ENSEIRB-MATMECA
Bordeaux-INP

2nd year Electronics
Internship Report

Development of a SoC FPGA Board for Data Acquisition

Student :
Leo Ponsin

Lab supervisor :
Andrej Trost
(University of Ljubljana)

Academic supervisor :
Dominique Dallet,
(ENSEIRB-MATMECA)

Contents

1 Introduction

1.1 Host organization

This internship was part of my 8th-semester curriculum at ENSEIRB-MATMECA (Bordeaux INP, France) and was carried out in Slovenia from May 26 to September 21, 2025, at the Faculty of Electronics of the University of Ljubljana.

I was hosted in a laboratory focusing on FPGA and embedded systems. I was supervised in Slovenia by Andrej Trost (lab supervisor, University of Ljubljana) and in France by Prof. Dominique Dallet (academic supervisor, ENSEIRB-MATMECA, Bordeaux INP).

The technical environment included a Red Pitaya STEMlab 125-14 (Gen 1) SoC-FPGA platform, Vivado 2020.1, a Linux/SSH toolchain, and C/SCPI and Python for client applications.

Figure 1: Faculty of Electronics

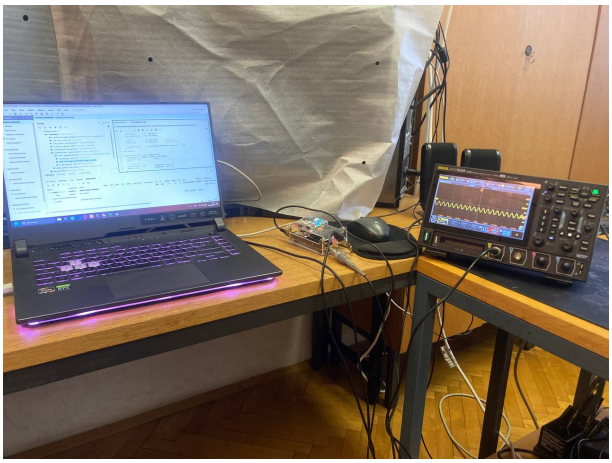

Figure 2: My setup

1.2 Context

Red Pitaya is a software-defined, open-source SoC-FPGA platform that provides instruments such as an oscilloscope, spectrum analyzer, and signal generator via a web interface.

For data acquisition, it combines real-time processing in the FPGA with C-based control on the embedded processor and SCPI commands for remote interaction.

However, the official operating system is distributed only as pre-compiled binaries and the public GitHub sources are often incomplete or insufficiently documented, so new projects rarely work out-of-the-box and require reverse engineering and manual troubleshooting. This context motivates the custom data-acquisition design developed in this report.

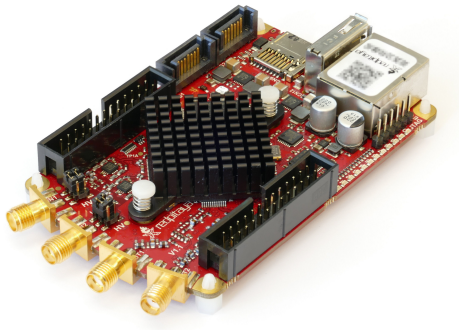

Figure 3: Red Pitaya board

Figure 4: Web interface

The core processing is done by an FPGA programmed to handle real-time signal acquisition and processing. This FPGA is controlled by software written in C, which runs on an embedded processor within the board. Communication between the user's PC and the Red Pitaya is managed through SCPI (Standard Commands for Programmable Instruments) commands, allowing standardized control of the instruments and functionalities remotely.

It has two 14-bit input channels (at ±1 V (LV) or ±20 V (HV)) and two 14-bit output channels (±1 V). Both with a sample rate of 125 MS/s.

The processor is a dual core ARM Cortex-A9, the FPGA a Xilinx Zynq 7010 SoC and it includes 4 Gbit of RAM.

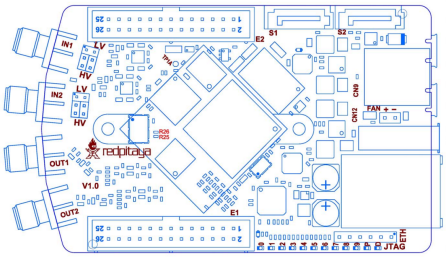

Figure 5: Red Pitaya topview schematic

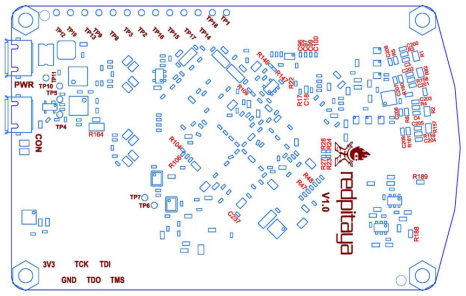

Figure 6: Red Pitaya bottomview schematic

The design couples FPGA for real-time processing, C on the PS for control, and SCPI for remote interaction, yielding a flexible, programmable chain. Access is via SSH (@192.168.1.15); documentation is partial, so practical setup required custom work.

1.3 Objectives and challenges

The main objective of this internship is to create a modular and user-friendly custom data acquisition design for the Red Pitaya (on OS2). In practice, the goal is to implement a complete acquisition chain: from ADC sampling, through synchronization and buffering in the FPGA, to data visualization and remote control via SCPI and Python.

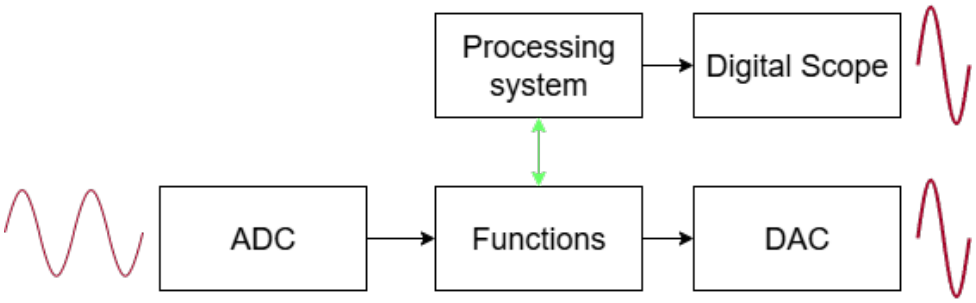

Figure 7: ADC to DAC and scope path

In this report, I describe how I built a fully custom Red Pitaya project for data acquisition, starting from scratch and without relying on the official guidelines, which did not work in my case.

