IPECC

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

Draft

This is a preliminary version of the IP documentation reduced to a tutorial for Xilinx FPGAs (later an appendix).

There are two versions of the tutorial: a short version, probably the one you want to read, where actions to take were 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 or beginners. Both versions use the same numbering so you can directly switch from the short version to the other 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 at the end of the doc so that you don't necessarily have to print them, in case you *would prefer not to*.

ANSSI

Agence Nationale de la Sécurité des Systèmes d'Information

© 2023 ANSSI

Appendix A

A Tutorial: IPECC on Xilinx FPGAs (short version)

This is a short version of the tutorial of appendix B that bannishes all claptrap by giving instructions in a concise and systematic way. Each action you'll need to execute has been numbered with each one matching the identical number in the verbose version. So in case something happens not to be clear below, you might directly find more information/help in the longer version (which also includes several troubleshooting indications). Likewise the figures in this version are identical to the ones of the longer version.

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

⟨~/.bashrc+⟩: add the line `export IPECC="~/IPECC"`

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
CLI $ evince latex/ecc_curve_iram.pdf &
```

This 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.

Select **Xilinx Unified Installer 2022.1: Linux Self Extracting Web Installer** (fig-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 In the install tool select **Vitis** (fig-5) (includes **Vivado**).

10 When selecting the devices to install (fig-5) 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).

- Ignore pop-up window inviting you to install the latest release of Vivado (fig-6).
- Ignore red warning threatening you that you are on an unsupported OS
- 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 8).

Vivado project creation

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

- Section **Project Name:** Project name: `ecc`
  Project location: `ipecc/hw/ip`
  Keep box *Create project subdirectory* unchecked
- Section **Project Type:** RTL Project
  Keep box *Do not specify sources at this time* checked
- 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).

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-9). 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*.

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-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 `ecc.srcs/sources_1/imports/hdl`. Parameters you must edit are shown on fig. 18 p. 70, 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-19 on p. 70).

☞ 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-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-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-20) and informations are logged onto the Vivado simulation console (fig-21). The numbers displayed in violet on the two figures 20 and 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 step **23**).

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-22 p. 73). 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]
```

You can see on the above excerpt that the correspondence with the parameters that need to be set in *Sage* is quite clear (fig-23 p. 73).

☞ 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-24 (p.74).

- 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-25, p. 74).

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-26 p. 75)

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-27 p. 75).

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-29-30 p. 76):

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-28 p. 75)

☞ 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.

[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 §**??**, see in particular algorithm **??** and the discussion pertaining to it in p.**??**.

[2] Remember this file is in `ipecc/hw/ip/ecc.srcs/sources_1/imports/hdl/`.

[3] Remember this file is in `ipecc/hw/ip/ecc.srcs/sim_1/imports/sim/`. This is the source file of the RTL testbench.

`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`

Now the two log files should be strictly identical, as attested by the `cmp` command (fig-31-32 p. 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-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. 6) 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-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-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-39)
- ▶ Clock Configuration ▶ PL Fabric Clocks ▶ Check box `FCLK_CLK1` ▶ Set Requested Frequency(MHz) to `200` ▶ OK (ignore possible warning about negative DDR DQS delay, fig-41).

The Zynq processor is upgraded (fig-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-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-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 ports (`irqo`, `dbgtrigger` and `dbghalted`) of the IP unconnected. These are outputs so it doesn't matter if they stay unconnected (synthesizers will trim them out).

The Block Design should now look like fig-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-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-48.
- Retry *Validate Design* (shortcut `F6`).

Vivado is now happy.

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-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'–2' and then you're being informed that *«Out-of-context modules were launched for generating output product»* (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. This seems to be an abstraction level very close to a synthesized netlist.

- Right-top corner of Vivado: select *I/O planning* layout (fig-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-51 taken from the online Reference Manual of the board). See also fig-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-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-54).

- Double-click on this file to display its content, the two `set_property` Tcl commands (fig-55) are more or less human-readable.
- *Flow Navigator* ▶ *SYNTHESIS* ▶ *Run Synthesis* ▶ *Launch Runs* ▶ Set *Launch Runs* to the max (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 launches 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-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-58 and fig-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) ▶ 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-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-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 compiling Linux **now**: for that make a small detour to step **44** then resume back the tutorial here, leaving the build running in the background. Even if it means customizing the kernel and rootfs afterwards to fit exactly the hardware, subsequent compilations will be much shorter.

Software part: driving IPECC from bare metal software

34 *Tools* ▶ *Launch Vitis IDE*

Keep default choice for the workspace (`~/workspace`, fig-72).

35 *Create Application Project* (fig-73) ▶ *New Application Project* window ▶ tab *Create a new platform from hardware (XSA)* ▶ XSA file: `~/ipecc/hw/soc/az7-ecc-axi.xsa` (leave *Generate Boot Components* box checked) ▶ *Next* ▶ *Application project name*: `ecc-test-stdalone` (fig-75) ▶ *Next* ▶ *Domain*: nothing to do ▶ *Next* ▶ *Templates:* default `Hello World` ▶ *Finish*.

36 *Explorer* (fig-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-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-84 (JTAG configuration).

Plug the board into your machine using the USB cable.

CLI
```
$ ls -1 /dev/ttyUSB*
/dev/ttyUSB0
/dev/ttyUSB1
```

40 See steps **37** - **38** of the detailed version of the tutorial to see how to create an USB VirtualBox filter for having the guest VM «snatching» the USB interfaces automatically upon plug.

41 **Xilinx cable installation** (adapt this to your Vivado install path)

CLI
```
$ cd ~/xilinx/
$ cd Vivado/2022.1/Vivado/2022.1/
$ cd xicom/cable_drivers/lin64/install_script/install_drivers
$ sudo ./install_drivers
[sudo] password:    (fig-88)
```

Unplug the board and plug it back again to have the configuration changes to take effect.

42 CLI
```
$ screen /dev/ttyUSB0 115200
```

Right-click on project name `ecc-test-stdalone` ▶ *Build Project* (fig-89-90).

Right-click again on project name `ecc-test-stdalone` ▶ *Run As* ▶ *Run Configurations* (fig-91) ▶ *Run Configurations* window ▶ *Single Application Debug* (fig-92) ▶ button  ▶ *Run* (fig-93).

After a few seconds you should see the message string from the `printf` appearing in the `screen` console (fig-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
$ 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
```

As you add along symbolic links you can see that Vitis/Eclipse dynamically detects the presence of the new source files and automatically adds them to the project (fig-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 ▶ *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-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-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: 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-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                                                           # I think we're safe
```

Notice on the `screen` console (fig-100) how the IP has detected the nullity of resulting point. For $[k]\mathcal{P}$ computations result is stored in point $\mathcal{R}_1$. We can see for instance that for `nn=32` 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 here 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-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-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 43, 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-105-106, possibly already selected).

Device drivers ▶ *Userspace I/O drivers* ▶ *Userspace I/O platform driver with generic IRQ handling* ▶ `<M>` (fig-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-108). Notice value `0x40000000` of IP base address. This matches value set by Vivado in SoC Design Block[4]. 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:

```
/* IPECC */
&ecc_0 {
   compatible = "generic-uio";
};
```

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

[4] If you open project `az7-ecc-axi` back in Vivado, open the Block Design and go to the *Address Editor* tab (fig-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.

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-110).

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

CLI `$ petalinux-config` (fig-104)

Image Packaging Configuration ▶ *Root filesystem type* ▶ `EXT4 (SD/eMMC/SATA/USB)` (fig-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-114)

Filesystem Packages ▶ *net* ▶ *netcat* ▶ *netcat* ▶ `[*]` (fig-115)

Filesystem Package ▶ *misc* ▶ *packagegroup-core-ssh-dropbear* ▶ *packagegroup-core-ssh-dropbear* ▶ `[*]` (fig-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-112 and fig-113):

```
earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk1p2
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-129), save and exit.

CLI `$ sync`
CLI `$ sudo umount /dev/mmcblk1p1`
CLI `$ sudo umount /dev/mmcblk1p2`

52 **Booting Linux**

Remove the microSD card and plug it into slot `J9` of the Arty board (fig-130).

Change position of jumper `JP4` as illustrated on fig-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-132) and to `USB` in the latter.

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-133):

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-134)

53 **Connect the board to your local network**

☞ Refer to **53** 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 136). 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.

Appendix B

A Tutorial: IPECC on Xilinx FPGAs

This chapter is a short 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-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.

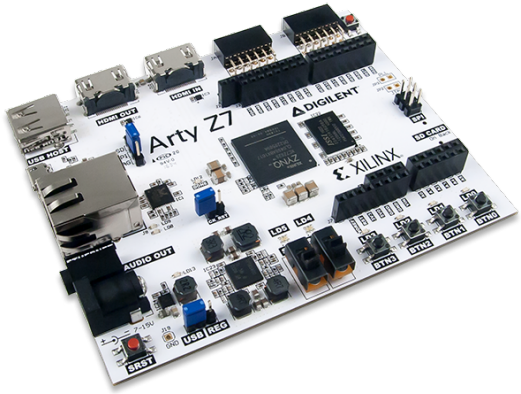


Figure 1: The Arty Z7 board from Digilent comes in two versions: 7z10 and 7z20.

☞ 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 will instanciate and configure the IP inside the programmable logic part (PL) of the device, simulate and synthesize the hardware using the Vivado FPGA tool chain. In a second part you will compile the Linux sources with its 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, we will 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) and a *known free version of Vivado* (2022.1 ML standard) 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.

Each action you will need to execute in this version has been numbered (there are **56** of these) with all the figures related to the tutorial (including the many tedious GUI screenshots) moved to the end of the chapter – thus you can spare paper and toner by not necessarily printing them.

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-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-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-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 Choose **Vitis** (fig-5). 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-5). Be sure however to leave the box *Install devices for Alveo and Xilinx edge acceleration platforms* checked, as it'll install required libraries.

Ignore the pop-up window inviting you to install the latest release of Vivado (fig-6). 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-7).

The required disk space should amount to a little more than 165 GB (`:-.`) The installation 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 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 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-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*. Files will be copied in `ipecc/hw/ip/ecc.srcs/sources_1/imports/hdl/`.

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`

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 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-10). Review the *New Project Summary* (fig-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-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-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-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`. This file contains every parameter you need to set in order to customize the IP for a specific hardware (see appendix **??** for more information on IP customization). Fig. 18 on p. 70 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 (`simlog = /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 **??**) 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/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-19 on p. 70). 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-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-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-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 20) and informations are logged onto the Vivado simulation console (see figure 21). The numbers displayed in violet on the two figures 20 and 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-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-20).
- Receiving number p and a triggers in the IP the computation of the two Montgomery constants (see §**??**): `ecc_scalar` enters **main FSM** state `cst` (see mark ❷ on figs. 20 and 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-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. 20 and 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 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 §**??**) 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 §**??**) 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 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-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. 20 and 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-21).

From mark 1 on fig-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 $\times$ 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-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 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 `simlog` 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[1]. These are the so-called «XY-shuffling» countermeasure and the so-called «Z-remask» countermeasure:

[1] This might be implemented in future releases.

- 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-18 on p. 70 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` 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-22 on p. 73). 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-23 on p.73. 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-23 on p.73). 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-24 (p.74).

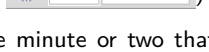
Two things can be noticed here. First, the four coordinates `[XY]R[01]` of points $\mathcal{R}_0$ and $\mathcal{R}_1$ match in both log files from the start, until computation reaches the scalar bit of weight 6. Remember indeed that in step **15** of the tutorial we programmed `zremask` = `4` in `ecc_customize.vhd`. Now since the IP starts parsing the scalar from its third least significant bit (viz. the bit of weight 2)[2] 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-25, p.74).

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-26 p.75.

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-27 p.75.

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
```

[2] The reason why the IP starts parsing the scalar only at its bit of weight 2 when computing $[k]\mathcal{P}$ is explained in §**??**, see in particular algorithm **??** and the discussion pertaining to it in p.**??**.

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-29 and 30 (p.76). 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-28 p.75.

☞ 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-31 and 32 (p.77) 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-33). This will ensure that the component instanciated in the TRNG part of the IP is actually the ES–TRNG design (see fig-34) and not the trivial simulation model. For our port of ES–TRNG please refer to §??. For the original publication of the ES–TRNG design by the COSIC research group of KU-Leuven, please see [**?**].

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-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 ⟨`sim/es_trng_sim.vhd`⟩, right-click on it and select *Remove file* from the contextual menu (see fig-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. 26) 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-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-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-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-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`[3].

Click OK and ignore the warning about the negative DQS delay on the DDR pins (fig-41). The Zynq processor in the block design has been upgraded with a few features (see fig-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`)

[3]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.

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-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-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-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-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-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 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-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-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-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[4] 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-51). What you should see now in the Vivado main window is illustrated in fig-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-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-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-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-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-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-58 and fig-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-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.

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[5]. 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:

[4] `https://digilent.com/reference/programmable-logic/arty-z7/reference-manual`

[5] This is a guarantee provided by Xilinx on their dual-clock BlockRAMs.

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 [**?**]) 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-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`[6] primitive named `bx2`, whose input clock port is indeed connected to a signal named `ro2out` (see fig-62) which is indeed driven by the output `O` of another instance, named `ro2_0`, of a `LUT3` primitive (see fig-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.

[6] 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.

☞ 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-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-65). These errors are related to the critical warnings we previously discussed about registers `r.nndyn.*`. As illustrated on fig-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-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-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-68 (whose colors are defined on fig-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-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-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-72).

35 Choose *Create Application Project* (fig-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-74). Leave the *Generate Boot Components* box checked, click *Next*. For the name of the application project choose `ecc-test-stdalone` (fig-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-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-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. 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[7]. You can check this in the settings of your guest machine, in the USB section. You should see something like on figure 83. Otherwise you'll have to first shutdown your VM, switch the settings to USB3 and run it again.

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**

[7] The point of installing the Extension Pack is precisely that it brings with it, among other things, the support for USB-3.

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-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-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-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
```

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 the VirtualBox manager window and open the settings for the virtual machine. Take a look at the USB section: now the USB-2 and USB-3 options should be available (fig-83). Select option `USB 3.0 (xHCI) Controller` and click OK, then reboot 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 84. This is to allow the board to be configured in JTAG mode.

Now plug the board into your machine. 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 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` and reboot the virtual machine.

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 inside the VM 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-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-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-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 Open a new terminal and type `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-89). Vitis/Eclipse builds the BSP and then the Hello World executable. Once this is done (fig-90), right-click again on project name `ecc-test-stdalone` in the *Explorer* tab and go to *Run As* ▶ *Run Configurations* (fig-91). In the *Run Configurations* window that opens click on *Single Application Debug* (fig-92) which should activate the top-left button  . Click on that button and in the new visual directly click on *Run* (fig-93). A pop-up window opens with a progress bar (fig-94) and after a few seconds you should see the message string from the `printf` appearing in the `screen` console (fig-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 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 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
```

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-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 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 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[8] (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`):

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 101. You can notice furthermore on the `screen` console (fig-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.

[8]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*).

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-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-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-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 43 (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-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-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-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-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-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-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-104) browse to *Image Packaging Configuration* ▶ *Root filesystem type* and select `EXT4 (SD/eMMC/SATA/USB)` (see fig-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-114). Similarly navigate to *Filesystem Packages* ▶ *net* ▶ *netcat* and select `netcat` (fig-115).

