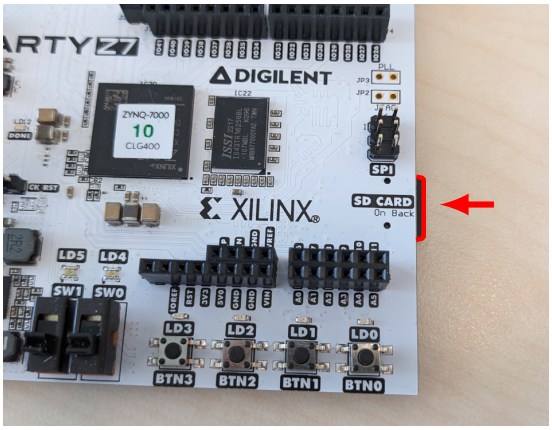


Figure F.130

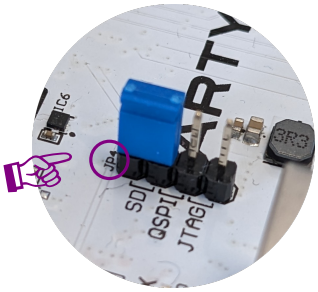


Figure F.131

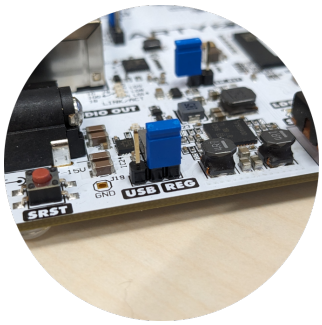


Figure F.132

Figure F.133

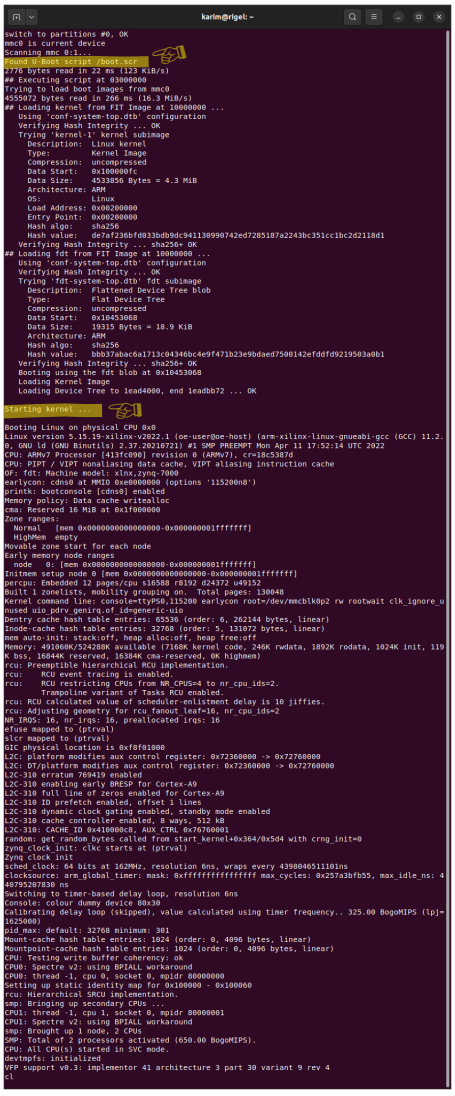


Figure F.134

```
Starting Dropbear SSH server: dropbear.
Starting rpcbind daemon...done.
starting statd: done
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting internet superserver: inetd.
NFS daemon support not enabled in kernel
Starting syslogd/klogd: done
Starting tcf-agent: random: crng init done
OK

PetaLinux 2022.1_release_S04190222 az7-ecc-axi /dev/ttyPS0

az7-ecc-axi login: petalinux
You are required to change your password immediately (administrator enforced).
New password:
Retype new password:
az7-ecc-axi:~$
```