It is structured as follows: first, I present preliminary experiments on OS1 to get familiar with the Red Pitaya environment. Then, I describe the development of the OS2 custom design step by step: synchronization of ADC data, FIFO buffering, simplified system bus, trigger logic, and visualization on a PC. Finally, I conclude with perspectives for future improvements.

For reference, all the FPGA sources, C/SCPI code, and Python client developed in this project are openly available at:
`https://github.com/Leooooop/Red-Pitaya-Custom-Acquisition`

2 Preliminary Work on Red Pitaya OS v1

2.1 Introduction to Red Pitaya v0.94

My lab supervisor first provided me with a Red Pitaya board running OS v1.0 (version 0.94) that already included a custom VHDL module in the design. The goal was to get initial experience with the Red Pitaya environment before starting my own project from scratch.

The hardware design was similar to the original Red Pitaya system: it included the scope connected to the web interface (useful to check modifications without an external oscilloscope), the daisy chain, and the standard ADC-to-DAC data path. The difference was that the original PID block was replaced by a custom module called proc. This module defined registers, implemented read/write logic, and was placed between the ADC and the DAC, allowing me to apply my own changes to the signal.

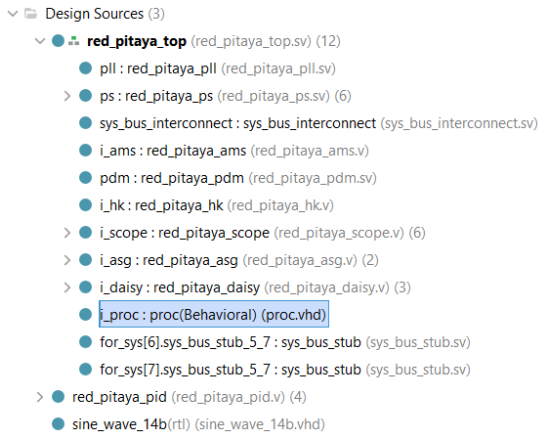

Figure 8: v94 hierarchy

```
entity proc is
  port (
    clk_i   : in  std_logic;                     -- bus clock
    rstn_i  : in  std_logic;                     -- bus reset - active low
    dat_a_i, dat_b_i  : in  std_logic_vector(13 downto 0);
    dat_a_o, dat_b_o  : out std_logic_vector(13 downto 0); -- output

    sys_addr  : in  std_logic_vector(31 downto 0);  -- bus address
    sys_wdata : in  std_logic_vector(31 downto 0);  -- bus write data
    sys_wen   : in  std_logic;                      -- bus write enable
    sys_ren   : in  std_logic;                      -- bus read enable
    sys_rdata : out std_logic_vector(31 downto 0);  -- bus read data
    sys_err   : out std_logic;                      -- bus error indicator
    sys_ack   : out std_logic                      -- bus acknowledge signal
    );
end proc;

architecture Behavioral of proc is
 signal  a, b: std_logic_vector(7 downto 0); -- amplitude registers
 signal mul_a: signed(22 downto 0);
 signal add_b: signed(13 downto 0);

begin

-- multiply signed inputs with 8-bit register, register values are unsigned
mul_a <= signed(dat_a_i) * signed('0' & a);

-- add an offset value to the input
add_b <= signed(dat_a_i) + signed('0' & std_logic_vector(resize(unsigned(b), 13)));
```

Figure 9: Declaration of custom proc module

To load custom hardware onto the Red Pitaya, a compiled bitstream must be uploaded to the board. This bitstream is generated using Vivado, with version 2020.1 being the most compatible for the Red Pitaya environment.

The Red Pitaya's FPGA design is based on a combination of a Vivado block diagram (using official IPs) and an RTL top-level module. The block diagram is mainly used to configure the processing system (PS) and other "slow" or prebuilt IPs, such as AXI buses or GPIO. In contrast, the RTL part handles high-speed logic, for example, data streaming through ADCs or the use of custom registers for signal processing.

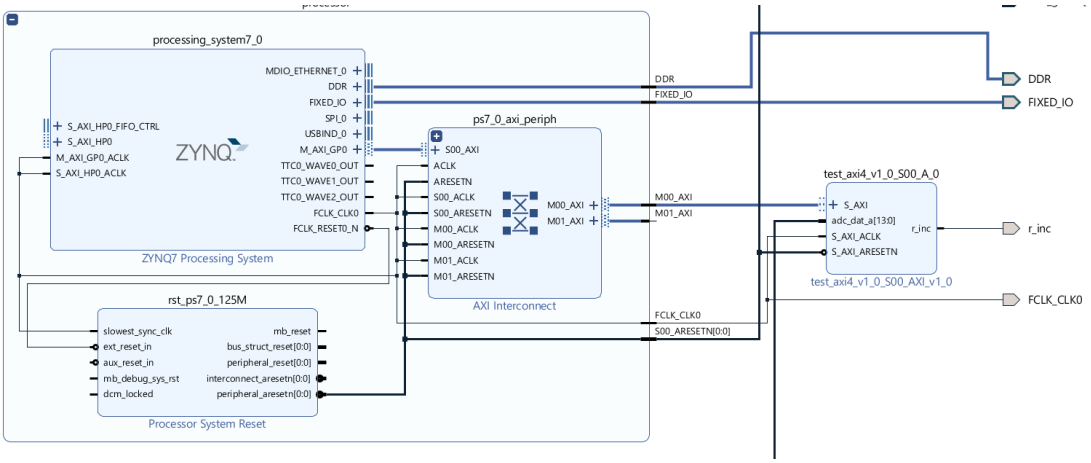

Figure 10: Vivado block diagram including the Processing System

The diagram clearly shows that data is exchanged between the PS and the FPGA logic (test_axi module) in both directions via an AXI interface. This interface allows the user to send commands from a terminal (via SSH) and write values into registers. These values are then used by the programmable logic to dynamically modify its behavior.