For the SSH server go likewise to *Filesystem Packages* ▶ *misc* ▶ *packagegroup-core-ssh-dropbear* and select `packagegroup-core-ssh-dropbear` (fig-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-112 and fig-113):

```
earlycon console=ttyPS0,115200 clk_ignore_unused root=/dev/mmcblk1p2
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/mmcblk1p2`) 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-117) and possibly a microSD card to SD card adapter (fig-118). If you use that one, make sure the tiny mechanical *lock* switch on the adapter is released (see fig-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-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-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-122) and click *Apply*. The partitions are erased from the SD card (fig-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-124). Right-click on the first `fat32` partition, select *Manage Flags* and check the `boot` box (fig-125) then close. You should get what is shown on figure 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/mmcblk0` and `/dev/mmcblk1` 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-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-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-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-130) and change position of jumper `JP4` as illustrated on fig-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-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[9].

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-133). 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 134.

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.

[9] 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}).

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 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 135) 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 136). 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, that 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.

Appendix C

Figures

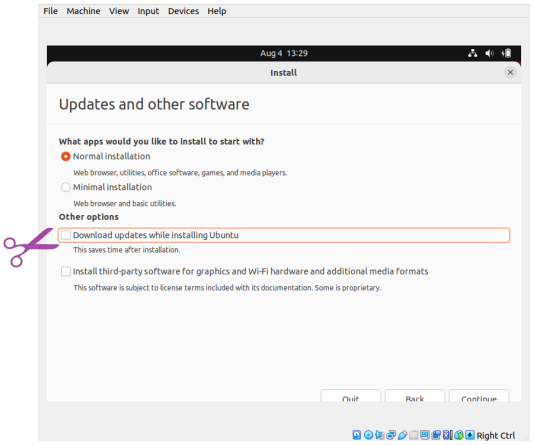


Figure 2

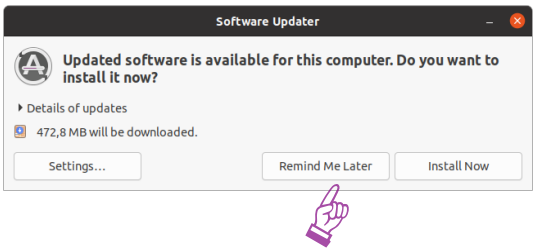


Figure 3

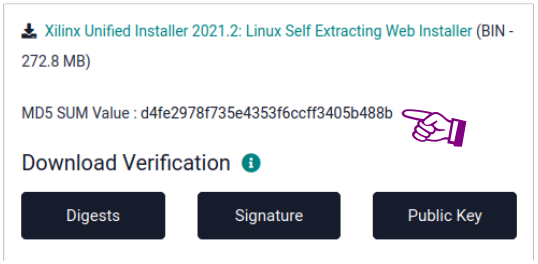


Figure 4

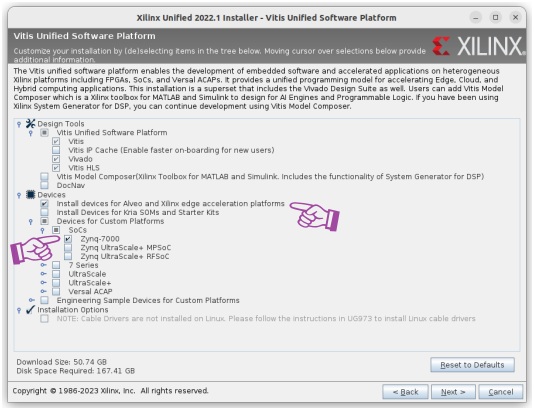


Figure 5

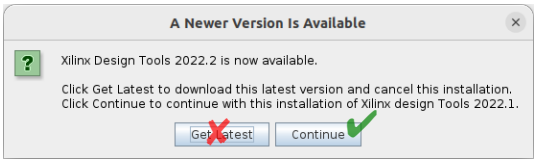


Figure 6

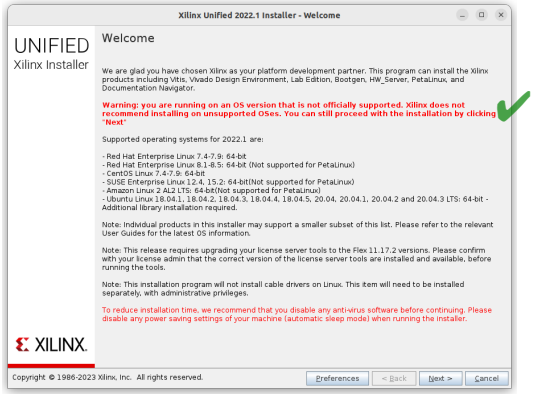


Figure 7

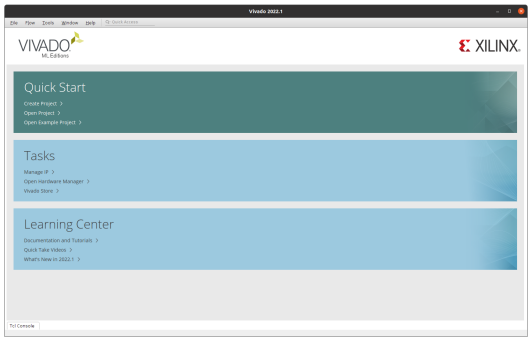


Figure 8

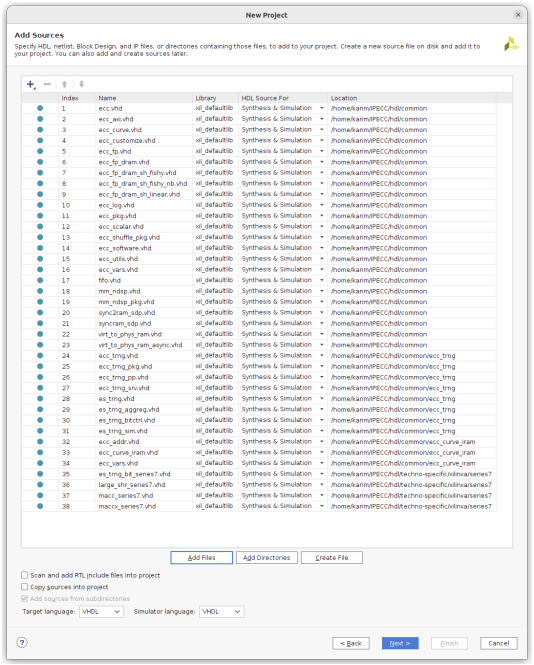


Figure 9

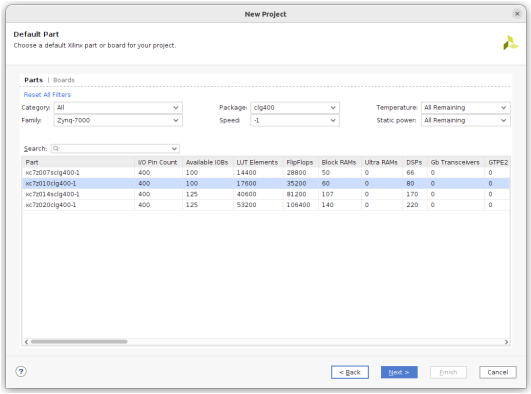


Figure 10

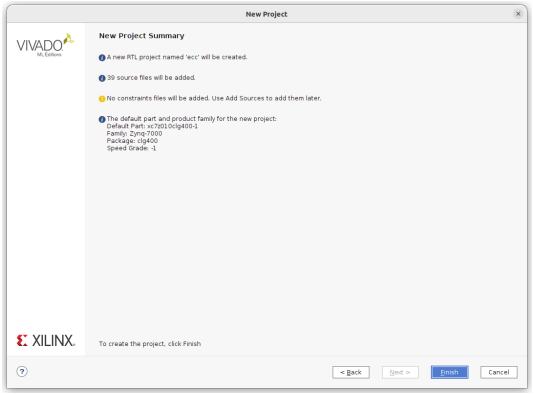


Figure 11

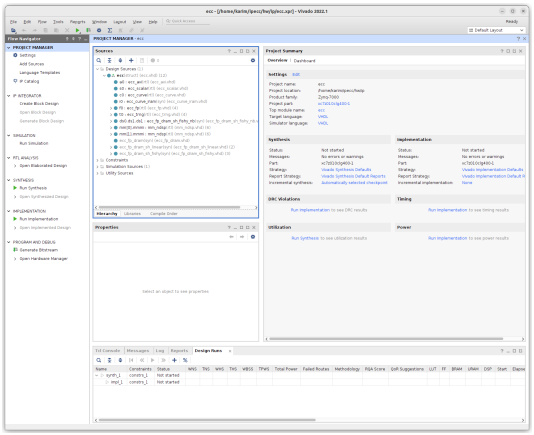


Figure 12

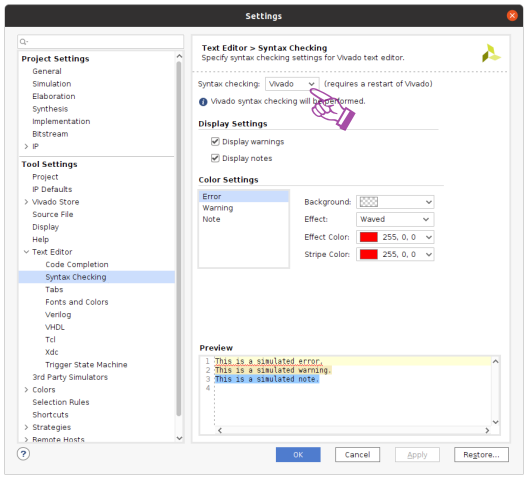


Figure 13

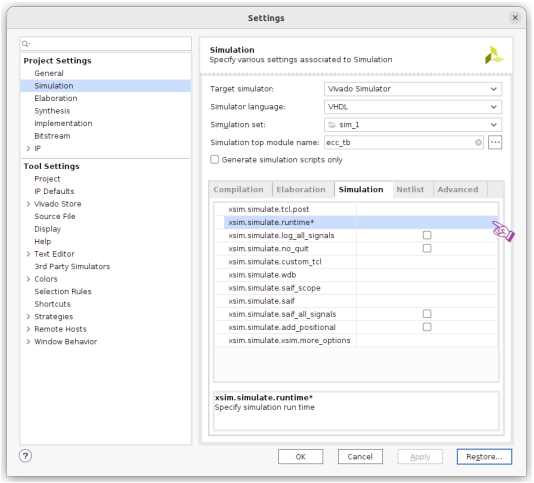


Figure 14

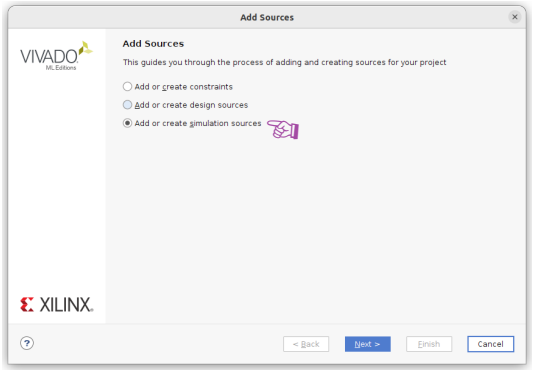


Figure 15

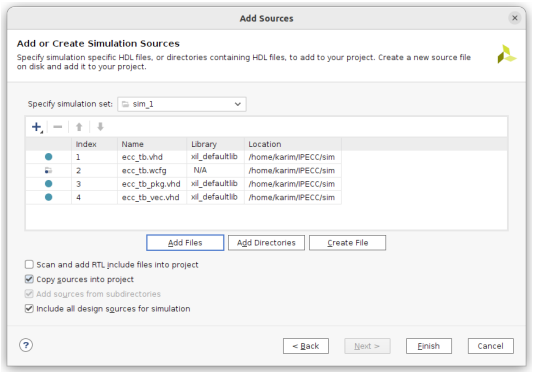


Figure 16

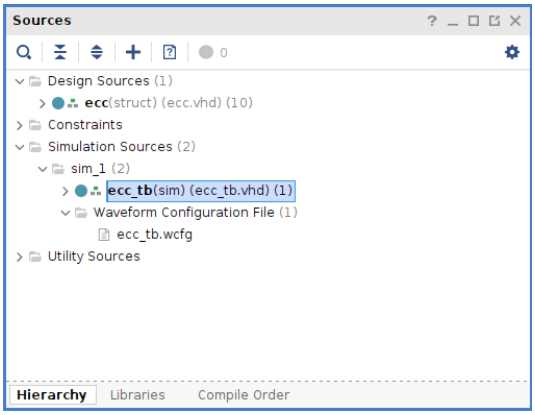


Figure 17

```
  -- *********************************
  -- Start of: user-editable parameters
  -- *********************************
  -- Please refer to the in-file documentation of parameters below (after the
  -- package specification), where parameters are described in the same order
  -- as they appear in the code hereafter.
  constant nn : positive := 256;
  constant nn_dynamic : boolean := TRUE;
  type techno_type is (spartan6, virtex6, series7, ultrascale, ialtera, asic);
  constant techno : techno_type := series7; -- set a 'techno_type' value
  -- -------------------------------
  -- Performance related parameters
  -- -------------------------------
  -- multwidth is only used if 'techno' = 'asic'
  -- (otherwise its value has no meaning and can be ignored)
  constant multwidth : positive := 32; -- 32 seems fair for an ASIC default
  constant nbmult : positive range 1 to 2 := 2;
  -- parameter nbdsp below is range-constrained because it must be >=2
  constant nbdsp : positive range 2 to positive'high := 6;
  constant sramlat : positive range 1 to 2 := 1;
  constant async : boolean := TRUE;
  -- ------------------------------------------------
  -- Side-channel countermeasures related parameters
  -- ------------------------------------------------
  constant debug : boolean := FALSE; -- FALSE = highly secure, TRUE = highly not
  constant blinding : integer := 0;  -- 96 seems fair for size of blinding rnd
  constant shuffle : boolean := TRUE; -- memory shuffling
  type shuftype is (none, linear, permute_lgnb, permute_limbs);
  constant shuffle_type : shuftype := permute_lgnb; -- set a 'shuftype' value
  constant zremask : integer := 4; -- quite arbitrary
  -- ------------------------
  -- TRNG related parameters
  -- ------------------------
  constant notrng : boolean := TRUE; -- set to TRUE for simu, to FALSE for syn
  constant nbtrng : positive := 4;
  constant trngta : natural range 1 to 4095 := 32;
  constant trng_ramsz_raw : positive := 4; -- in kB
  constant trng_ramsz_axi : positive := 4; -- in kB
  constant trng_ramsz_fpr : positive := 4; -- in kB
  constant trng_ramsz_crv : positive := 4; -- in kB
  constant trng_ramsz_shf : positive := 16; -- in kB
  -- -------------
  -- Miscellaneous
  -- -------------
  constant axi32or64 : natural := 32; -- 32 or 64 only allowed values
  constant nblargenb : positive := 32;  -- change these two parameters only if
  constant nbopcodes : positive := 512; -- you really know what you're doing
  -- ---------------------------
  -- Simulation-only parameters
  -- ---------------------------
  constant simvecfile : string := "/tmp/ecc_vec_in.txt"
  constant simkb : natural range 0 to natural'high := 0; -- if 0 then ignored
  constant simlog : string := "/tmp/ecc.log";
  constant simtrngfile: string := "/tmp/random.txt";
  -- *******************************
  -- End of: user-editable parameters
  -- *******************************
```

Figure 18