Figure F.135

```
1 # VHD_SRC must be set with the ABSOLUTE path where IPECC microcode
2 # has been built (DO NOT use relative path!)
3 VHD_DIR=/home/myself/IPECC/hdl/common/ecc_curve_iram
4
5 # CONFIG #############
6 ARM_CC ?= arm-linux-gnueabihf-gcc
7 CFLAGS = -Wall -Wextra -Wpedantic -O3 -g3 -mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard -static
8 CFLAGS += -DWITH_EC_HW_DEBUG
```

Figure F.136

```
az7-ecc-axi$ stdbuf -oL nc -l -p 20000 | ./ecc-test-linux-uio
Driver in UIO mode
IP in debug mode (HW version 1.2.25)
nn min|average|max: 35|47|60
         [k]P     P+Q     [2]P      -P    P==Q    P==-Q   PonC   Total
  ok:     677      92       68      66      84       92     60    1139
 nok:       0       0        0       0       0        0      0       0
total:    677      92       68      66      84       92     60    1139
```

Figure F.137

Appendix G

VHDL source files

Source file	Simulation	Synthesis
hdl/common/ecc.vhd	✔	✔
hdl/common/ecc_axi.vhd	✔	✔
hdl/common/ecc_curve.vhd	✔	✔
hdl/common/ecc_customize.vhd	✔	✔
hdl/common/ecc_fp.vhd	✔	✔
hdl/common/ecc_fp_dram.vhd	✔	✔
hdl/common/ecc_fp_dram_sh_fishy_nb.vhd	✔	✔
hdl/common/ecc_fp_dram_sh_fishy.vhd	✔	✔
hdl/common/ecc_fp_dram_sh_linear.vhd	✔	✔
hdl/common/ecc_log.vhd	✔	✔
hdl/common/ecc_pkg.vhd	✔	✔
hdl/common/ecc_scalar.vhd	✔	✔
hdl/common/ecc_shuffle_pkg.vhd	✔	✔
hdl/common/ecc_software.vhd	✔	✔
hdl/common/ecc_utils.vhd	✔	✔
hdl/common/ecc_vars.vhd	✔	✔
hdl/common/fifo.vhd	✔	✔
hdl/common/mm_ndsp_pkg.vhd	✔	✔
hdl/common/mm_ndsp.vhd	✔	✔
hdl/common/sync2ram_sdp.vhd	✔	✔
hdl/common/syncram_sdp.vhd	✔	✔
hdl/common/virt_to_phys_ram.vhd	✔	✔
hdl/common/virt_to_phys_ram_async.vhd	✔	✔
hdl/common/ecc_curve_iram/ecc_addr.vhd	✔	✔
hdl/common/ecc_curve_iram/ecc_curve_iram.vhd	✔	✔
hdl/common/ecc_curve_iram/ecc_vars.vhd	✔	✔
hdl/common/ecc_trng/ecc_trng.vhd	✔	✔
hdl/common/ecc_trng/ecc_trng_pkg.vhd	✔	✔
hdl/common/ecc_trng/ecc_trng_pp.vhd	✔	✔
hdl/common/ecc_trng/ecc_trng_srv.vhd	✔	✔
hdl/common/ecc_trng/es_trng.vhd	✘	✔
hdl/common/ecc_trng/es_trng_aggreg.vhd	✘	✔
hdl/common/ecc_trng/es_trng_bitctrl.vhd	✘	✔
hdl/common/ecc_trng/es_trng_sim.vhd	✔	✘
hdl/techno-specific/asic/es_trng_bit_asic.vhd	✘	✔
hdl/techno-specific/asic/large_shr_asic.vhd	✔	✔
hdl/techno-specific/asic/macc_asic.vhd	✔	✔
hdl/techno-specific/asic/maccx_asic.vhd	✔	✔
hdl/techno-specific/ilatera/es_trng_bit_ialtera.vhd	✘	✔
hdl/techno-specific/ilatera/large_shr_ialtera.vhd	✔	✔
hdl/techno-specific/ilatera/macc_ialtera.vhd	✔	✔
hdl/techno-specific/ilatera/maccx_ialtera.vhd	✔	✔
hdl/techno-specific/xilinxa/series7/es_trng_bit_series7.vhd	✘	✔
hdl/techno-specific/xilinxa/series7/large_shr_series7.vhd	✔	✔
hdl/techno-specific/xilinxa/series7/macc_series7.vhd	✔	✔
hdl/techno-specific/xilinxa/series7/maccx_series7.vhd	✔	✔
hdl/techno-specific/xilinxa/ultrascale/es_trng_bit_ultrascale.vhd	✘	✔
hdl/techno-specific/xilinxa/ultrascale/large_shr_ultrascale.vhd	✔	✔
hdl/techno-specific/xilinxa/ultrascale/macc_ultrascale.vhd	✔	✔
hdl/techno-specific/xilinxa/ultrascale/maccx_ultrascale.vhd	✔	✔
sim/ecc_tb.vhd	✔	✘
sim/ecc_tb_pkg.vhd	✔	✘
sim/ecc_tb_vec.vhd	✔	✘

