Redpitaya: first Vivado project example, using the RF ADC and DAC

G. Goavec-Mérou

October 15, 2024

This documents aims at providing basics on:

- creating a basic Vivado project and the associated block design,
- adding IP and connections between these processing blocks as well as towards the FPGA pins,
- generating the bitstream,
- converting the bitstream to a format usable with GNU/Linux and configuring the FPGA.

This presentation will aim at connecting the Redpitaya radiofrequency ADC output to the DAC input (Fig. 1).

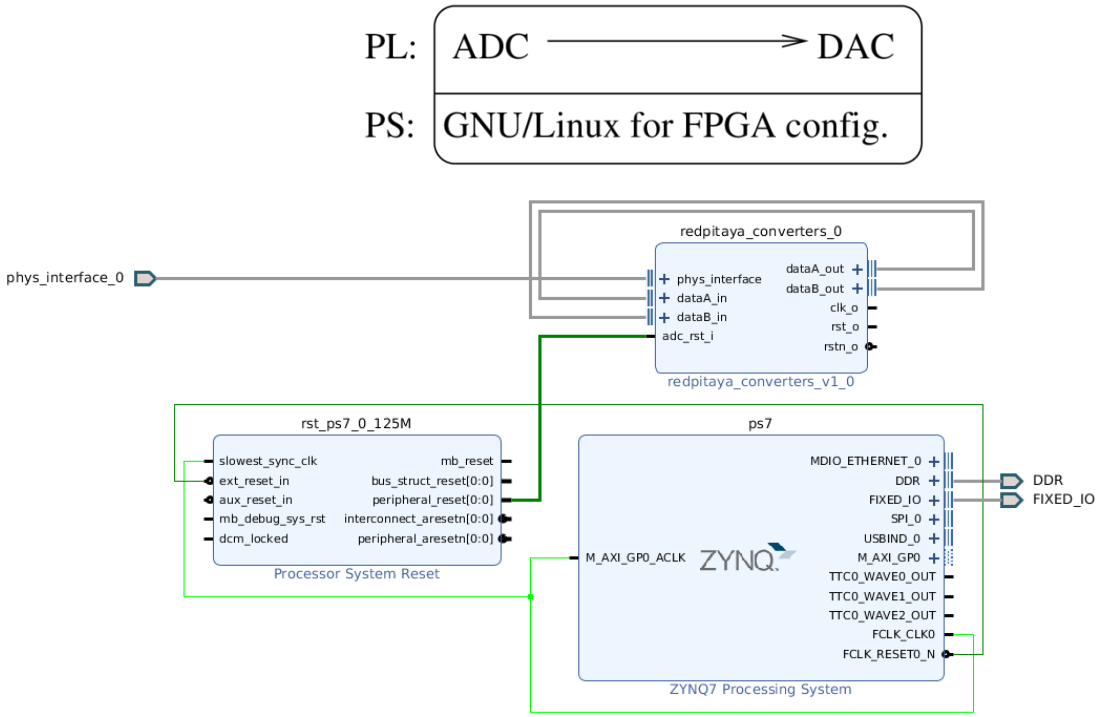

Figure 1: Objective of the tutorial (top) and block design (bottom) including the processor, and the combined ADC/DAC block including clocking circuit.

1 Creating the design

Creating a new design for the Redpitaya requires configuring a project for the Zynq 7010 embedded on the board (Figs. 2, 3, 4 and 5): despite not being defined in Xilinx Vivado, we provide manually the proper Zynq declination instead of the platform settings (Fig. 4).

Such a result is achieved by selecting a `RTL Project` so that all additional configurations are performed manually. The option *Do not specify sources at this time* prevents `Vivado` from asking the list of source files at the creation of the project (Fig. 3).