```
== NEW CURVE #0
nn=21
p=0x1ce54b
a=0x0ec20f
b=0x1bb973
q=0x1ce256
== TEST [k]P #0.0
Px=0x00851a
Py=0x0a0e0f
k=0x1c0ac1
nbbld=14
kPx=0x0acc93
kPy=0x0e007f
== TEST P+Q #0.1
Px=0x0e58a7
Py=0x0b0eb7
Qx=0x07136d
Qy=0x032a06
PplusQx=0x0c11ee
PplusQy=0x07c882
== TEST [2]P #0.2
Px=0x0adc61
Py=0x14fae0
twoPx=0x1462de
twoPy=0x0cfb3a
== TEST -P #0.3
Px=0x0a3f63
Py=0x16043d
negPx=0x0a3f63
negPy=0x06e10e
== TEST isPoncurve #0.4
Px=0x041730
Py=0x19233e
false
== TEST isP==Q #0.5
Px=0x01abef
Py=0x040c8d
Qx=0x0e2427
Qy=0x037216
false
== TEST isP==-Q #0.6
Px=0x123a44
Py=0x18f264
Qx=0x123a44
Qy=0x03f2e7
true
```

Figure 19

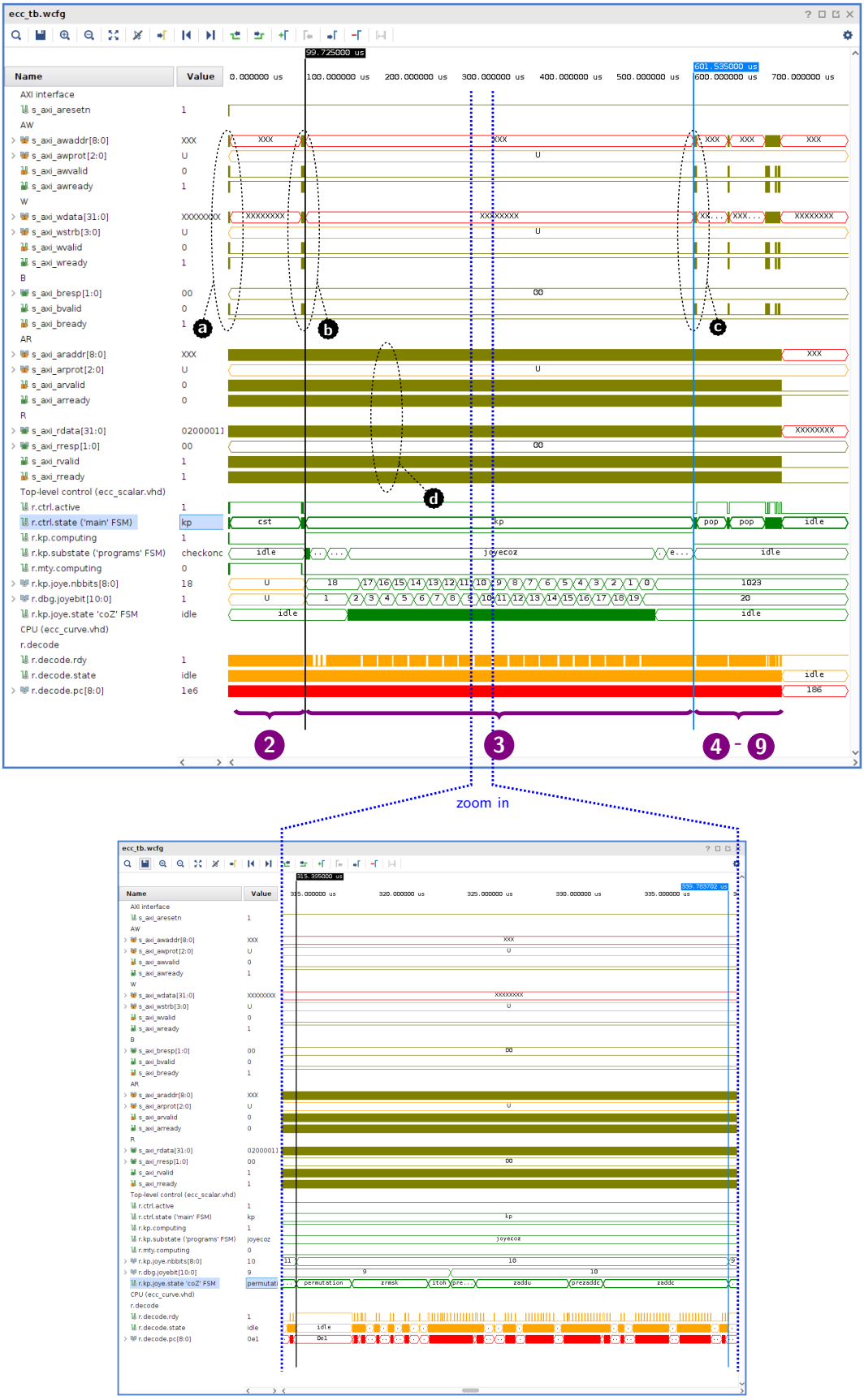


Figure 20: Simulation waveform of our input test-vectors file. Lower half of the figure shows a detailed view of the processing of one bit of the scalar (the bit of weight 10).

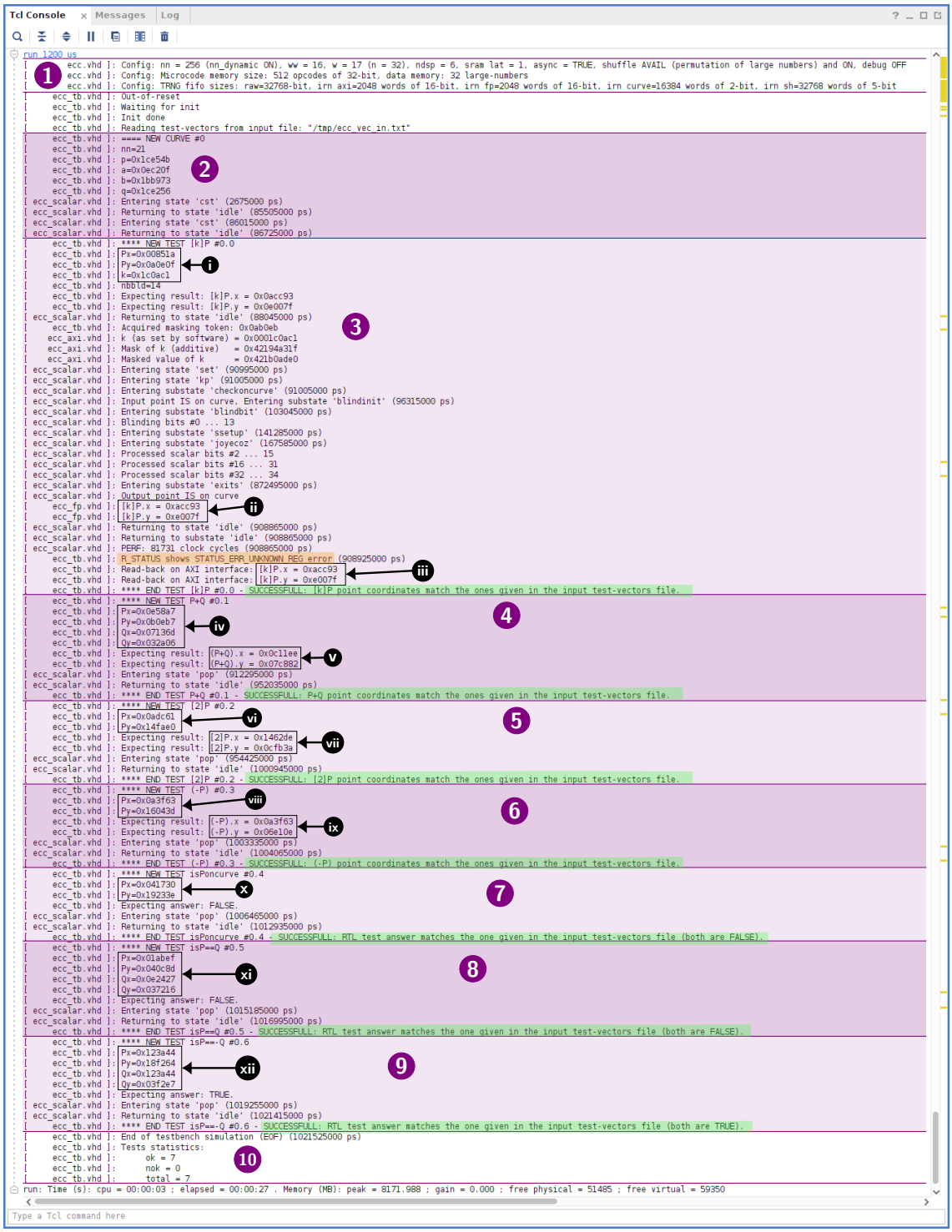


Figure 21: Simulation console log of our input test-vectors file.

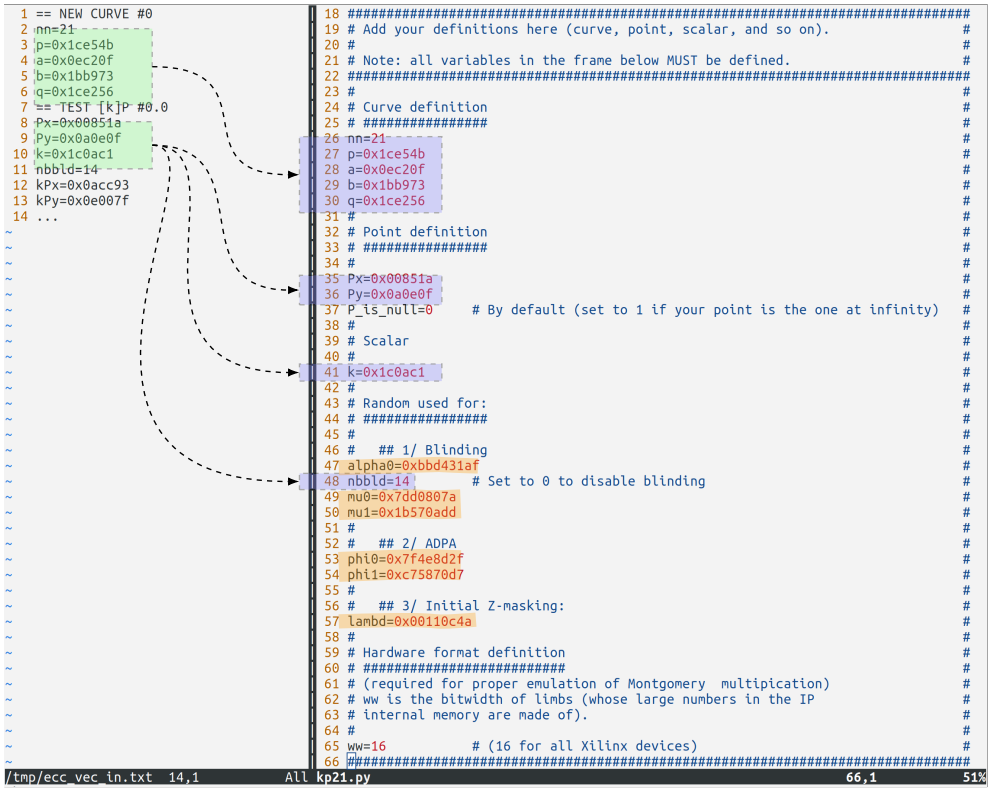


Figure 22: Copy-paste cryptographic values for curve and points from the test-vector file (in green) into the Python script (in blue). Remaining parameters (in orange) are random values we can quickly extract from the RTL log file (`/tmp/ecc.log`) as seen on fig-23 just below.

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 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-22 just above.

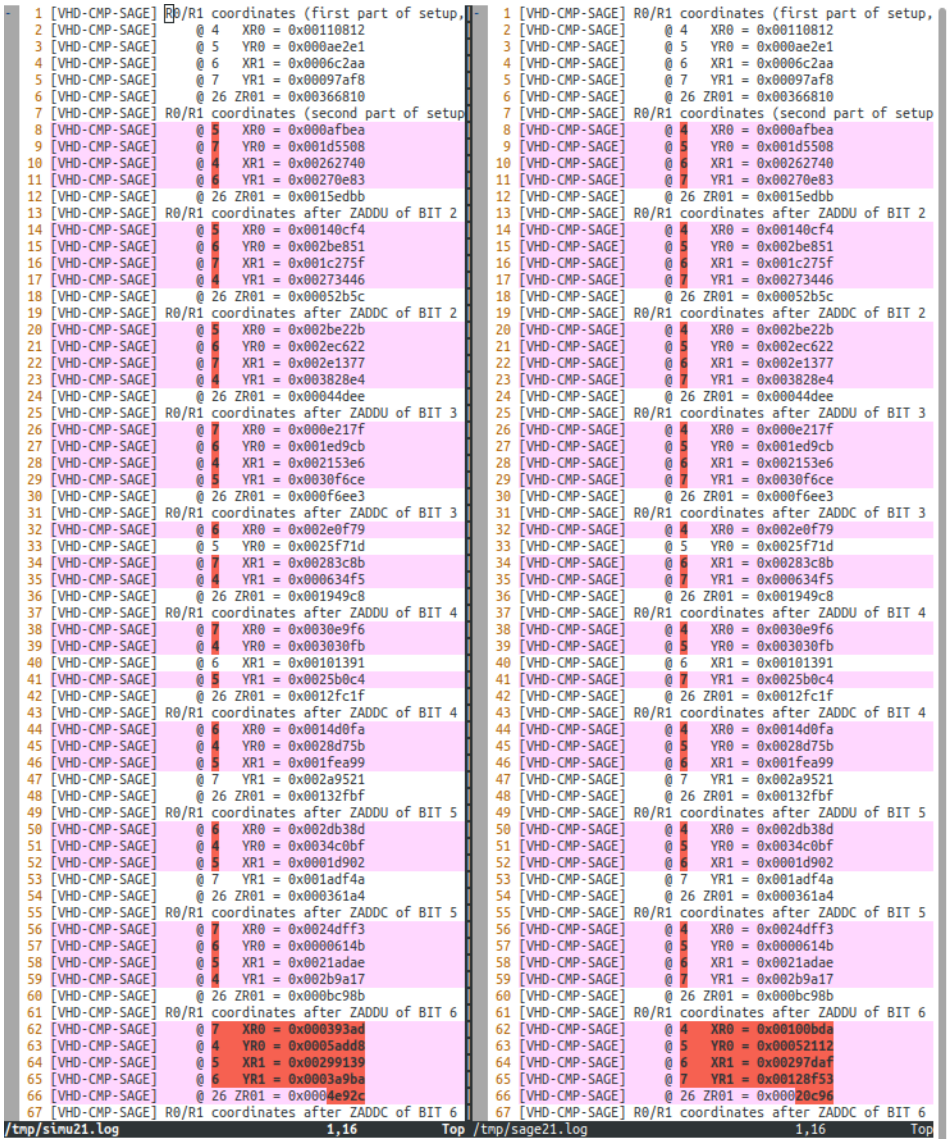


Figure 24: Differences between the RTL simu log (on the left) and the *Sage* script (on the right) for $[k]\mathcal{P}$ operation. Coordinates X,Y,Z of $\mathcal{R}_0$ and $\mathcal{R}_1$ are identical from the start of computation until the bit of weight 6 of the scalar is hit (the «Z-remask» countermeasure 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).

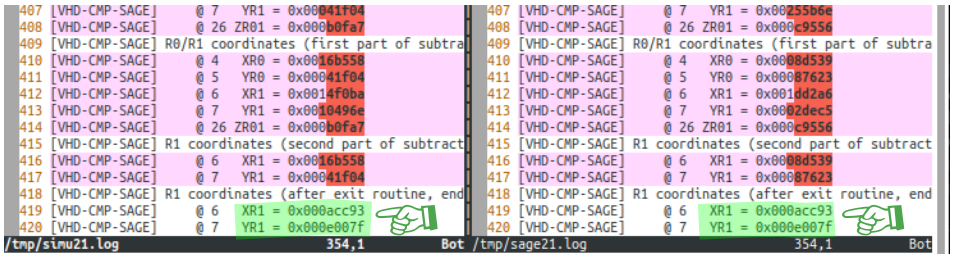


Figure 25: Despite the randomizations taking place all along the computation (see fig-24 just above) the final coordinates of $\mathcal{R}_1 = [k]\mathcal{P}$ naturally match.

```
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 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 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 shoftware driver would do at runtime. See also fig-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 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.