The `test_axi4` block in the figure defines a set of memory-mapped registers that can be accessed and modified by the user through the processing system.

The following figure summarizes the *typical* base address map used by standard Red Pitaya bitstreams.

All registers are 32-bit and aligned on 4-byte boundaries (offsets +4). Projects may remap modules.

Start	End	Module Name
0x40000000	0x400FFFFF	Housekeeping
0x40100000	0x401FFFFF	Oscilloscope
0x40200000	0x402FFFFF	Arbitrary Signal Generator (ASG)
0x40300000	0x403FFFFF	PID Controller
0x40400000	0x404FFFFF	Analog Mixed Signals (AMS)
0x40500000	0x405FFFFF	Daisy Chain
0x40600000	0x406FFFFF	FREE
0x40700000	0x407FFFFF	Power Test

Table 1: Typical default base addresses on Red Pitaya bitstreams (32-bit, 4-byte aligned).

Usage note. In this report, I primarily use the Oscilloscope base (`0x4010_0000`) for the scope/trigger path; module IDs are placed at offset 0 whenever applicable.

This means that to modify a parameter of a given module (for example, the oscilloscope), we must write into a register located within its address range, e.g. between `0x40100000` and `0x401FFFFF`. If we decide that `0x40100004` corresponds to the scale parameter, then changing the value of this register directly changes the oscilloscope scale.

This hybrid FPGA design approach enables both high-level system configuration and low-level hardware acceleration within the same project.

2.2 Hardware Experiment on OS 1

To experiment with the hardware, I reused the existing register declarations in the proc module. I then implemented a simple processing operation: multiplying the ADC data by a factor and adding an offset, both controlled through registers accessible from the Processing System.

The snippet below shows how the multiplication factor is handled. A register a is defined and updated whenever the Processing System writes to the address 0x40300054. The written value is limited to 0x0F to avoid overflow. This register is then used in the signal processing path to scale the ADC data.

Snippet showing the multiplication

```
signal mul_a: signed(22 downto 0);
mul_a <= signed(dat_a_i) * signed('0' & a);
dat_a_o <= std_logic_vector(mul_a(17 downto 4));
```

This piece of code multiplies the input data dat_a_i by the register value a. Because the product is wider than 14 bits, only bits [17:4] are kept, which corresponds to a right shift of 4 (division by 16). This ensures that the output remains compatible with the 14-bit width of the Red Pitaya ADC/DAC data path.

The register logic is also illustrated below. When the system bus detects a write at offset 0x54, the value is stored in a. Reads at the same address return the current register value, while address 0x50 is reserved for an identification code.

Register write/read logic for the gain

```
if sys_wen='1' then
    if sys_addr(19 downto 0)=X"00054" then
        if (sys_wdata(7 downto 0) < x"0f") then
            a <= sys_wdata(7 downto 0);
        else
            a <= X"0f";
-------------------------------------------------------------------------------
with sys_addr(19 downto 0) select
    sys_rdata <= X"FE220001" when x"00050", -- ID
                 X"000000" & a when x"00054",
                 X"00000000" when others;
```

In practice, writing a value between 0 and 0x0F to the register at address 0x40300054 changes the multiplication factor applied to the ADC input. The effect of this operation can be directly observed on the Red Pitaya web oscilloscope interface. The value can be changed by typing monitor "addr" "val" into the Red Pitaya Linux terminal and no value if one only wants to read the content of the register.

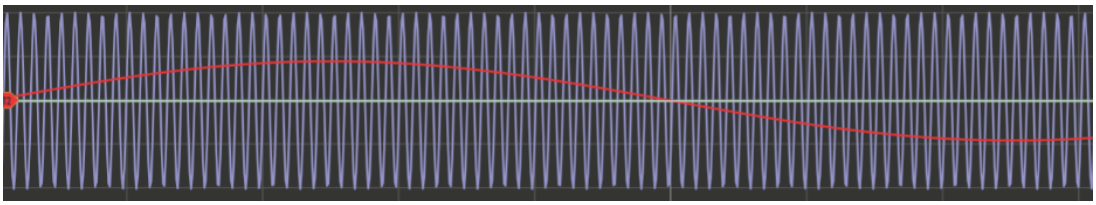

Figure 11: Out signal (red) and ADC data (purple) on oscilloscope

We can see that the output signal is smaller than the ADC one. The time scale was changed to clearly see both signals.

Now that we have a simple logic we can try to control it by the software.

2.3 Software control on OS 1

To control the FPGA logic, I first created a small C driver (`rp.c` and `rp.h`) defining the main functions used to connect to the board, read and write registers, and cleanly release resources. This avoids repeating for example low-level `mmap` code in every program and keeps the applications more readable. Code below shows the initialization function of the driver and the function I can write with.

```c
int rp_init(off_t base_addr) {
    fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) {
        perror("open /dev/mem");
        return -1;
    }

    adr = mmap(NULL, PAGE_SIZE, PROT_READ
        | PROT_WRITE, MAP_SHARED, fd, base_addr);
    if (adr == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return -1;
    }

    return 0;
```

Figure 12: Driver's init function

```c
int multiply(int value)
{
  off_t BASE = 0x40300000; // PID base address;

  rp_init(BASE);

  printf("Initial val was: %d\n", rp_read(0x54));

  /* write to "a" register */
  rp_write(0x54,value);

  printf("It is now set to: %d\n", rp_read(0x54));
  printf("input value: %d (0x%X)\n", value, value);

  rp_cleanup();
  return 0;
```

Figure 13: multiply.c function

Once the driver was in place, I could implement the actual control programs. As an example, since I previously presented the multiplication logic in VHDL, the code shows how the multiply register is controlled from software using the driver's functions.

This program simply calls the driver's functions to write into the multiply register. It can be compiled and executed directly, providing a clean alternative to manual register access from the terminal.