Table G.1: All VHDL sources for IPECC, for simulation and/or for synthesis.

Appendix H

Components and signals

Figure H.1 (next page) shows the block-diagram view of IPECC including all components and signals. This corresponds to the top-level entity ecc of RTL architecture.

For the sake of clarity, figure H.2 shows the same block-diagram architecture but includes only a subset of signals. As compared to figure H.1, the following groups of signals have been omitted: signals related to behavorial simulation (in brown on fig. H.1), signals related to debug feature (blue), signals related to the dynamic prime size feature (orange), signals related to the asynchronous Montgomery multipliers feature (red) and signals related to the big number memory shuffling countermeasure (violet/magenta).

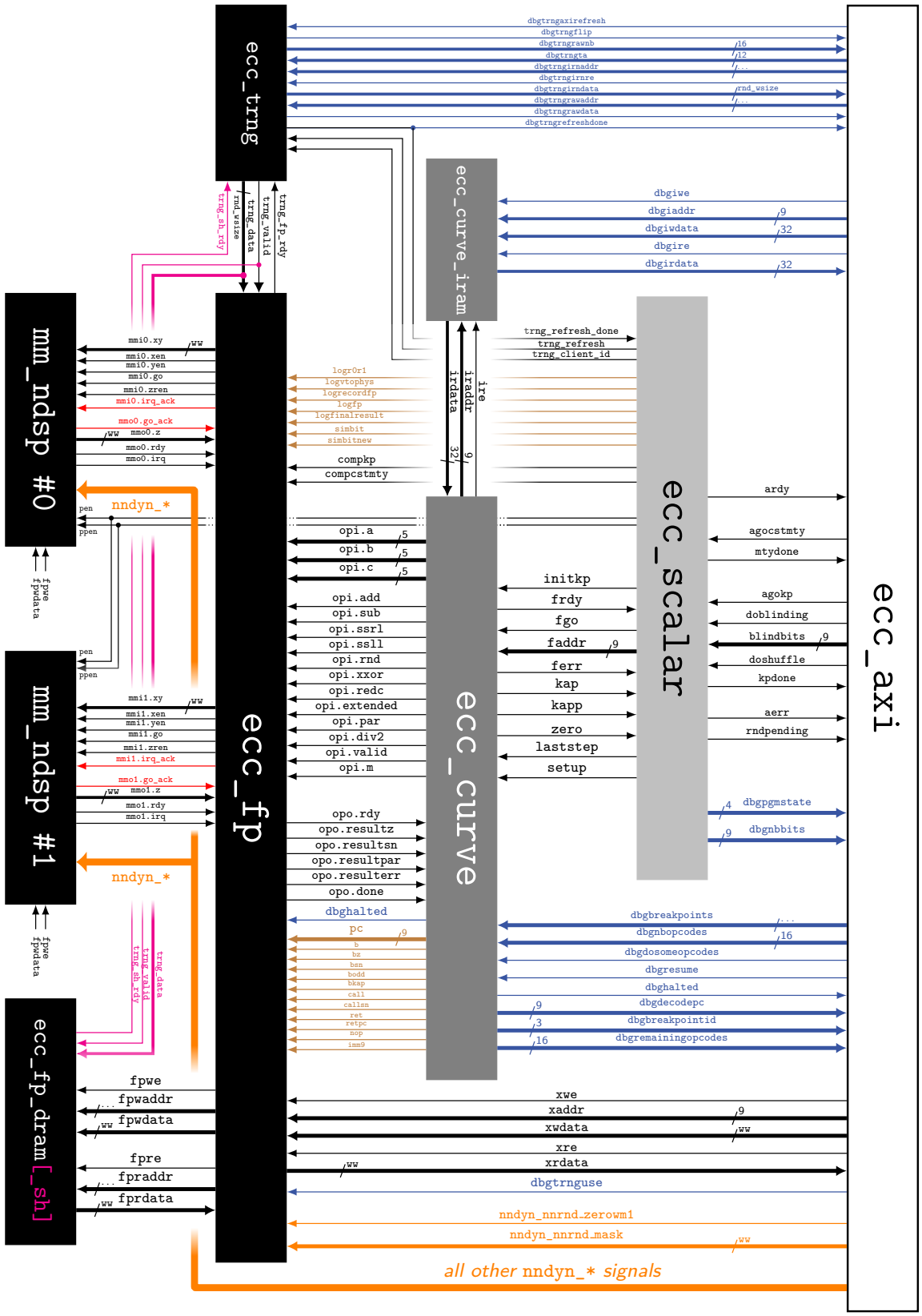


Figure H.1: Architecture of IPECC showing all components and all signals

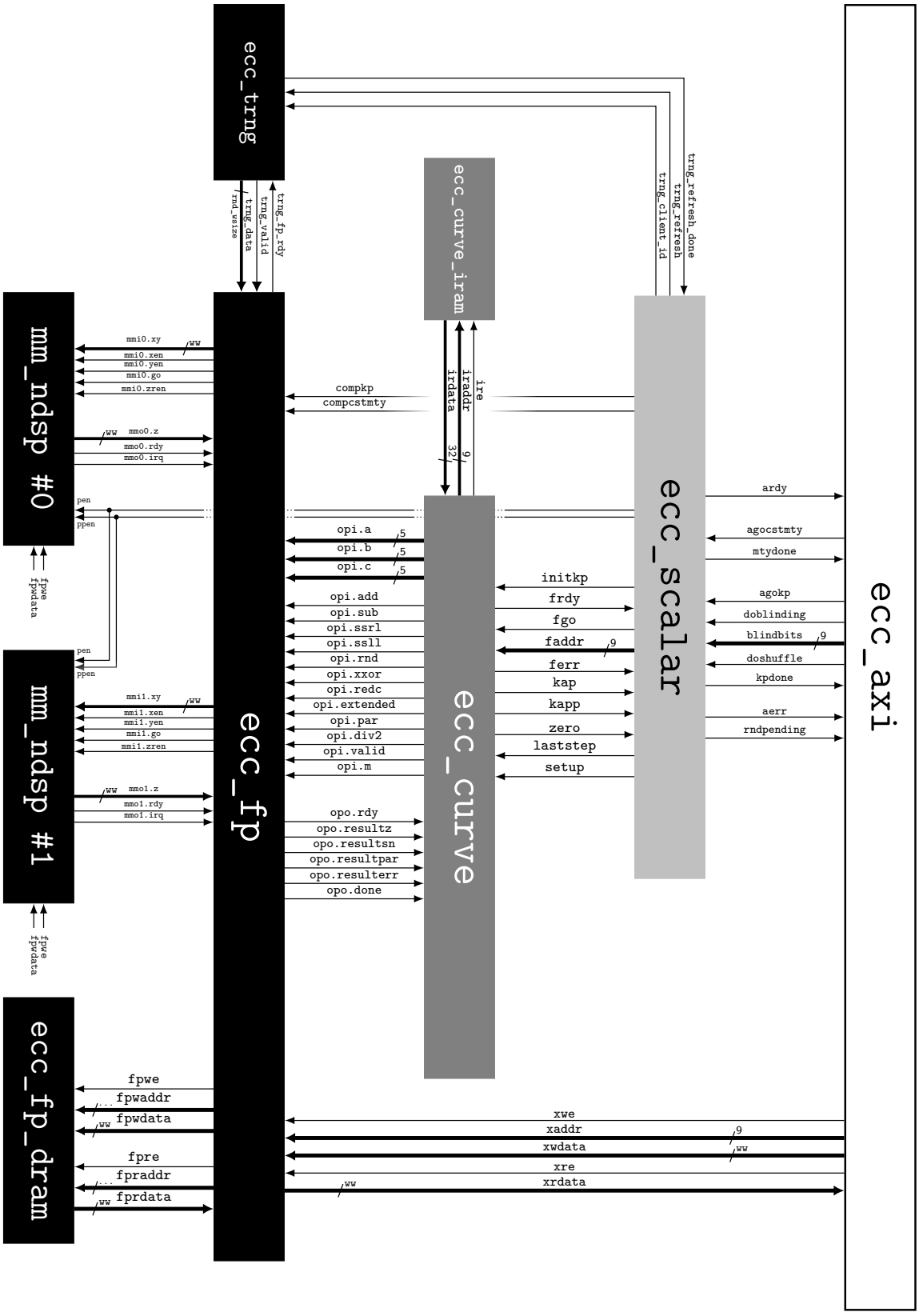


Figure H.2: Architecture of IPECC showing main signals only (all signals related to behavioral simulation, debug feature, dynamic prime size feature, asynchronous Montgomery multipliers feature and big number memory shuffling countermeasure have been removed)

Bibliography

[ARM11] ARM. *AMBA AXI and ACE Protocol Specification (ARM ID=IHI0022D, ID102711)*. 2011.