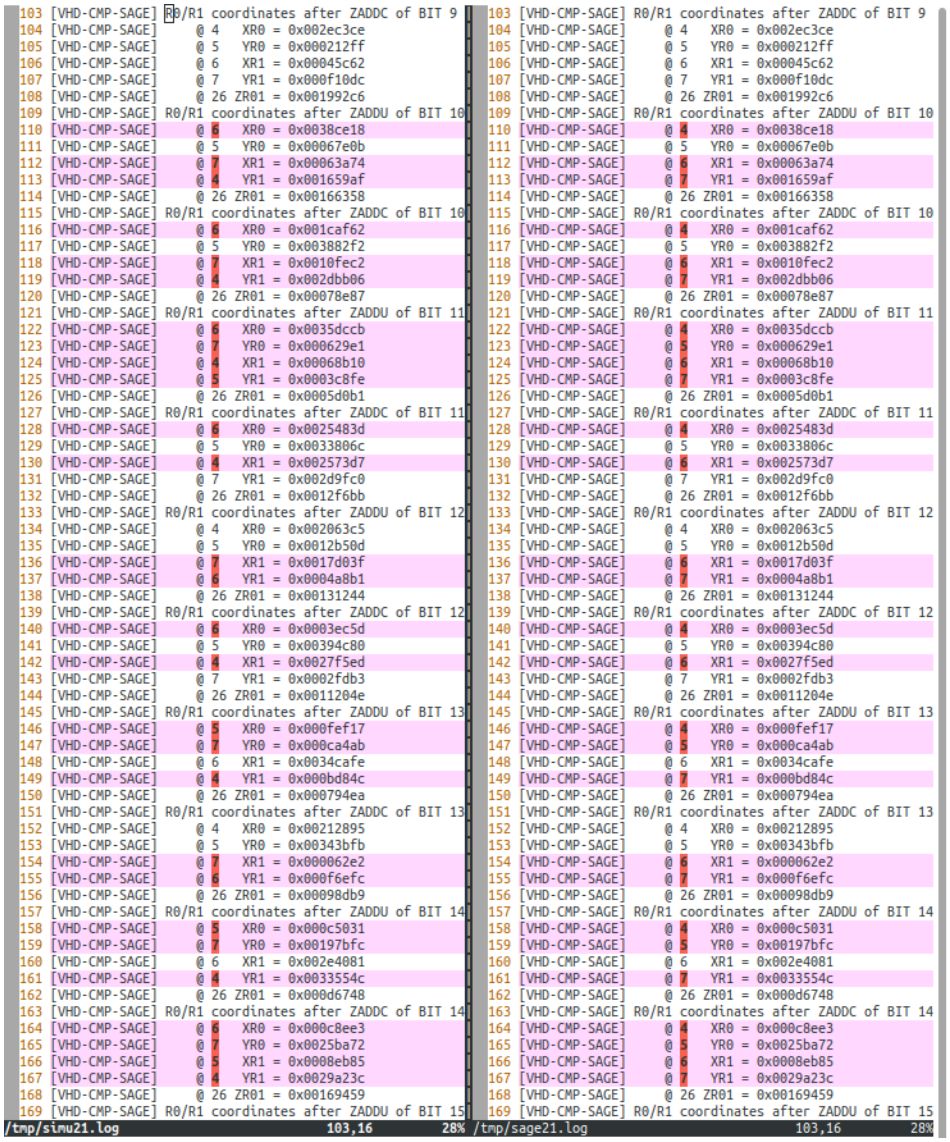


Figure 29: Differences between the RTL simu log (on the left) and the *Sage* script (on the right) for $[k]\mathcal{P}$ operation after the «Z-remask» countermeasure has been disabled. The coordinates match all along between the two logs, only physical addresses of coordinates of points $\mathcal{R}_0$ and $\mathcal{R}_1$ stay randomized during the RTL simulation.

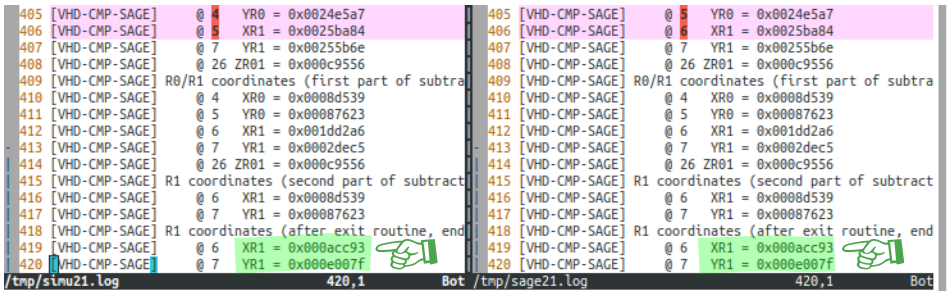


Figure 30: Once the scalar parsing loop is complete and before entering conditional subtraction of the base point $\mathcal{P}$ the physical addresses of coordinates of $\mathcal{R}_0$ and $\mathcal{R}_1$ and restored back to their static deterministic values. For explanation on the final conditional subtraction at the end of $[k]\mathcal{P}$ computation, please refer to §**??**, in particular algorithm **??** and its related discussion.

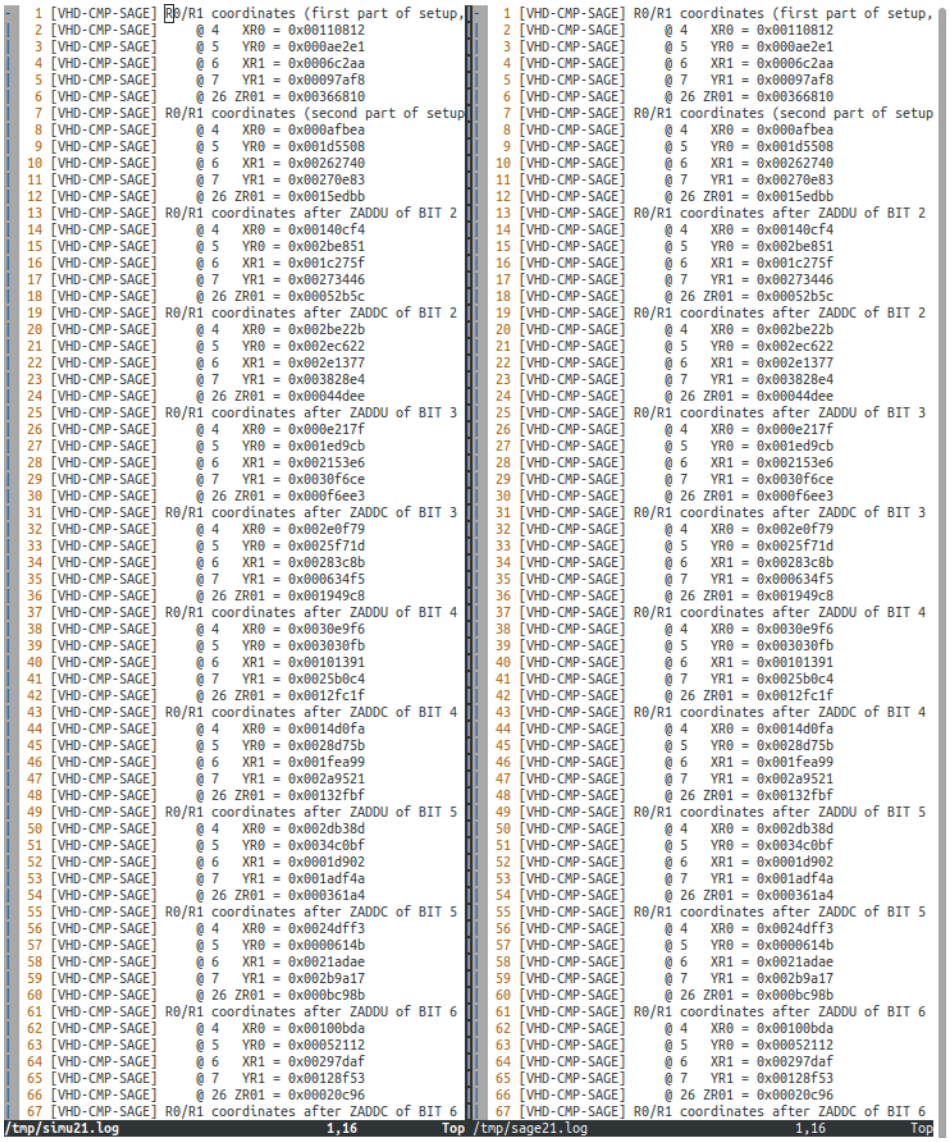


Figure 31: Once both countermeasures «Z-remask» and «XY-shuffling» have been removed, content of RTL simu log (on the left) and *Sage* script log (on the right) fully match.