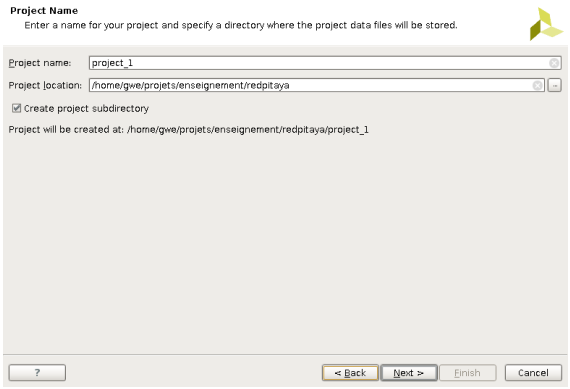

Figure 2: Selecting the project name and storage location

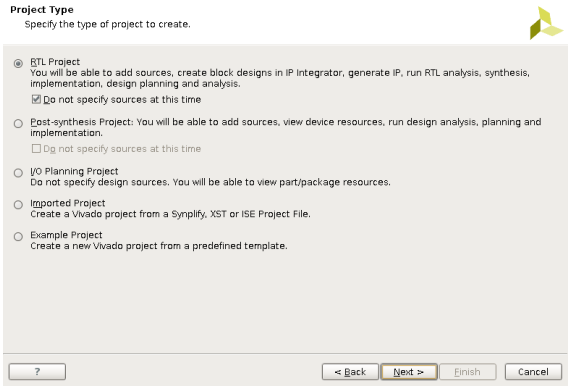

Figure 3: Selecting the project type.

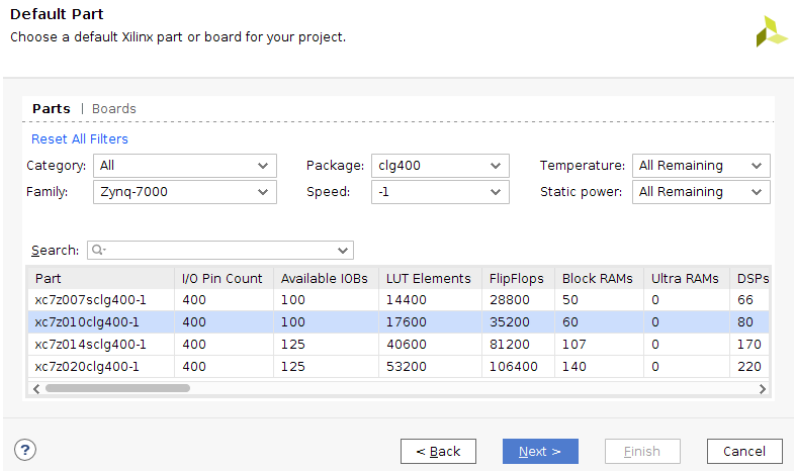

Figure 4: Selecting the Zynq SOC type: the Redpitaya is fitted with a `xc7z010clg400-1` model of the Zynq, hence a Zynq-7000 in a "clg400" package, and a speed grade set to -1.

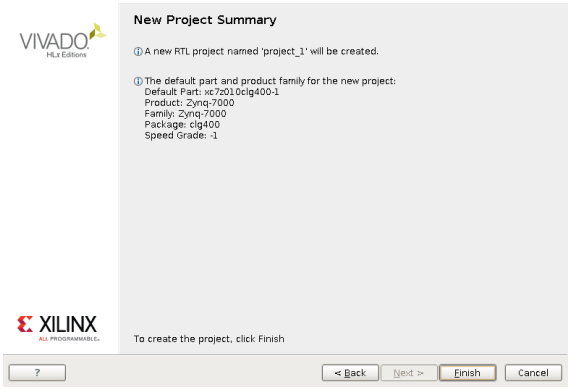

Figure 5: Final summary of the configuration.

2 Creating the block design

The classical approach offered by Vivado is to assemble blocks graphically: while we shall depart later from this approach for large designs, we will use it for the smaller designs of the first tutorials. Assembling IPs graphically is achieved using the *block design* tool.

In the left menu, double click on *Create Block Design*. Selecting the design name does not really matter but will define the final bitstream name: for consistency sake we **advise to use the same name than the name of the project**.

The first item to be added is the *processing system* (representing CPU in the block design). Such a result is achieved by displaying (`CTRL + i` shortcut) a window allowing for the selection of all available IPs. In the list, add *ZYNQ7 Processing System* (search keyword `zynq`). Failing to add this IP, even if not needed, will result in a system freeze when configuring the FPGA from GNU/Linux.

Once this block has been added, a green horizontal bar appears with the text *Run Block Automation*. Running this option will route the few mandatory connections.

At the beginning of a project creation *block design* has no knowledge of the Redpitaya hardware configuration (amount of RAM, peripherals ...): defining such a configuration of the processing system is needed for further work. Such a result is achieved by double-clicking on the *processing system* block: on top of the newly created window, in the `Presets` menu, select *Apply configuration* and load the configuration file `redpitaya.tcl` (or `redpitaya16.tcl` for the 16-bit Redpitaya version [1]) found in the `red_vivado_support` directory of the https://github.com/trabucayre/redpitaya/ repository, or locally at
`/somewhere/oscimpDigital/fpga_ip/preset/redpitaya.tcl`.

In case of error about the clock settings of the Ethernet interface, make sure `LANG="en_US.UTF-8"` or erroneous decimal separators will be used.