2.4 Basic SCPI server

SCPI (Standard Commands for Programmable Instruments) is a text-based protocol used to control instruments over a network. Commands are written as strings (for example OUTPUT1:STATE? or GEN:RST). Usually commands with ? answer to the client and the other write in registers/change a status. On the Red Pitaya, the SCPI server runs on the Processing System and forwards the commands to the FPGA registers, making it possible to configure and read hardware parameters remotely.

The official SCPI server of Red Pitaya is provided as a precompiled program. This means it cannot be customized easily to add new commands for my own registers. In addition, the official server is very large and complex, including many features that I did not need. When I tried to use it together with my custom logic, it also created conflicts with the existing applications already running on the board. For this reason, I decided to build a minimal SCPI server with only the commands necessary to control my registers.

File	Description
`scpi_commands.c`	Contains the implementation of custom SCPI commands. Each function corresponds to a command (e.g. `TRIG:LEV 0x2200`).
`scpi_commands.h`	Header file declaring the prototypes of the SCPI commands. It makes the functions available to the main SCPI server.
`scpi_main.c`	The main file of the SCPI server. It initializes the command parser, loads the custom command set, and listens for incoming SCPI requests on the Red Pitaya.
`client.py`	A simple Python client running on my computer. It connects to the Red Pitaya by IP address and port, then sends SCPI strings to control or read back the registers.

Table 2: Structure of the minimal SCPI server

To avoid conflicts the port I use is 5010 (original port 5000).
This structure is what I will also use during my OS 2 development.

3 Development of the custom OS 2 version

After the update to OS 2 version (downloaded on the official website) I have to download the FPGA repository from GitHub available at `https://github.com/RedPitaya/RedPitaya-FPGA`.

3.1 PS <-> PL

As I couldn't get the whole project following the guidelines (using commands with build project etc) I decided to start from an example provided by Anton Potočnik ("Stopwatch" from `https://antonpotocnik.com`) which included basic communication between Linux processing system and the programmable logic.

I rapidly connected my own axi IP "axi_test" to the diagram. It is a Vivado generated axi-4 lite IP connected to an Axi Interconnect and the PS.

I can then add my own registers, inputs and outputs and to test it I create a 14 bits sine wave with an array and connect it to the input of the IP.

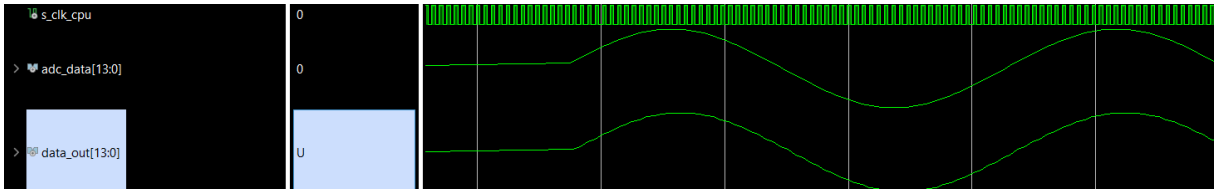

Figure 14: Simulation of axi4-lite behavior

When the sine wave is connected to the input, the output is the same, so the path is working. The next step is to connect the actual ADC to the IP.

3.2 ADC and clock synchronization

To connect the ADC to the Processing System (PS), it is important to consider that the ADC and the PS run on different clock domains. Therefore, the data must be synchronized before being used by the PS.

Here is a schematic representation of the required synchronization principle that I will detail:

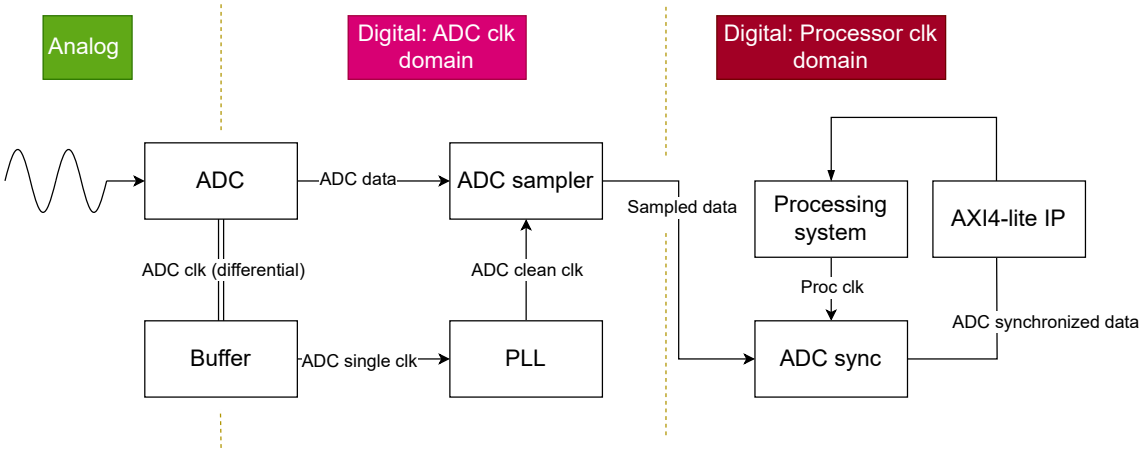

Figure 15: ADC data synchronization principle with the PS

CLK generation ⇒ The ADC is driven by a differential 125 MHz clock coming from outside the board.

In our design, this clock is first passed through a buffer (Vivado standard IP) to generate a single-ended version.

It is then fed into a PLL, which provides a cleaner and more stable clock before being used by the design and sent to the double registers.

Data path ⇒ To transfer the ADC data into the AXI4-Lite IP, the signals must first be synchronized across clock domains.

This is achieved with a two-register synchronizer: the first register is clocked by the ADC clock from the PLL, and the second one by the PS clock.

The output of the second register is then safely connected to the AXI module, allowing the ADC samples to be read one by one without metastability issues.