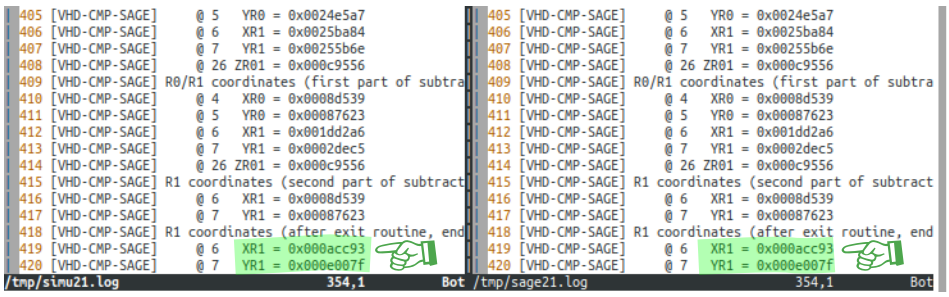


Figure 32

```
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 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-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 34: VHDL component `es_trng.vhd` corresponds to the true physical random generator used in IPECC. It is named after the design ES–TRNG originally published by KU-Leuven, see §?? and [?].

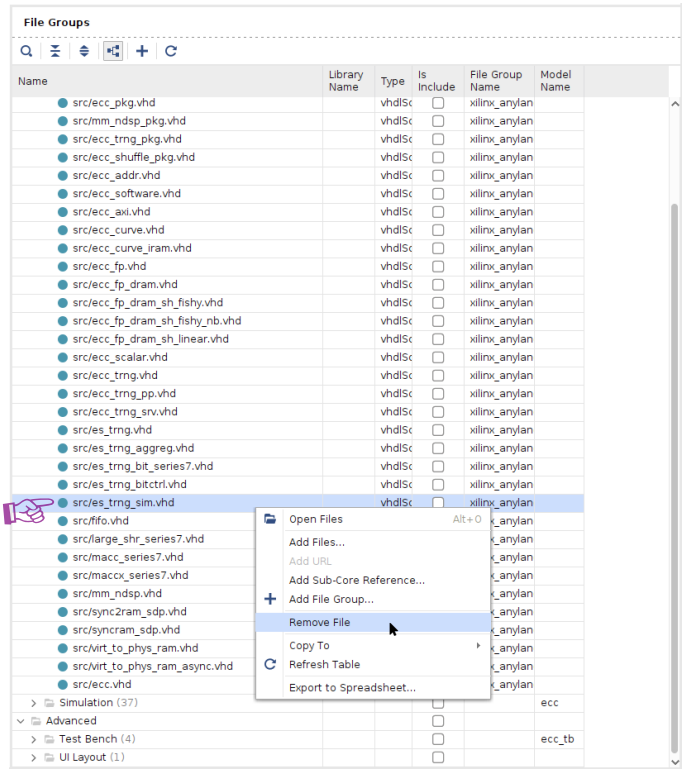


Figure 35: Exported files for the packaging of the IP must not include the simulation model `ecc_trng_sim.vhd`.

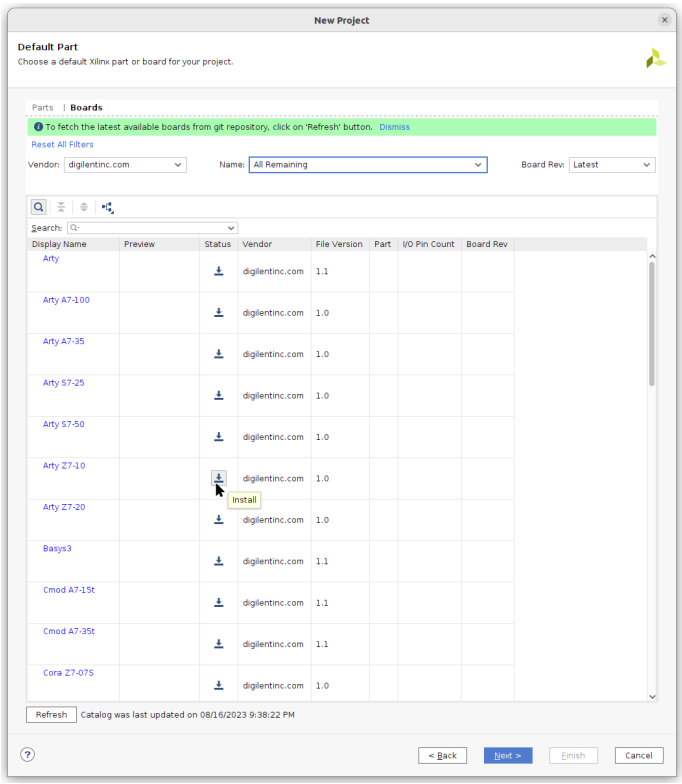


Figure 36

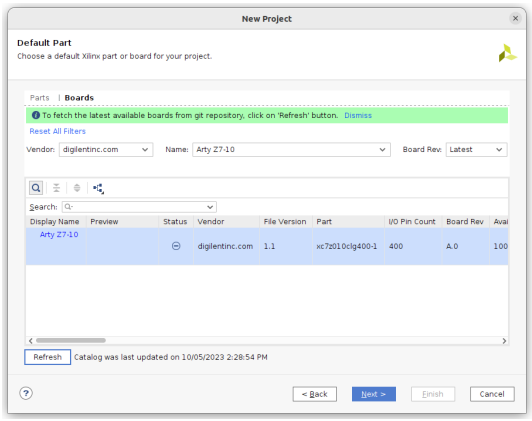


Figure 37: Make sure the board Arty Z7 10 is selected! (its line must be highlighted)

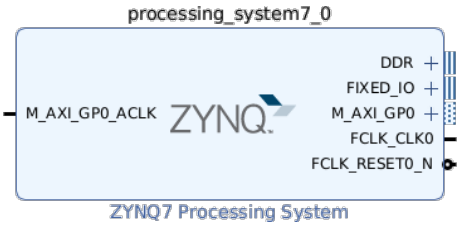


Figure 38

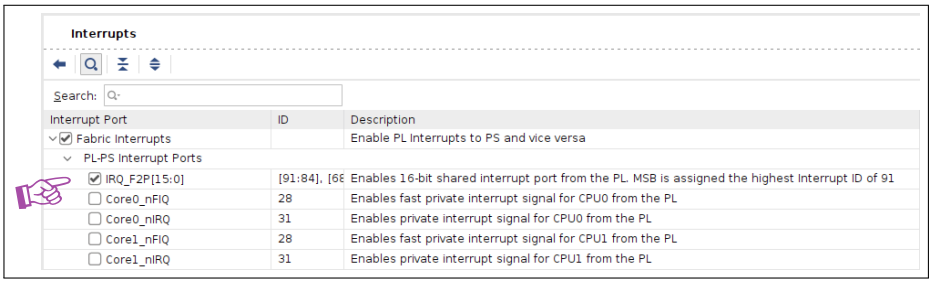


Figure 39

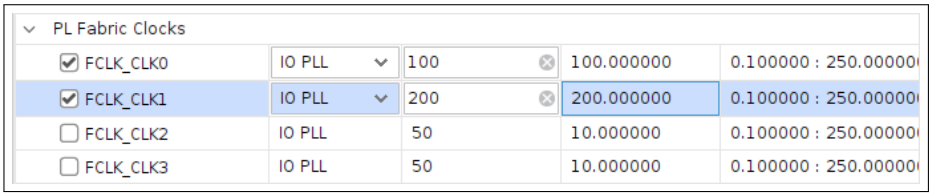


Figure 40

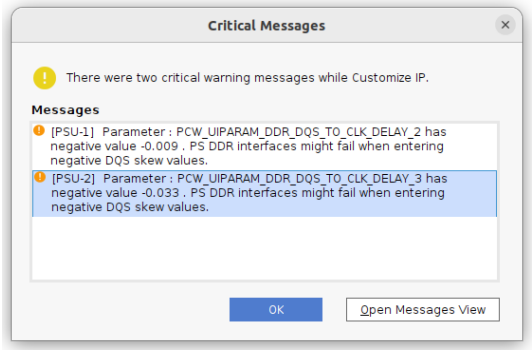


Figure 41

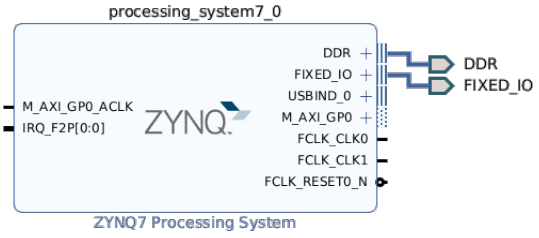


Figure 42

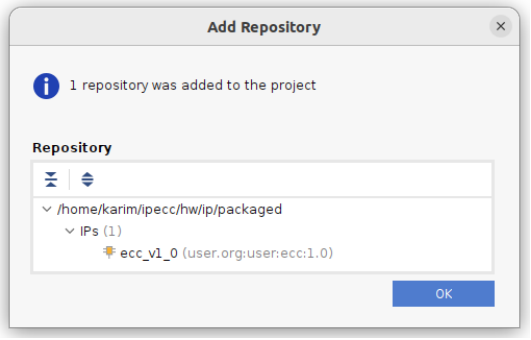


Figure 43

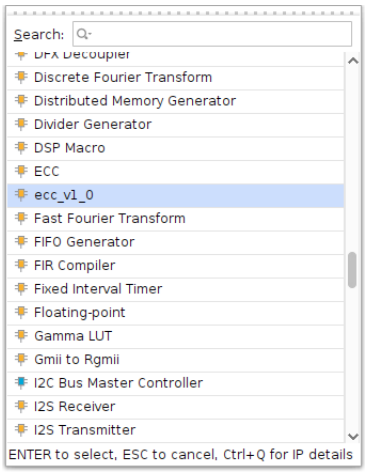


Figure 44

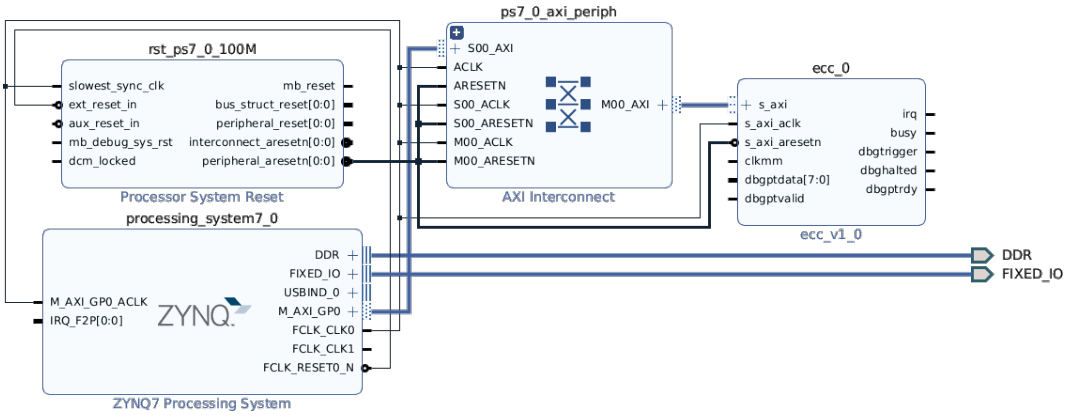


Figure 45

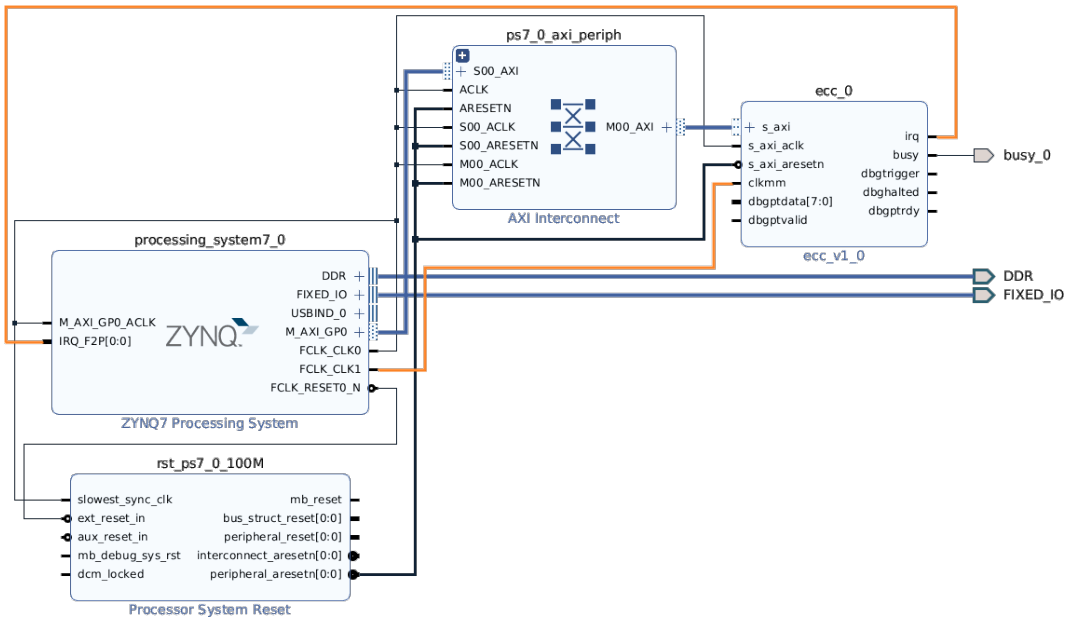


Figure 46

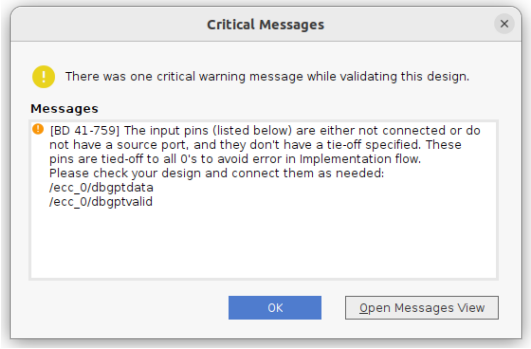


Figure 47

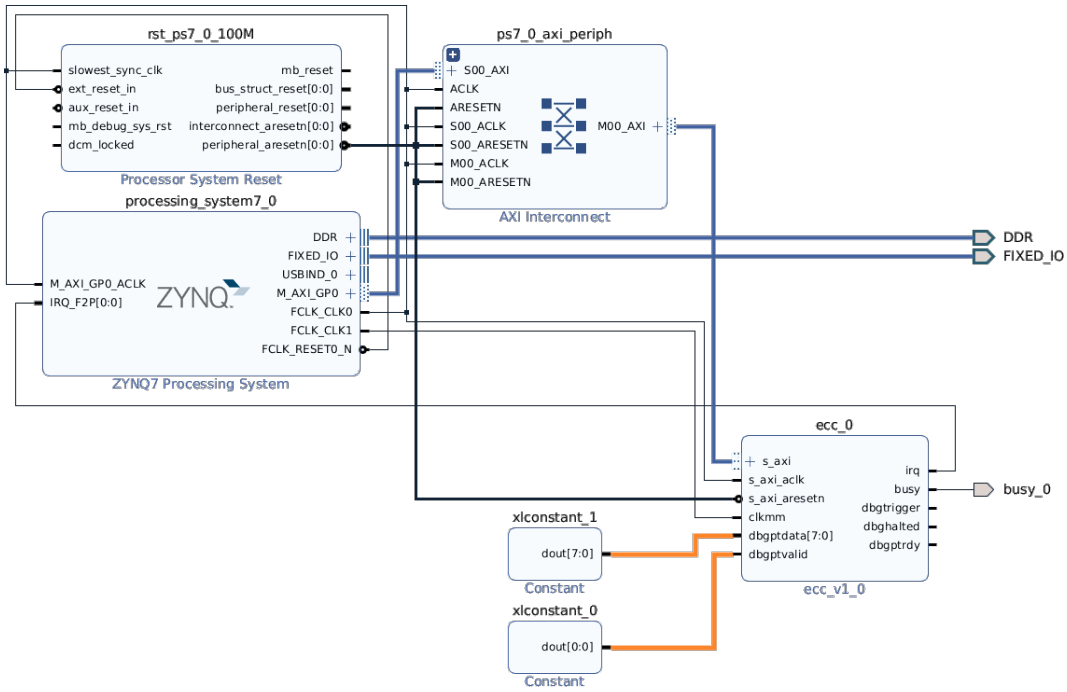


Figure 48

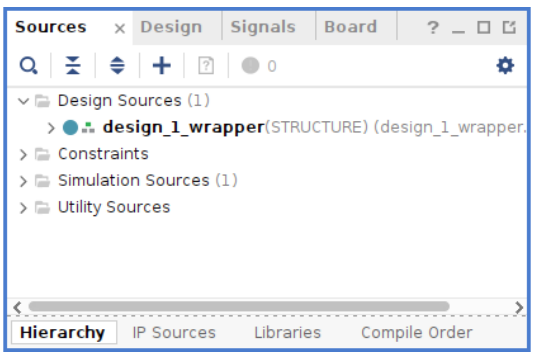


Figure 49

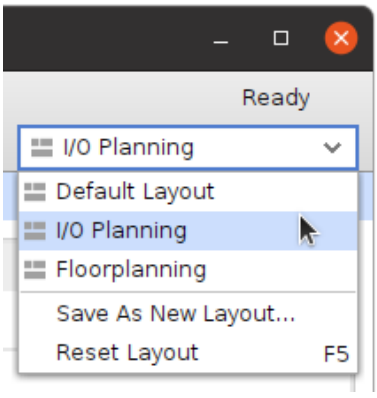


Figure 50

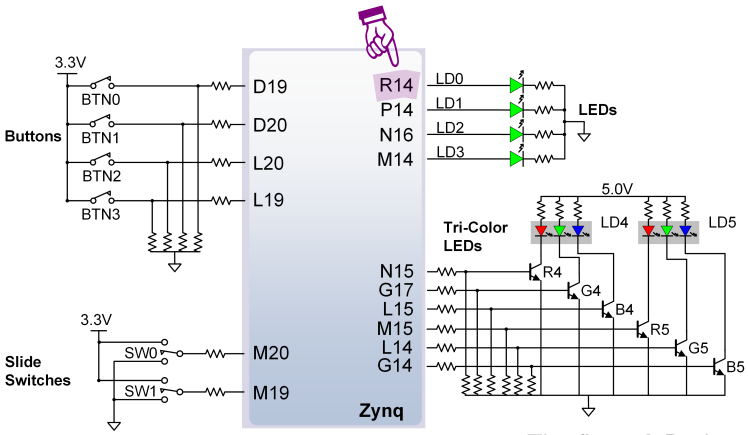


Figure 51

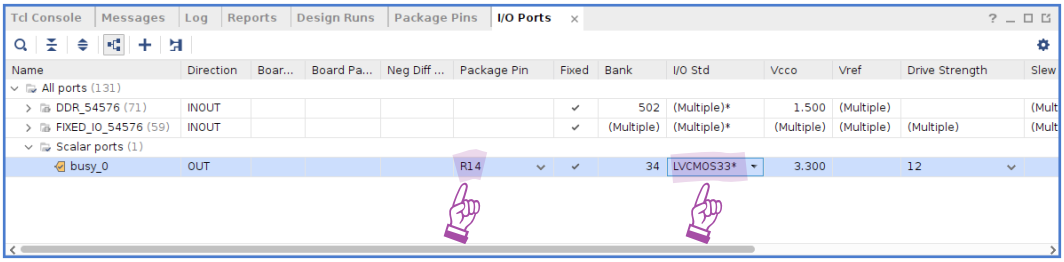


Figure 52

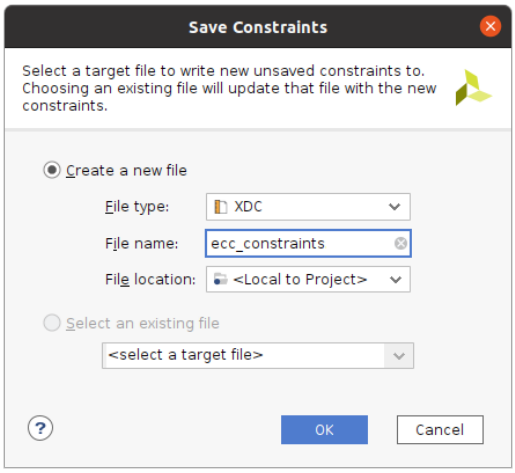


Figure 53

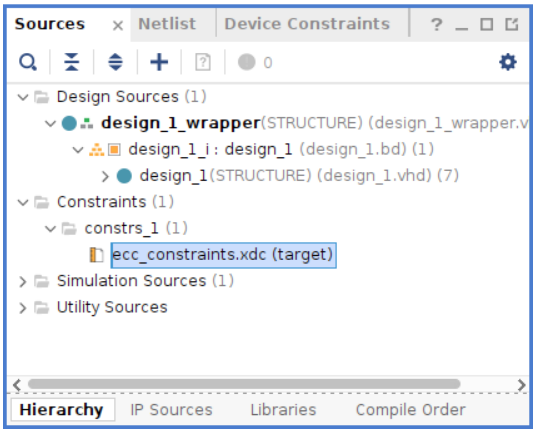


Figure 54

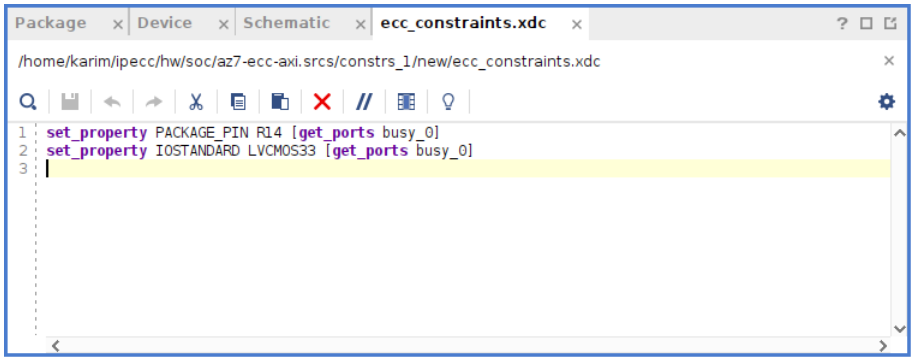


Figure 55

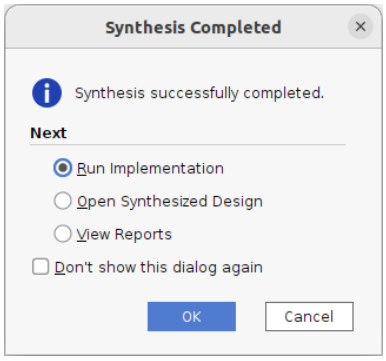


Figure 56

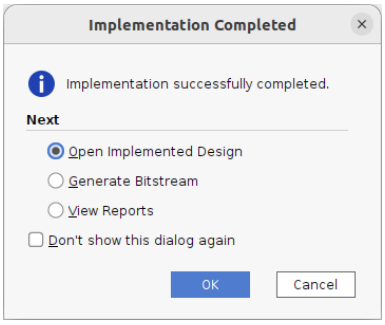


Figure 57

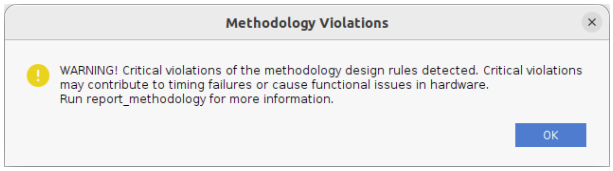


Figure 58

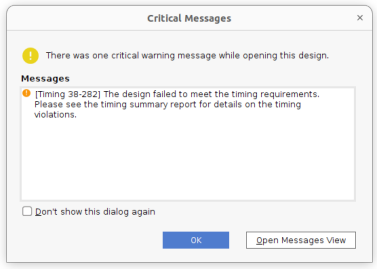


Figure 59

```
+-----------+------------------+-----------------------------------------------+-----------
| 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 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 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 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 63

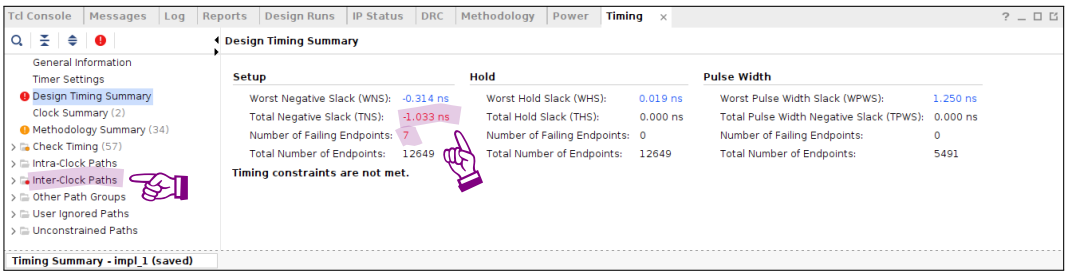


Figure 64

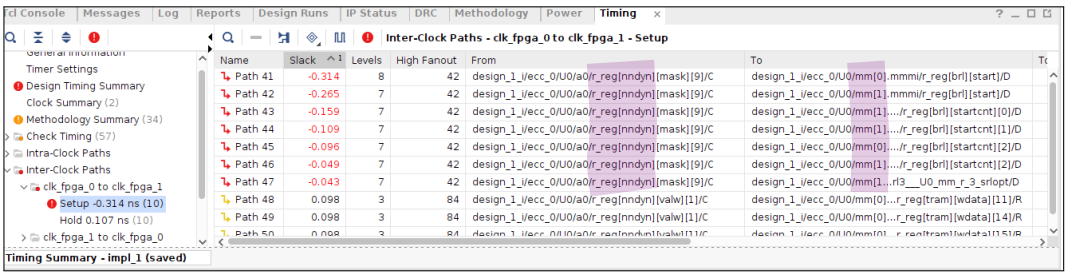


Figure 65

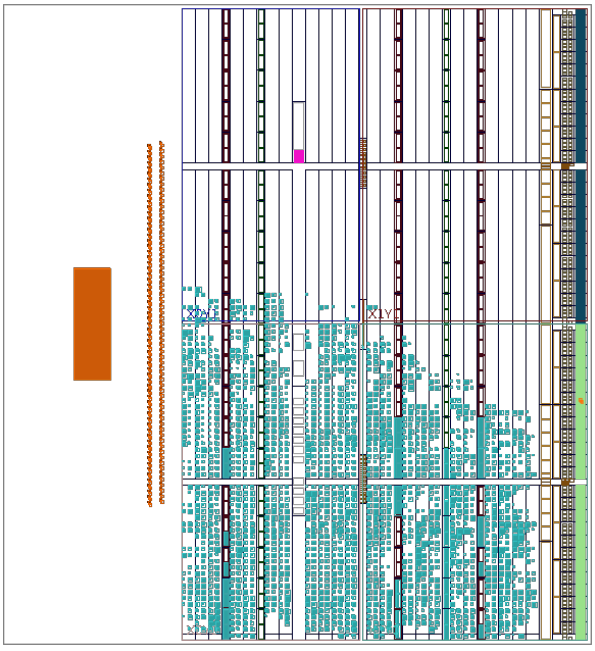


Figure 66: Layout of the design. Cells colored in blue indicate occupied logic ressources.