[BJ02] Éric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In *International Conference on Theory and Practice of Public Key Cryptography*, 2002.

[BSS+99] I. Blake, G. Seroussi, G. Seroussi, N. Smart, London Mathematical Society, and N.J. Hitchin. *Elliptic Curves in Cryptography*. Lecture note series. Cambridge University Press, 1999.

[CFA+12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2012.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology — ASIACRYPT'98*, pages 51–65, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Cor99] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[GJM10] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-z addition formulæ and binary ladders on elliptic curves. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[GJM+11] Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on weierstraß elliptic curves from co-z arithmetic. *J. Cryptogr. Eng.*, 1(2):161–176, 2011.

[HM11] Darrel Hankerson and Alfred Menezes. *Elliptic Curve Cryptography*, pages 397–397. Springer US, Boston, MA, 2011.

[HMV04] D Hankerson, A Menezes, and S Vanstone. *Guide to Elliptic Curve Cryptography*. 2004.

[IIT03] Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. A practical countermeasure against address-bit differential power analysis. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 382–396, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Joy07] Marc Joye. Highly regular right-to-left algorithms for scalar multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2007.

[JT01] Marc Joye and Christophe Tymen. Protections against differential analysis for elliptic curve cryptography. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2001.

[Mel07] Nicolas Meloni. New point addition formulae for ecc applications. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, pages 189–201, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[Mon87] P. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MVOV96] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[Riv11] Matthieu Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptology ePrint Archive*, 2011:338, 01 2011.

[SV93] M. Shand and J. Vuillemin. Fast implementations of rsa cryptography. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 252–259, 1993.

[VD10] Alexandre Venelli and François Dassance. Faster side-channel resistant elliptic curve scalar multiplication. 01 2010.

[YRG+18] Bohan Yang, Vladimir Rožić, Miloš Grujić, Nele Mentens, and Ingrid Verbauwhede. ES-TRNG: A high-throughput, low-area true random number generator based on edge sampling. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):267–292, Aug. 2018.