I can then write a C code which reads a 1000 times the register and fill a CSV file inside the Red Pitaya:

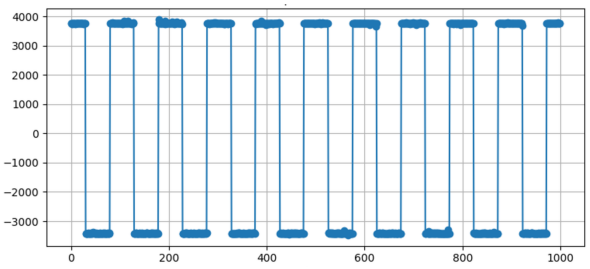

Figure 16: Square wave acquired by the PS

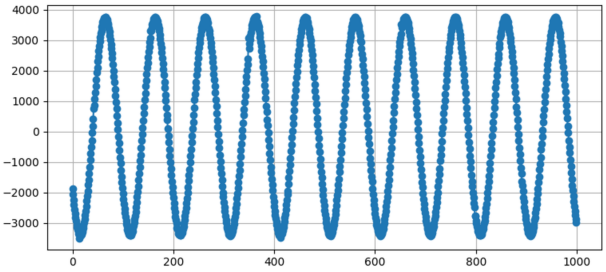

Figure 17: Sine wave acquired by the PS

The data are shown by a python code outside the Red Pitaya with the CSV file that I copy from the board.

One main issue with that design is that the PS is too slow to follow the pace of the ADC. One data sample is available at a time and if the PS is not reading as fast as the sample rate (125 MHz, impossible) some data are lost and also it is impossible to visualize high frequency signal.

That's why I decided to use a FIFO to buffer the data like the official design.

3.3 Use of FIFO

The double clock FIFO I used comes from Clifford Cummings well-known asynchronous FIFO design (SNUG 2002). It has the usual signals like full and empty, and its depth can be configured. I use it to buffer ADC data before sending it to the PS.

I did not design the FIFO myself because asynchronous FIFOs are difficult to implement correctly. A naive design may work in simulation but fail in real hardware. Using a proven implementation avoids these risks and ensures reliable data transfer between the ADC and the PS.

This design uses gray code, which allows comparisons without multi bit errors. Gray code is a numbering system where only one bit changes at a time, which avoids synchronization errors when transferring data between different clock domains. More explanations can be found here:

`http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf`

If I set r_inc (the read pointer) to 1 when I read the data, the result is almost the same as when using only one register with a sine wave input. This happens because the FIFO is empty at t=0, it starts filling with data at 125 MHz, and the read side cannot keep up at the same speed. As a result, once the FIFO is full, its behavior is essentially the same as a single register, except for the very first samples which are correctly aligned.

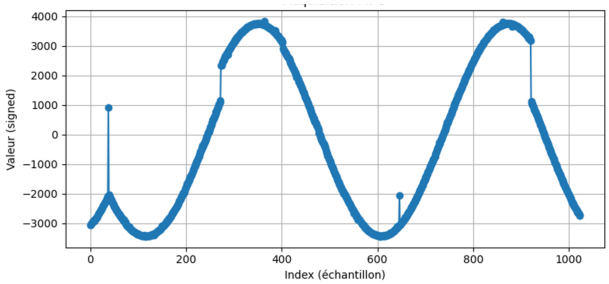

Figure 18: First FIFO implement capture

The FIFO output follows the sine wave input, but some discontinuities appear.

These glitches occur because the `empty` and `full` flags are not checked so the read pointer continues to increment even when no valid data is available, and the write pointer may advance when the FIFO is already full.

As a result, the pointers lose synchronization and the output signal shows abrupt jumps.

But before checking the flags I have to simplify the Bus between the PS and the PL.

3.4 Creation of an easier bus (sys bus)

AXI4-Lite is heavy to modify and time-consuming to connect in RTL. The simpler sys_bus, already used in OS1, needs only about 10 signals instead of 30, making it much easier for custom registers.

Channel / Path	**AXI4-Lite signals**	**sys_bus signals**
Clock / Reset	`ACLK`, `ARESETN`	(shared)
Write Addr	`awaddr[AW-1:0]`, `awprot[2:0]`, `awvalid`, `awready`	`waddr[AW-1:0]`
Write Data	`wdata[DW-1:0]`, `wstrb[(DW/8)-1:0]`, `wvalid`, `wready`	`wdata[DW-1:0]`, `wsel[(DW/8)-1:0]`, `wvalid`
Write Resp	`bresp[1:0]`, `bvalid`, `bready`	`wrdy`, `werr`, `wlen[3:0]`, `wfixed`
Read Addr	`araddr[AW-1:0]`, `arprot[2:0]`, `arvalid`, `arready`	`raddr[AW-1:0]`, `ren`
Read Data	`rdata[DW-1:0]`, `rresp[1:0]`, `rvalid`, `rready`	`rdata[DW-1:0]`, `rrdy`
Total signals	∼30	∼10–12

Table 3: Comparison between *AXI4-Lite* and *sys_bus* interfaces (prefixes omitted).

In the old bus, each module had to be wrapped as an AXI4-Lite IP. The logic was split between the AXI wrapper and RTL, and adding a new module required changes in the block design and AXI Interconnect.

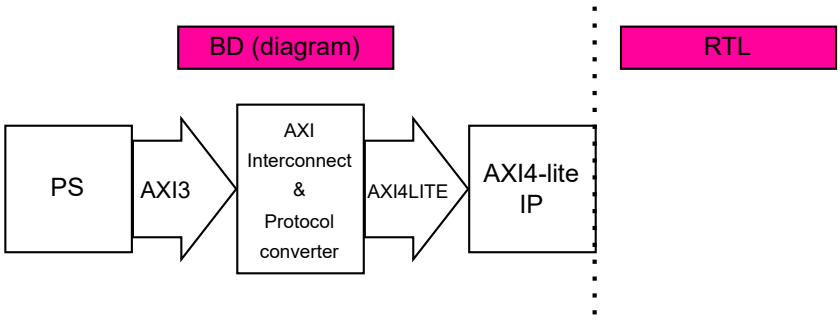

Figure 19: Old Bus

In the new bus, the AXI3 from the PS is converted once into a lightweight sys_bus. All the logic is handled in RTL, and new modules can simply be attached to the sys_bus, making the design much easier to extend.