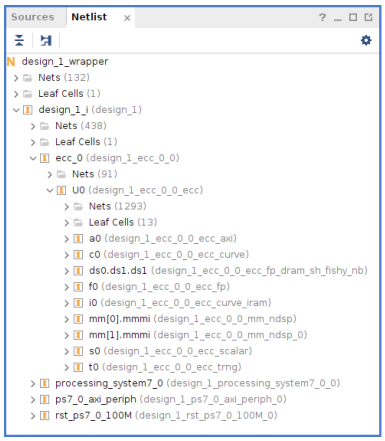


Figure 67

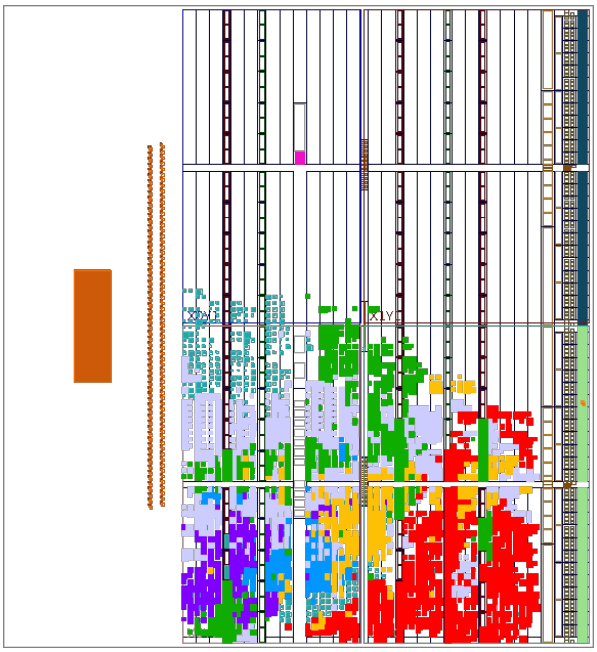


Figure 68

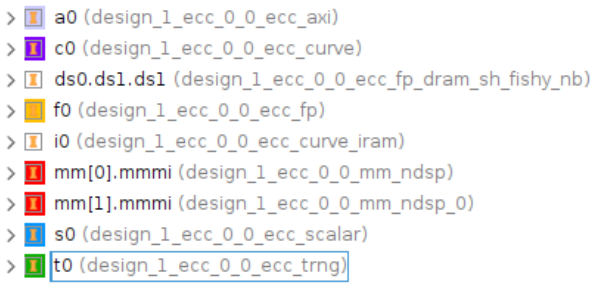


Figure 69: Color legend for figure 68.

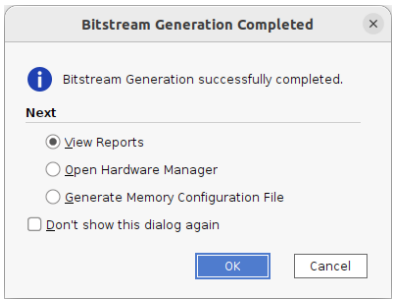


Figure 70

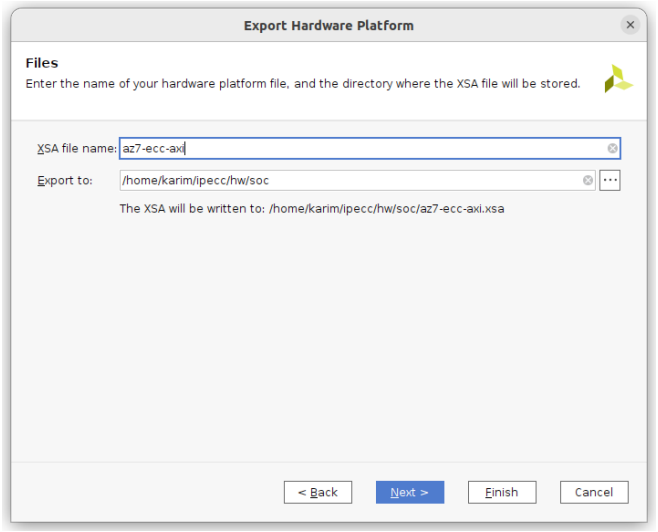


Figure 71

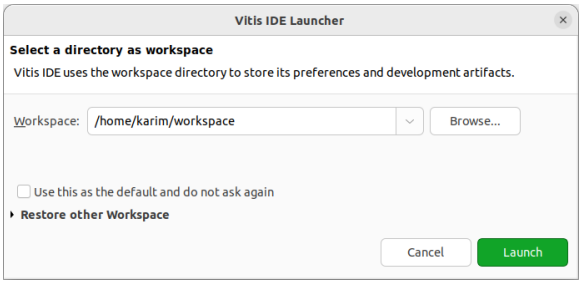


Figure 72

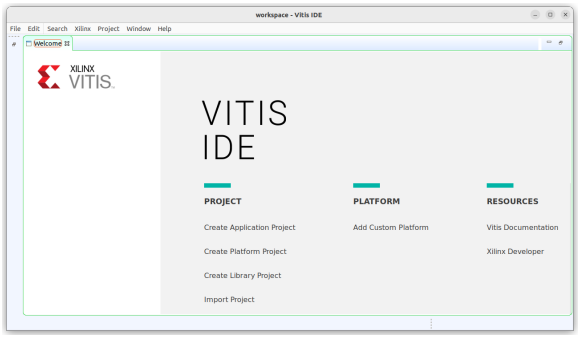


Figure 73

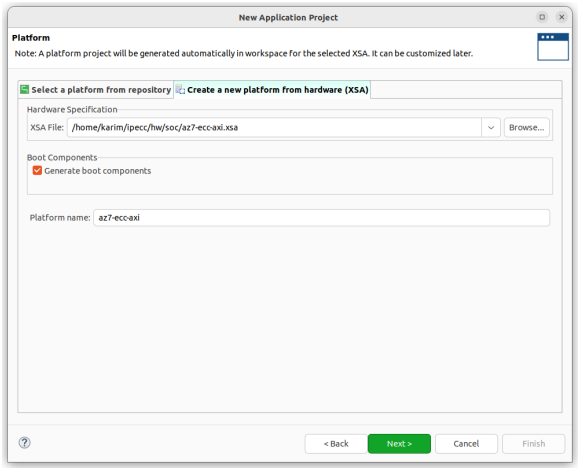


Figure 74

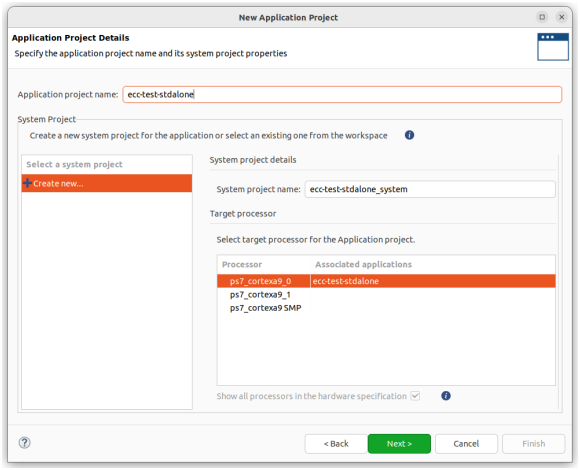


Figure 75

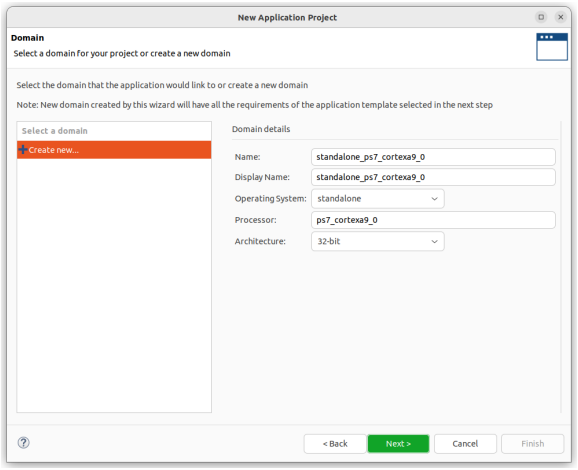


Figure 76

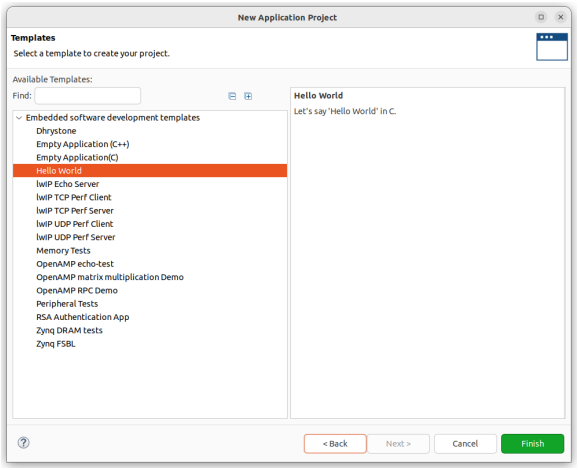


Figure 77

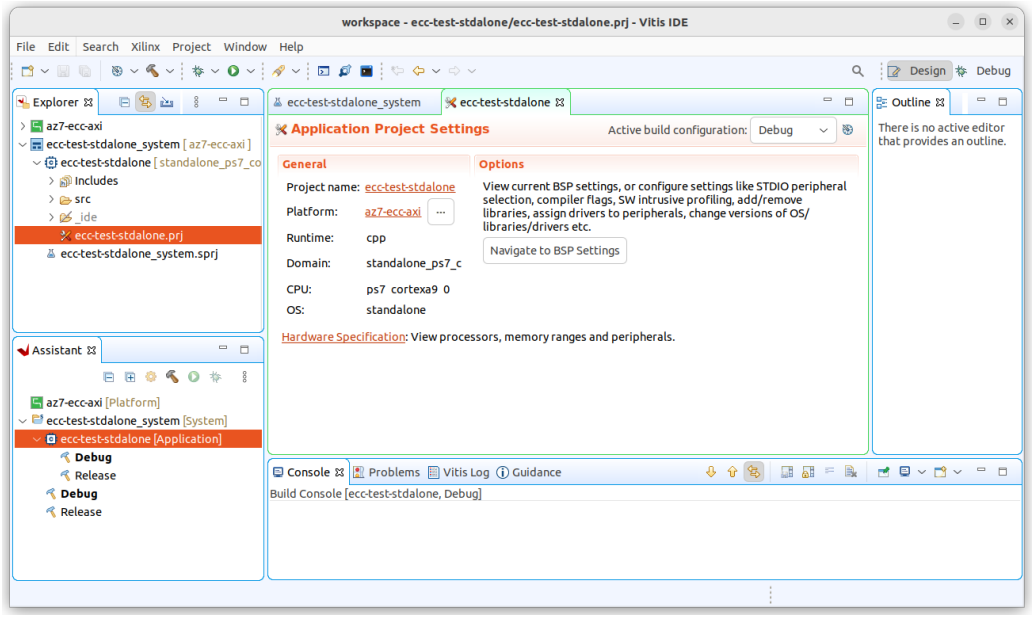


Figure 78

```
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 79

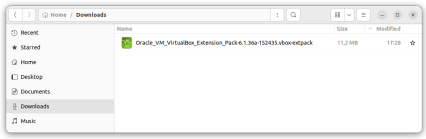


Figure 80

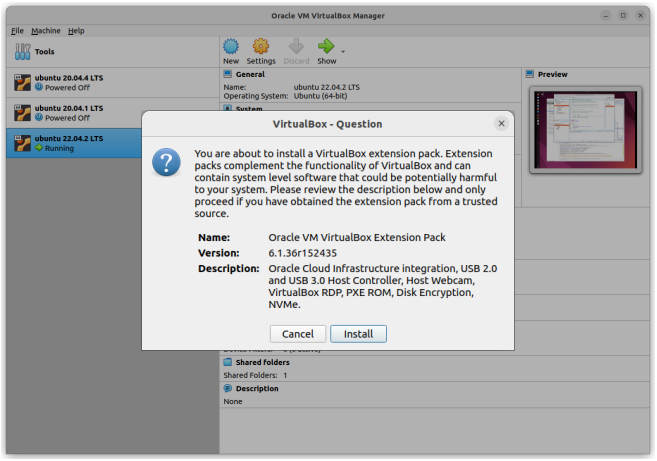


Figure 81

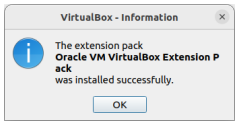


Figure 82

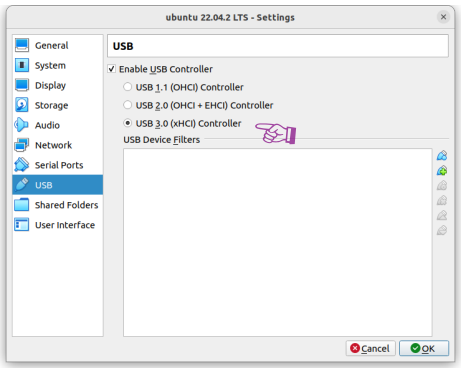


Figure 83

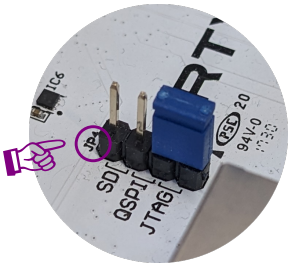


Figure 84

```
[ 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 85

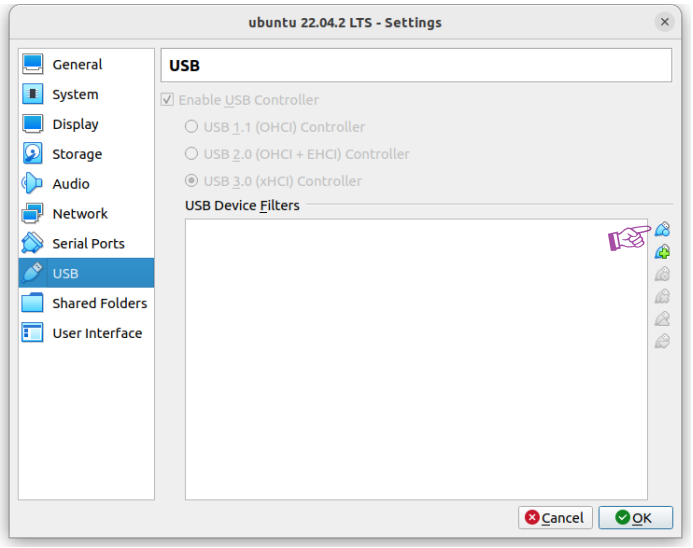


Figure 86

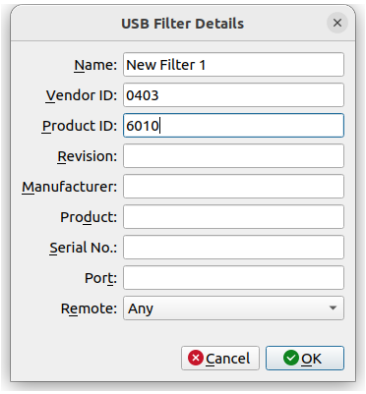


Figure 87

```
$ 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 88

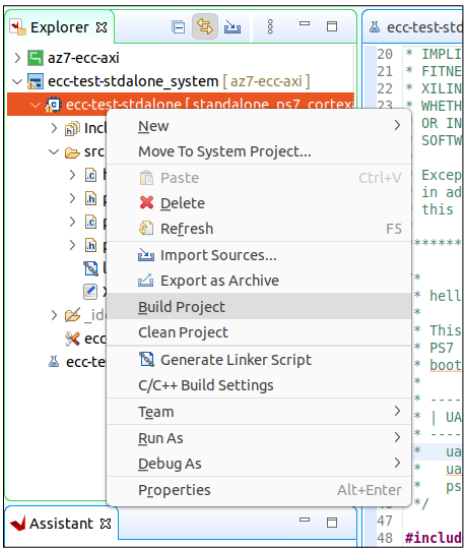


Figure 89

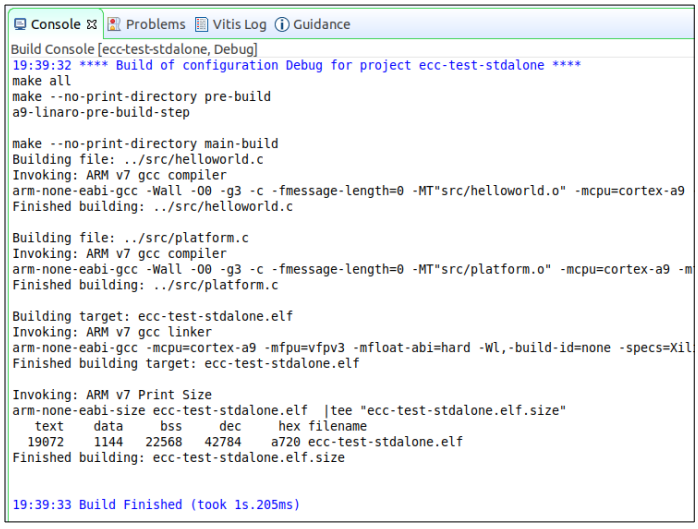


Figure 90

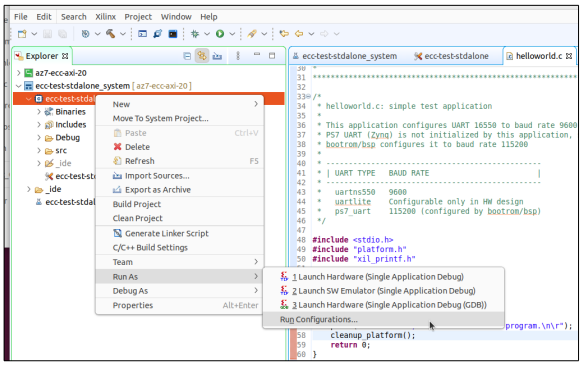


Figure 91

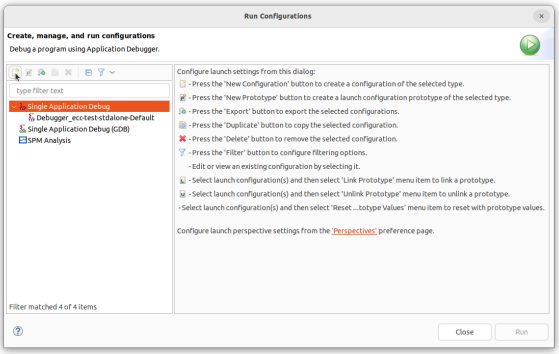


Figure 92

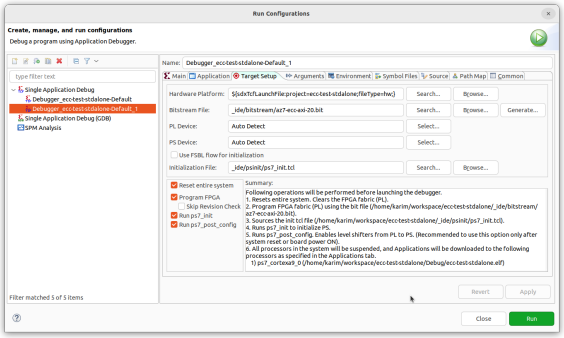


Figure 93

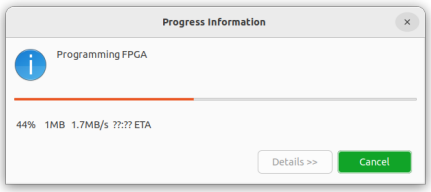


Figure 94

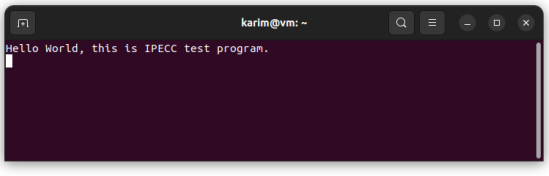


Figure 95

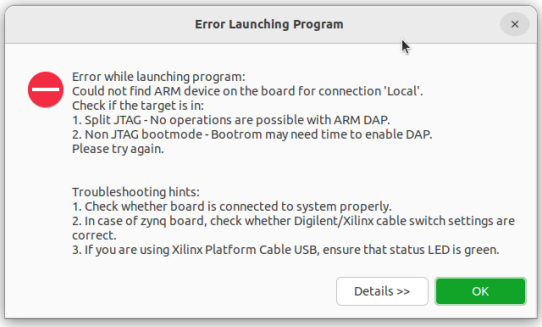


Figure 96

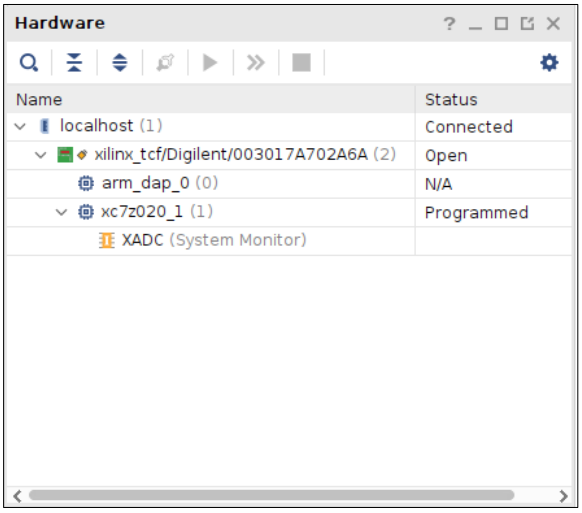


Figure 97

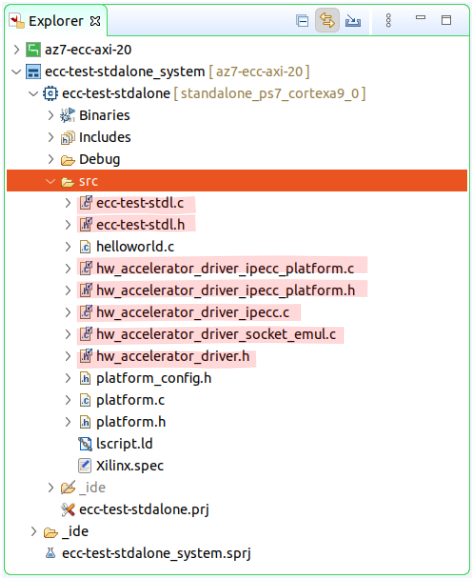


Figure 98

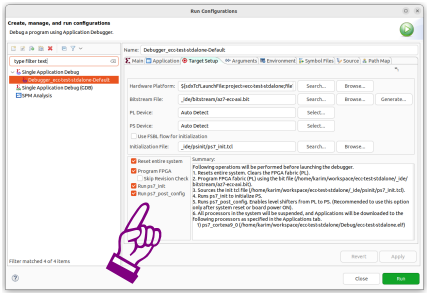


Figure 99