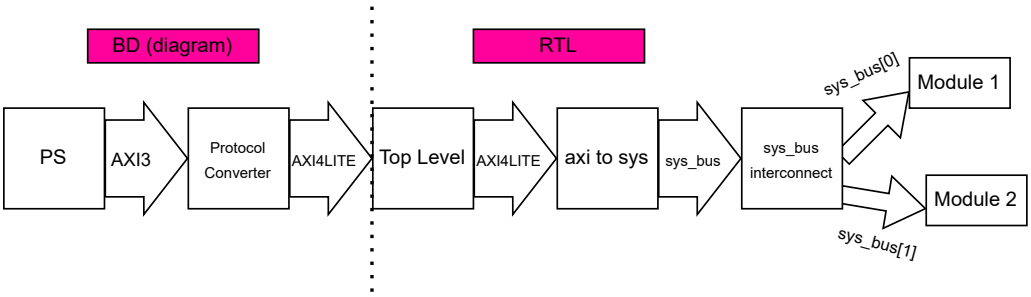

Figure 20: New Bus

I can then create more easily a trigger and a module to control data acquisition from the ADC with my FIFO.

3.5 Trigger and acquisition

To behave like a real oscilloscope, the data must first be acquired in the FIFO. Then, the PS reads all the samples. Once the FIFO is empty again, a new acquisition can start. That's why a trigger which decides when to acquire and when to stop is needed.

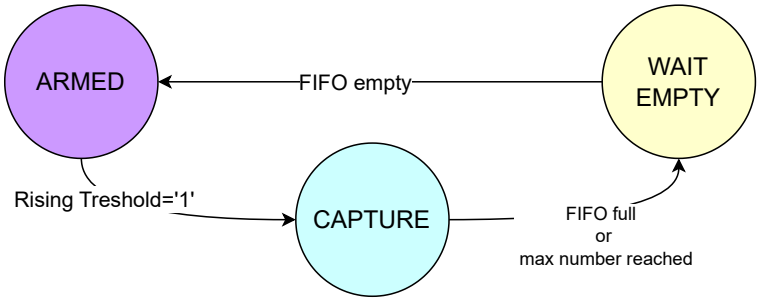

Figure 21: Diagram showing how the FSM is working

Now that the trigger is designed, I can try to take the data to an external PC and show how the signal looks like.

3.6 ADC visualization on a PC

To visualize the signal on a PC, three elements are needed: a scope module in the hardware design, a C program running on the board triggered via SCPI, and a client from the PC that gets the data and plots it on a graph.

All the required elements have already been described.

3.6.1 Scope

The scope module uses sys_bus to send the data to the PS, the FIFO and the FSM trigger. Here are the offsets of the different registers (BASE: 0x40100000);

```
constant C_ID              : std_logic_vector(31 downto 0) := x"00123456";
constant REG_ID            : unsigned(19 downto 0) := x"00000";
constant REG_STATUS        : unsigned(19 downto 0) := x"00004";
constant REG_DATA          : unsigned(19 downto 0) := x"00008";
constant REG_TRIG_LEVEL    : unsigned(19 downto 0) := x"0000C";
```

Figure 22: Scope registers offsets

This scope allows the user to read the first element of the FIFO, to set a trigger value, and to check the capture status (waiting, capturing, etc.). It also sends the flag `r_empty` to the trigger FSM. Flags such as `r_empty` or `capturing_adc` are transferred from one clock domain to another using a double flip-flop synchronizer (the same principle as the double-register technique discussed earlier).

3.6.2 SCPI and C code

Now that the data can be read from the FIFO, a program can gather it and add it to the SCPI server commands.

SCOPECH1? code snippet

```
if (strcmp(cmd, "scopech1?") == 0) {
    // ... INIT (hardware setup done once)

    // Stream "v0,v1,...,v9999\r\n"
    while (count < N_SAMPLES) {
        uint32_t st = (uint32_t)rp_read(REG_STATUS);

        if (st & ST_VALID) {
            uint32_t raw = rp_read(REG_DATA) & 0x3FFFu;
            int16_t  val = -tc14_to_s16(raw);
            snprintf(out, sizeof(out), "%s%d", count==0?"":",", val);
            write(fd, out, strlen(out));
            count++;
        } else if (st & ST_EMPTY) {
            usleep(50);
        }
    }
    write(fd, "\r\n", 2);
} // ...
```

The snippet shows how the SCPI server handles the command `scopech1?`. After initialization, the program repeatedly checks `REG_STATUS`, reads samples from `REG_DATA`, and streams them as ASCII CSV text to the client.

3.6.3 Python client

To use the client we need to specify the connection parameters (`PORT` and `IP`), configure the socket, and gather the data.

The client collects the data frame by frame (10 000 samples each), in the same way as the hardware scope. This allows visualization of full periods from 10 MHz down to 10 kHz. At 10 MHz, the 125 MHz sampling rate of the ADC is already close to the practical oversampling limit.