```
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 100

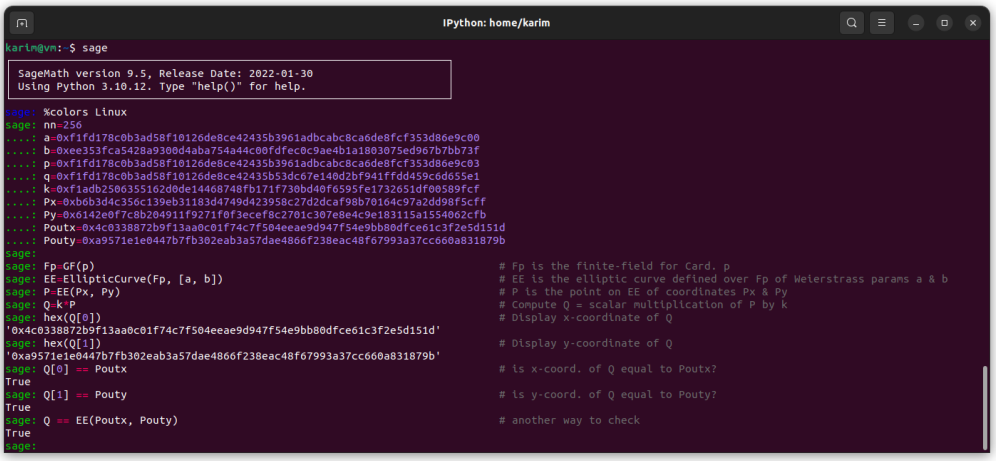


Figure 101

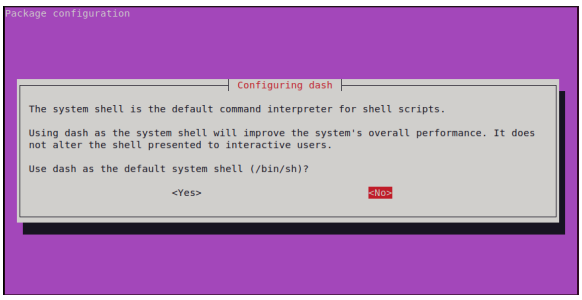


Figure 102

Figure 103

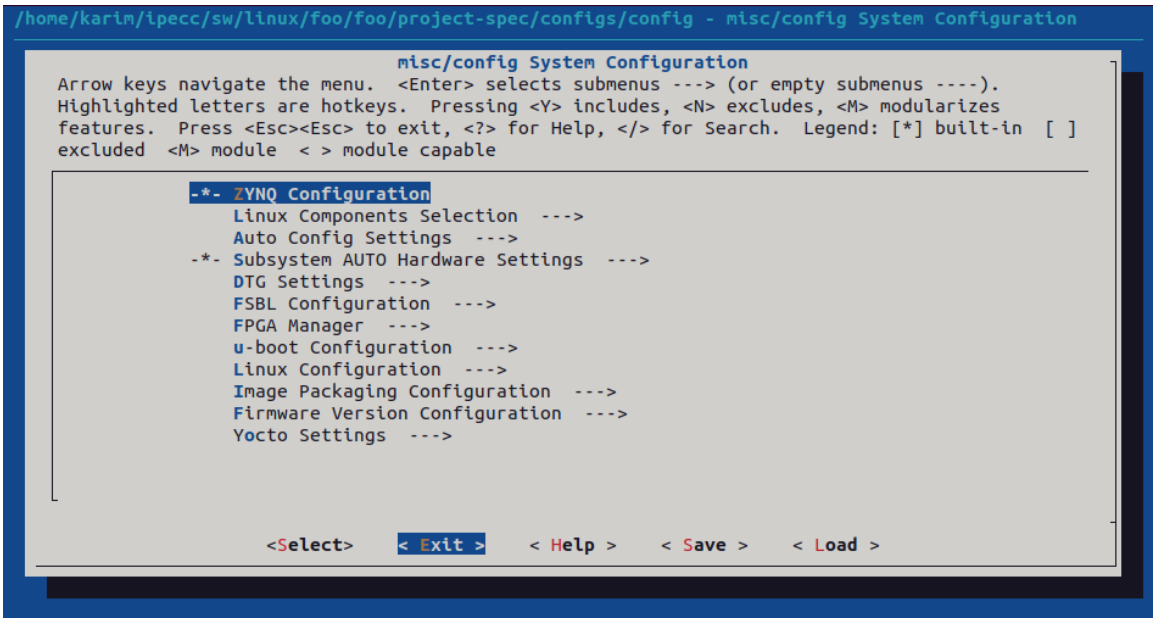


Figure 104

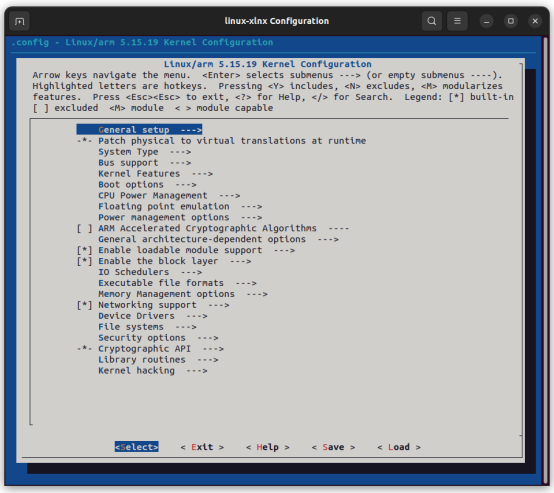


Figure 105

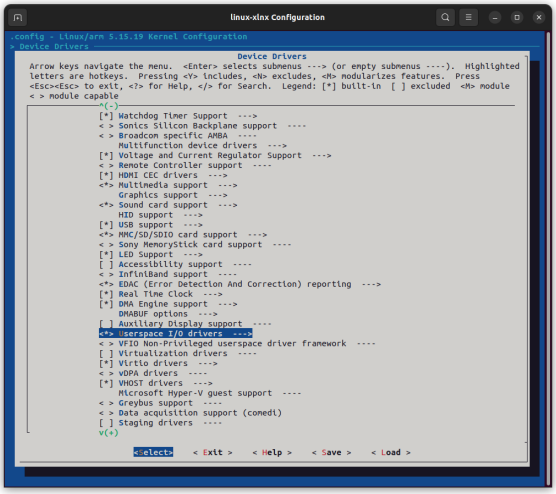


Figure 106

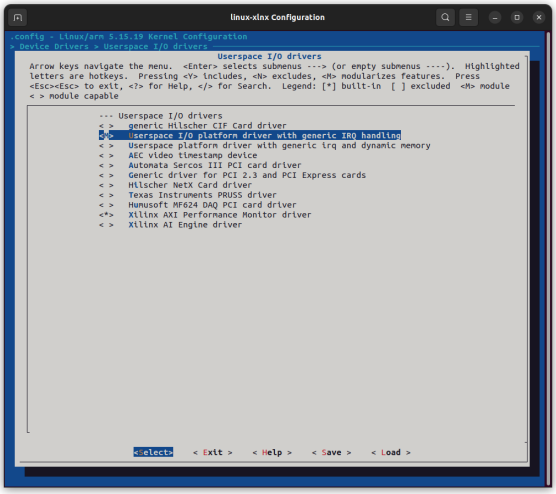


Figure 107

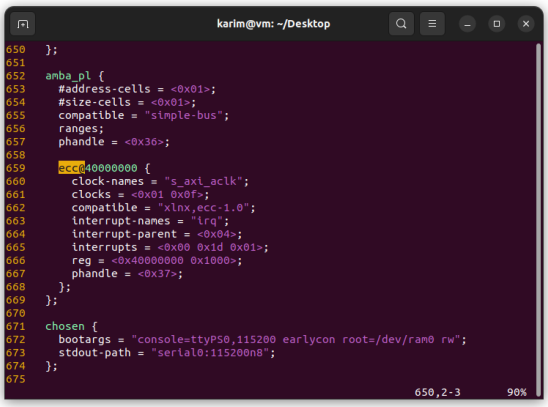


Figure 108

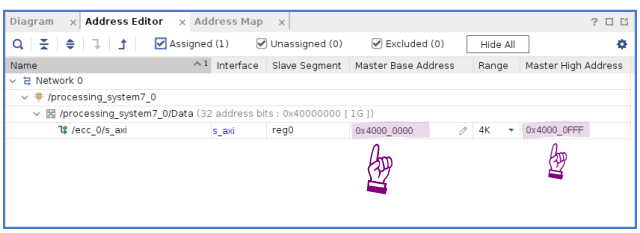


Figure 109

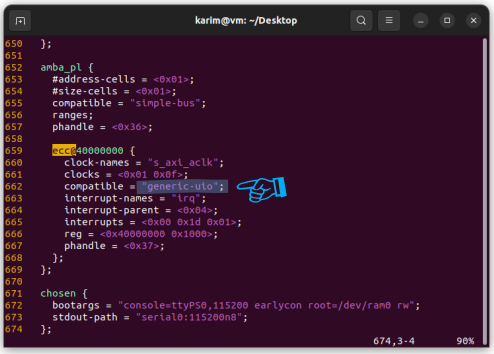


Figure 110

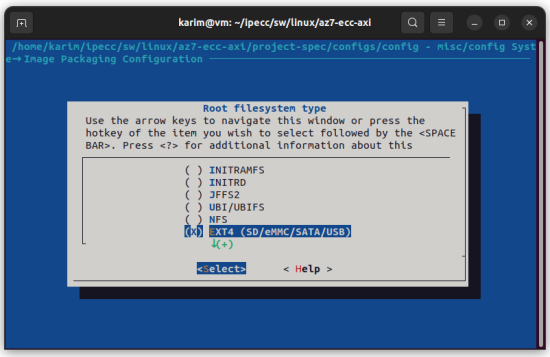


Figure 111

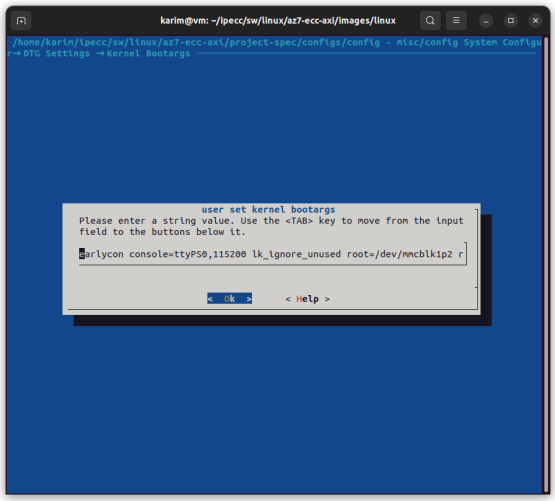


Figure 112

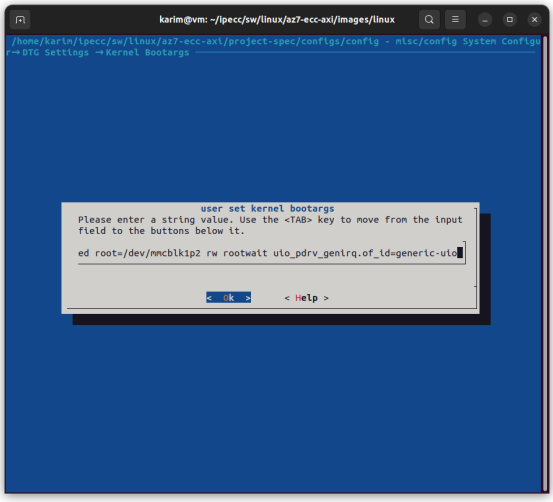


Figure 113

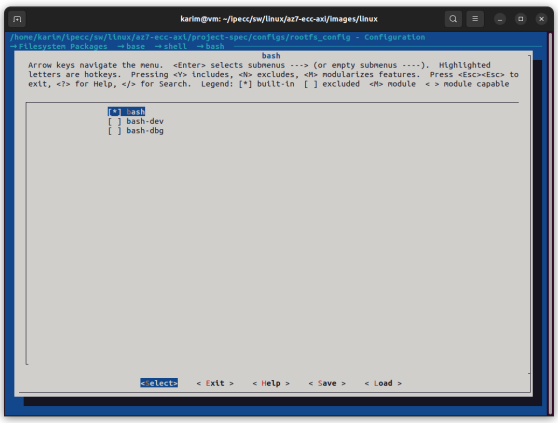


Figure 114

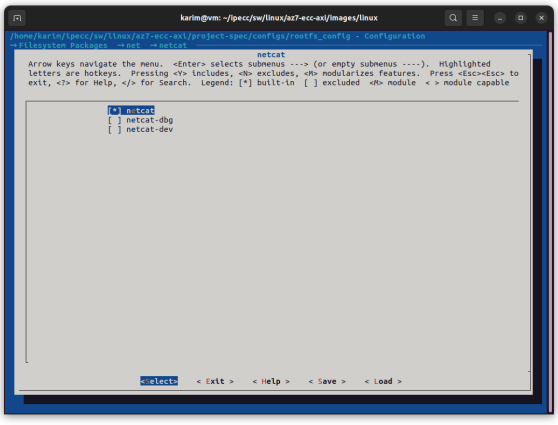


Figure 115

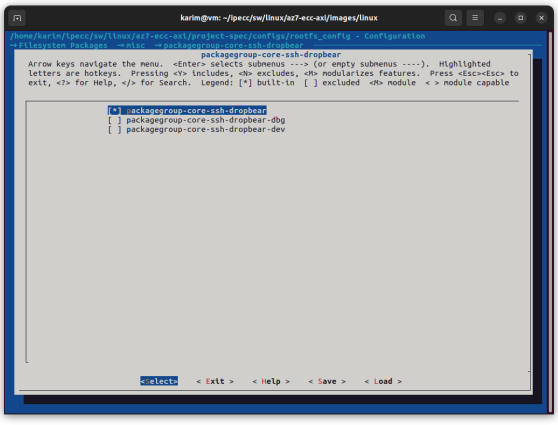


Figure 116

Figure 117

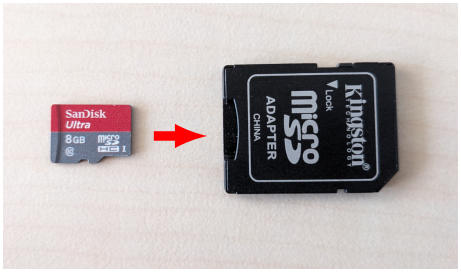


Figure 118

Figure 119

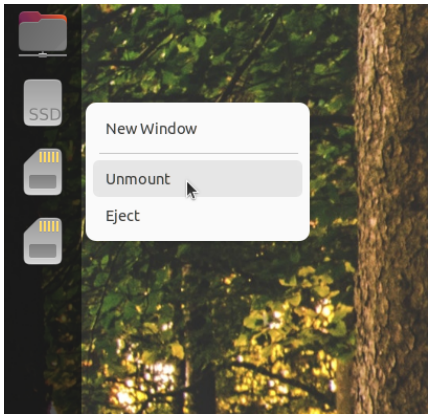


Figure 120

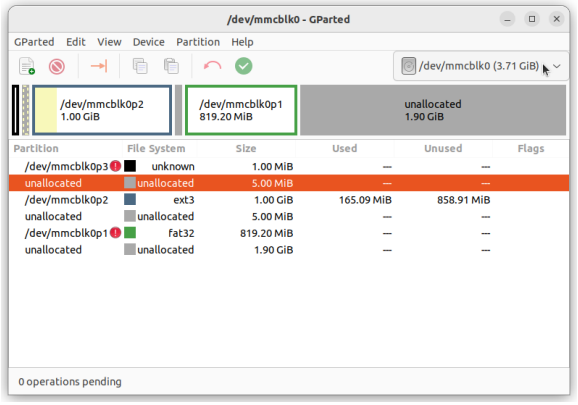


Figure 121

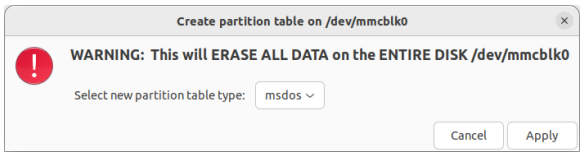


Figure 122

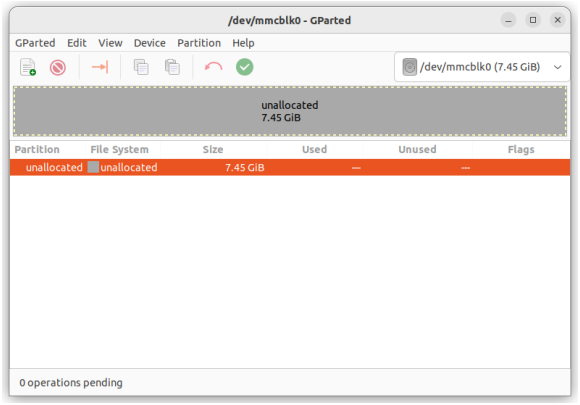


Figure 123

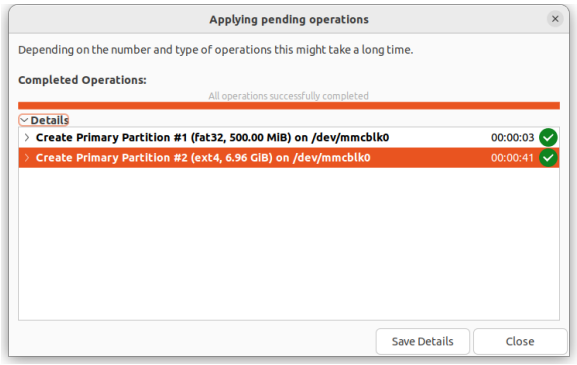


Figure 124

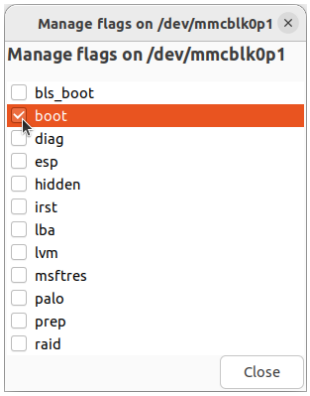


Figure 125

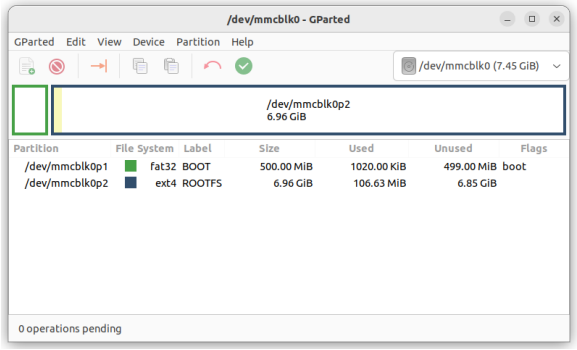


Figure 126

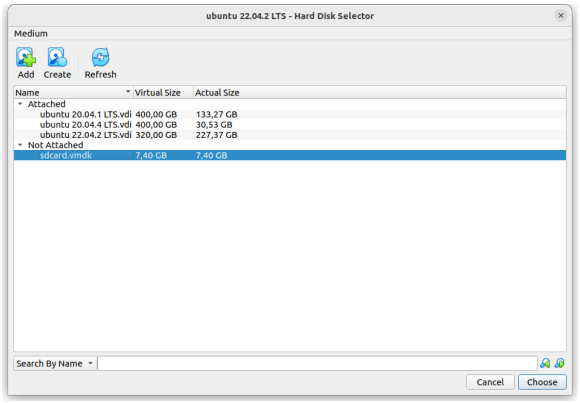


Figure 127

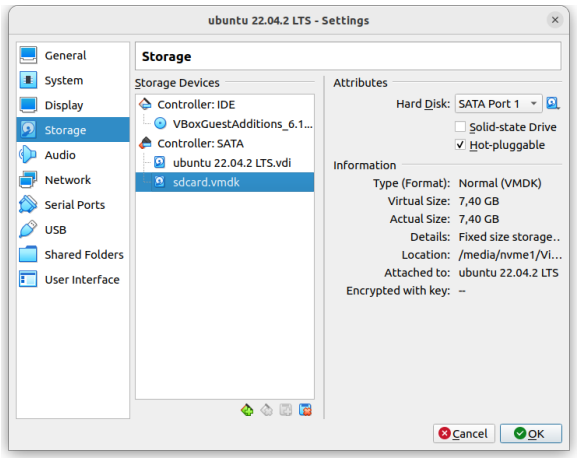


Figure 128

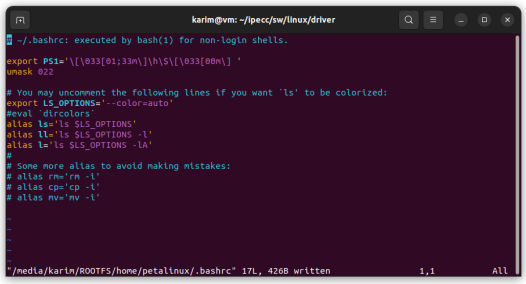


Figure 129

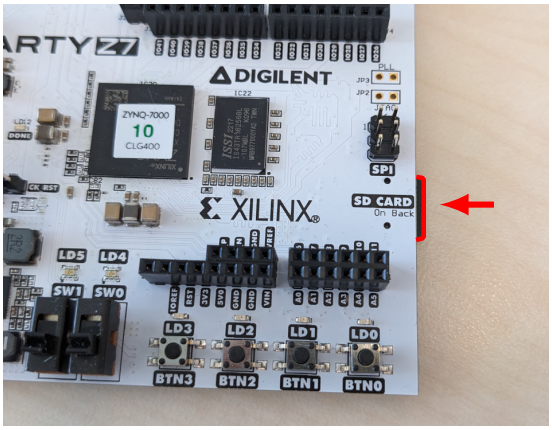


Figure 130

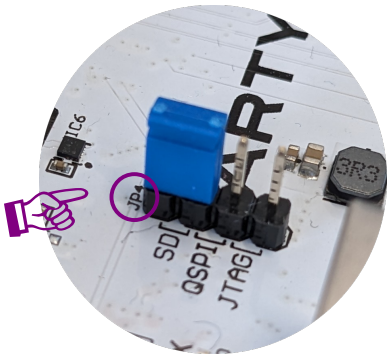


Figure 131

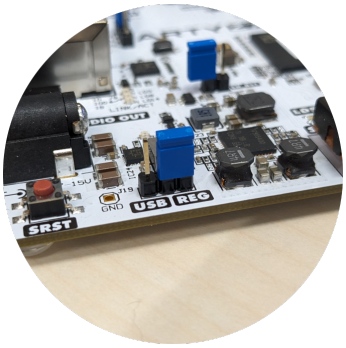


Figure 132

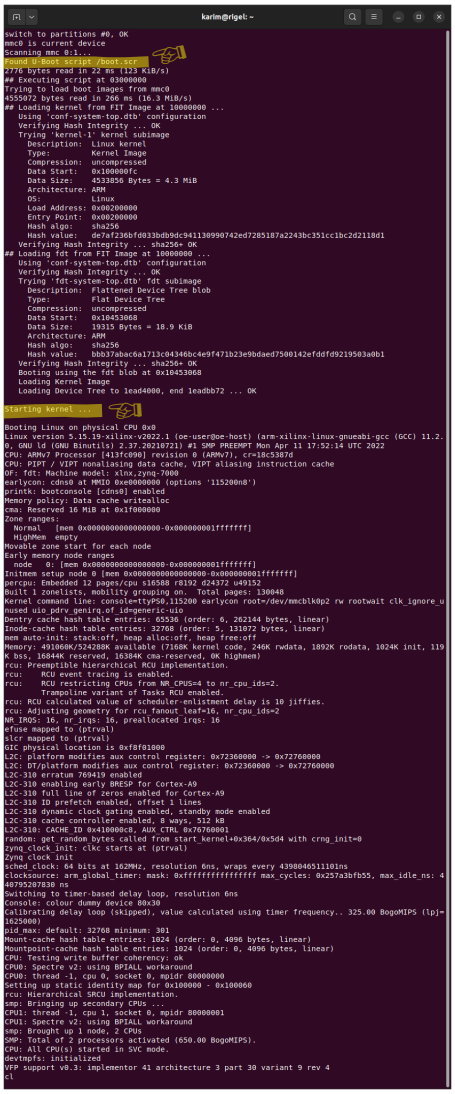


Figure 133

```
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 134

```
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 135

```
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 136