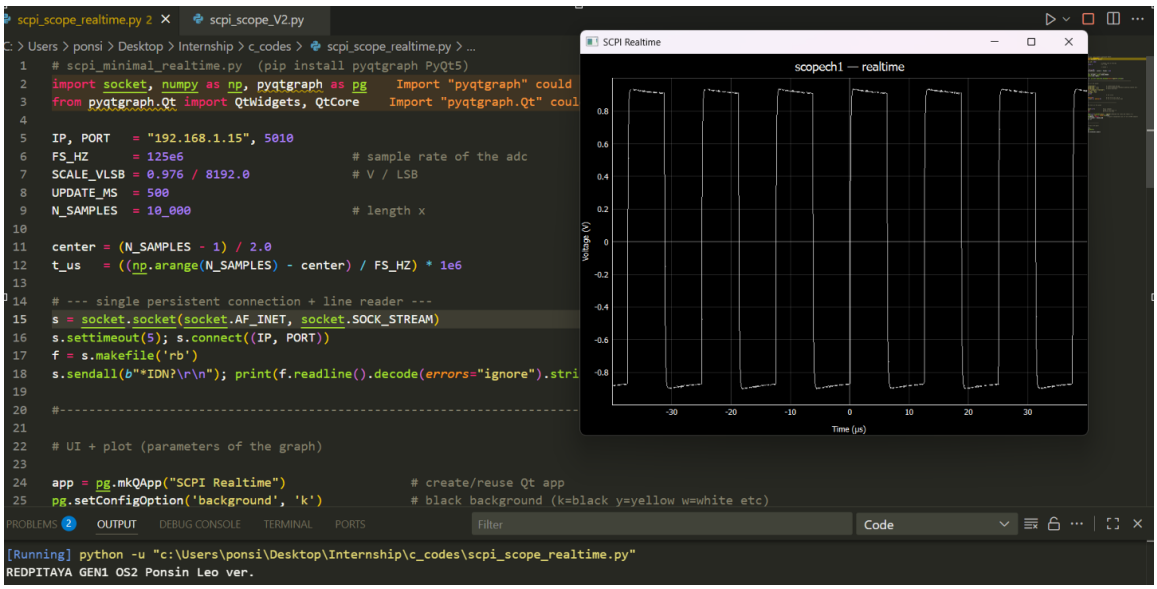

Figure 23: Screenshot from the client environment

The code is running on the PC and updating every 0.5 seconds the data array. Then it uses a Qt-based application for graphical visualization. The values of volt are converted here (+8192 LSB = 0.976 V).

I also developed a separate SCPI code to control the trigger. However, it cannot run at the same time as the scope server, since both programs access the same hardware resources and network port.

3.7 DAC visualization

To achieve near real-time visualization of the input signal, the design remains fully in the RTL domain. A first FIFO transfers data into a register accessible from the PS (processor clock domain), and a second FIFO converts it back to the DAC domain (same as the ADC).

This register can enable offset and gain adjustments, similar to the custom module implemented in the OS1 version at the beginning of the project.

Figure 24: DAC visualization

This is the last feature included in my design but there are a lot of things that can optimize it and upgrade the user experience.

4 Conclusion

4.1 Summary

Starting from initial experiments on the OS 1 version, I progressively developed from scratch a custom OS 2 design for the Red Pitaya. The result is a basic but fully functional data acquisition chain, easier to understand and free of unnecessary complexity or "black-box" components. This work can serve as a foundation for future projects and educational purposes. Future students can easily add their own processing modules (filters, decimation, new triggers, etc.) by following the same structure.

4.2 Improvements and Perspectives

Several extensions could be implemented to improve the current system:

- **Decimation and averaging** to adapt the acquisition to different signal bandwidths and improve noise performance.
- **Cleaner client application** with better visualization and user interface, especially to change the trigger while the scope is running.
- **Automatic acquisition fallback** if the trigger configuration is wrong or missing, to avoid blocking the measurement.

I hope that this work will be reused as a learning tool or as a starting point by other students, providing an alternative perspective to the official documentation and forum contributions.

I would also like to thank the University of Ljubljana for hosting me during this internship, and I am especially grateful to my supervisor for the guidance and support throughout the project.

Appendices

Addendum **Economics and Management**
The chosen theme for this addendum is: Project organization

A Project Organization

This annex summarizes how I planned and executed the work as a solo project. It complements the technical chapters by focusing on process, not on RTL details.

A.1 Context and constraints

The internship ran from May 26 to September 21, 2025, at the Faculty of Electronics in Ljubljana, Slovenia. I worked alone, on a mobile bench (no fixed desk), with a Red Pitaya STEMlab 125-14, a signal generator, an oscilloscope, and my computer. The toolchain was Vivado 2020.1 and a Linux/SSH workflow. Documentation from public repositories was sometimes incomplete, so part of the effort involved reading sources, trying minimal examples, and doing small amounts of reverse engineering when needed. I also spent time experimenting to get more familiar with how everything worked together.

A.2 Working principles

Short, visible steps. I picked one small goal at a time: a module close to a visible output or easy to simulate. I often started with a hard-coded behavior to "see something" immediately (for example, using a 14-bit sine wave module on Vivado), then exposed registers and integrated the block into the full chain (ADC → FIFO → simple bus → trigger/scope → SCPI/Python).

Clear hierarchy. I kept wrappers small and a clean top-level, so infrastructure (clocks, resets, buses) stayed readable and independent from the high-speed data path. For example, the ADC-to-DAC wrapper includes two FIFOs and the module running on the PS clock. This is the opposite of the older, monolithic OS1 style that had large files mixing different logic, like in version 0.94 where the board diagram also contained the AXI4-to-sys_bus conversion and the sys_bus interconnect all in one place. My approach made navigation, reuse, and debugging easier for me, but I think also for future users.

Module inventory (figure). I tried to summarize, in a non-RTL diagram, all the small modules inside the wrappers. It is basically an inventory of the hardware. The FIFOs are the same except for their size, which is why they are zoomed out in the diagram.

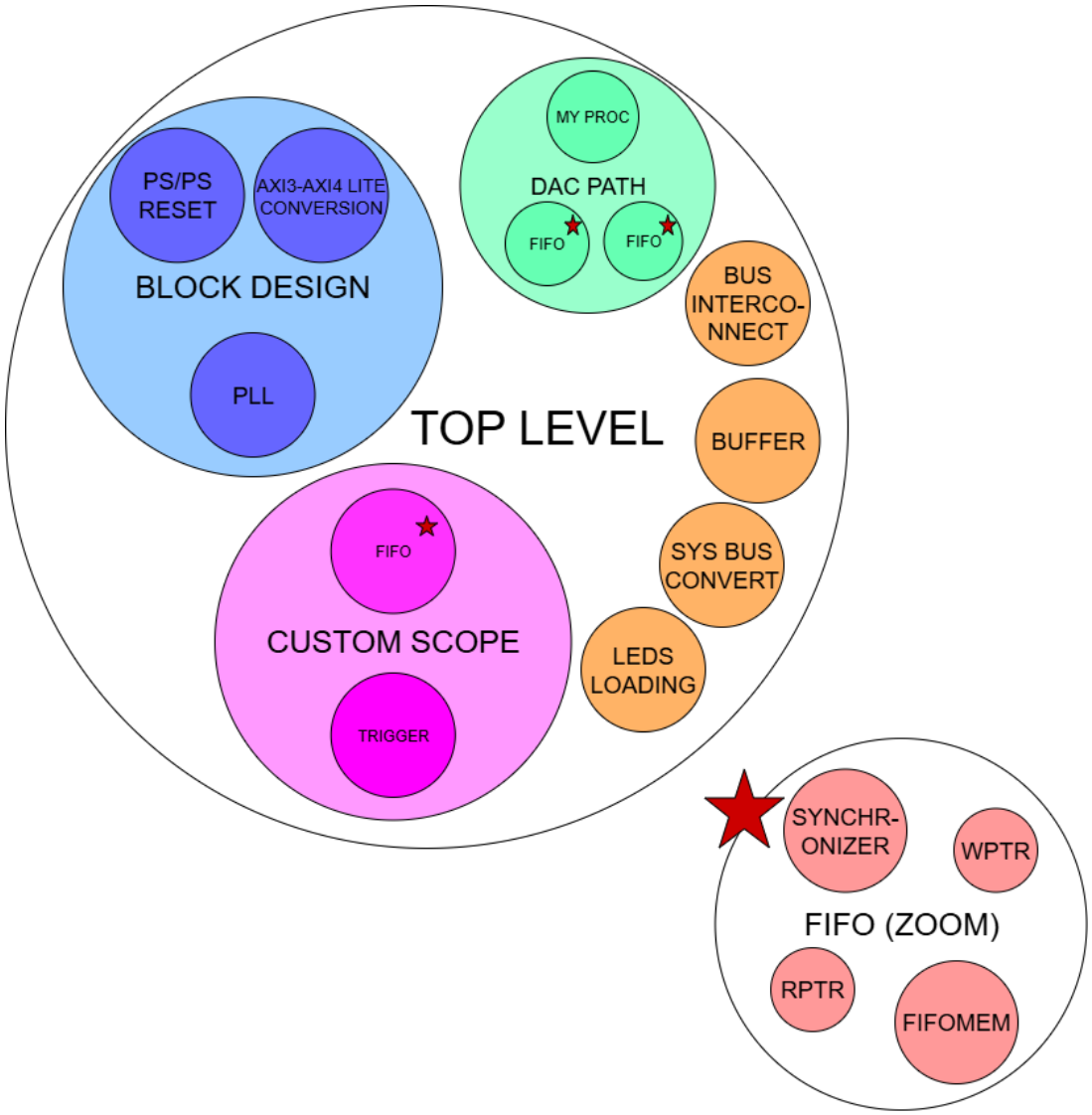

Figure 25: Inventory of my hardware modules

A.3 Tooling and conventions

Deploy fast. I used a single script, `upload.bit`, launched from my main project folder. It finds the latest Vivado artifact under `.runs`, converts `.bit` → `.bif` → `.bin` with the official Red Pitaya flow, and copies the file to `/opt/redpitaya/fpga`. On the board, I worked under `/opt/redpitaya`. This removed manual convert/copy steps; I did not keep logs for uploads. It follows the Red Pitaya official bitstream rules for OS2.

Quick checks. Each custom block exposes a unique register **ID** at offset 0, to check that we are writing to the right addresses. There is also a **leds_loading** indicator which visually confirms that the new bitstream is active and that the board did not reboot while writing to it. Reboots can happen when addresses are not written or exposed correctly. I also added the ID to the minimal SCPI server with `*IDN?`, which replies *"REDPITAYA GEN1 OS2 Ponsin Leo ver."*, enough to verify end-to-end connectivity in seconds.

Register map rules. All registers are 32-bit and aligned on 4-byte boundaries (offsets

0x00, 0x04, ...). The module ID is always at offset 0. These rules keep drivers and manual `monitor` reads simple.

Ports and paths. The SCPI server uses a dedicated port (5010) to avoid conflicts with the official compiled one. Bitstreams are deployed to `/opt/redpitaya/fpga`.

A.4 Testing, acceptance, and CDC

Simulation vs board. I used simulation for early blocks (OS1 and start of OS2) and with the internal sine-wave path. When two clock domains were involved, simulation became less representative: even if we tested with different frequencies and offsets, the clocks in simulation were still too synchronized. So I prioritized tests on the board to see the real effects.

CDC rule. Signals crossing between `adc_clk` and `ps_clk` are handled with two-flip-flop synchronizers and a proven asynchronous FIFO (for data). FIFOs are used for multi-bit signals, and when it is only flags, two flip-flops are enough. I also enabled IOB registers on critical I/O to tighten device-boundary timing.

Acceptance. For every test I did, the first check was the ID at offset 0 of every register and the visualization of the `leds_loading`. Then I checked if "monitor" on the board terminal could write and read other registers. After that, I checked if the values inside the CSV files (at the beginning of development) and on the scope (later) were similar to the input signal. Sometimes it was inverted, sometimes the FIFO was not empty. So I had to repeat experiments more than once after each change.

A.5 Prioritization and task sourcing

When a big change was made, I prioritized making it robust and user friendly before moving to another task. Otherwise, if a problem appeared later, I would have to check both the new module and the old ones again. Since I had many modules, this could become really annoying.

A.6 Risks and mitigations (top-3)

CDC/timing at 125 MHz. Mitigated by async FIFO + 2-FF synchronizers; clocks may be declared asynchronous if required.

PS throughput and data loss. Mitigated by buffering (frames of 10k samples) and status-driven reads.

Mobile bench and setup drift. Mitigated by a wiring checklist, short cables, and reference captures.

A.7 Deliverables and traceability

Artifacts. FPGA sources (RTL/wrappers), Vivado project, SCPI server code with minimal commands, C driver, Python client, register map, and notes. Everything is on GitHub at `https://github.com/Leooooop/Red-Pitaya-Custom-Acquisition`.

Versioning. Incremental tags (e.g., FIFO OK, *sys_bus* integrated, trigger set to <val>).

Hand-over. A set of README files explains how the project is organized and how to make it work, including common traps to avoid and step-by-step instructions. There is also this report, which describes more of the technical work behind it.