3 Configuring Vivado to use custom IPs

`Tools` → `Settings` → `IP` → `Repository` → `+` and add `somewhere/oscimpDigital/fpga_ip`. This operation is completed only once on a given Vivado installation, when accessing for the first time the custom IPs provided by the OscImp project.

[1] In case of the 16-bit Redpitaya, we wish to echo the 16-bit ADC measurement to the 14-bit DAC. A bit-shift – https://github.com/oscimp/oscimpDigital/blob/master/doc/IP/shifter.md – to the right by two bits is necessary to match data bus sizes. This shifter is automatically inserted when generating the Vivado project from the TCL script found in the `design` directory.

4 Inserting a new block in Vivado

Handling ADC, DAC and the associated clocking circuitry is being taken care of by a single processing block: `redpitaya_converters`. This block is designed to handle the legacy 14-bit Redpitaya as well as the newer 16-bit Redpitaya.

Since this design will not allow communicating with the PS, some blocks that will be used later are not added, such as the *axi interconnect* and the *Processor System Reset*. The latter block is however mandatory in the current case since it handles reset signals. Hence, having again hit `CTRL + i`, select *Processor System Reset* (search keyword `reset`). Now connect the `redpitaya_converters` `adc_rst_i` input to the `proc_sys_reset` output named `peripher_reset`.

Furthermore, clock settings must be manually defined since here we do not rely on AXI communication to set these signals automatically as will be done later: connect `FCLK_CLK0` (of ps7) to `M_AXI_GP0_ACLK` (same block) and `slowest_sync_clk` (of `rst_ps7`) ; and connect `FCLK_RESET0_N` of the PS7 to `ext_reset_in`.

Finally, since the functionality of this design is to mirror the ADC input to the DAC output, connect `dataA_in` to `dataA_out` and `dataB_in` to `dataB_out`.

5 Connecting blocks to the FPGA pins

The block describing the ADC, DAC and internal signals must be connected to the FPGA pins (Fig. 1).

Exporting a signal to the outer world is achieved by using the *make external* command obtained by selecting a given signal on a block (the line and its name should turn brown) and right-mouse click, or using the shortcut `CTRL + t`: apply this command to the `phys_interface` of the `redpitaya_converters` block.

The `make external` command we have just used (CTRL+t shortcut) has exported each signal and now requires defining which of the FPGA pins they are connected to. Such constraints are defined by dedicated files with the `.xdc` extension. For the IP we have used in this design, these files are provided in the sub-directory with the IP name in the repository and must be added:

- in the `Sources` tab on the left of the schematic, unwrap `Constraints` and right-click on `constrs_1` (Fig. 6) and select *Add Sources*
- *Add or create constraints*;
- using the "+" button, *Add Files* and select the `xdc` files
  - `redpitaya_converters.xdc` must alway be selected;
  - add either `redpitaya_converters_adc.xdc` or `redpitaya_converters_adc16.xdc` depending whether the legacy (14-bit) or newer (16-bit) Redpitaya is used

  located in the IP directories of the `oscimpDigital/fpga_ip/redpitaya_converters` repository.
- before validating with `Finish`, select *Copy constraints files into project*, otherwise the project will refer to the repository file using absolute paths, preventing the use of the project if moved to another computer or directory (collaborative work).

6 Bitstream generation

The project is now completed, but prior to generating the bitstream a last step is mandatory: creating a wrapper whose function is to assemble the various HDL source codes. This file also provides the `top` file of the design.

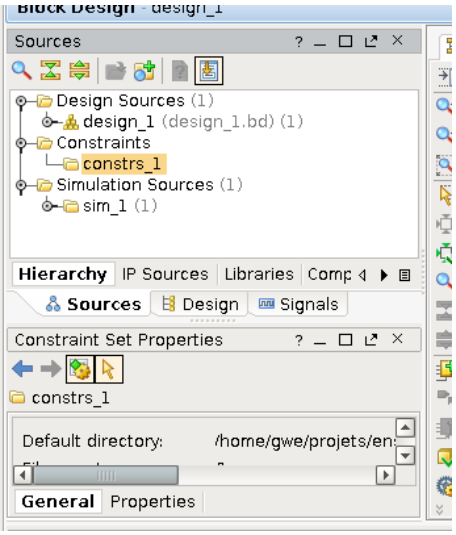

Figure 6: Adding constraints for mapping signals to FPGA pins.

Such a result is achieved by right-clicking in the `Sources` tab the name of the block design (Fig. 7) and selecting `Create HDL Wrapper`. Having completed this step, we click on `Generate Bitstream` in the lower left part of the `Vivado` graphical interface.

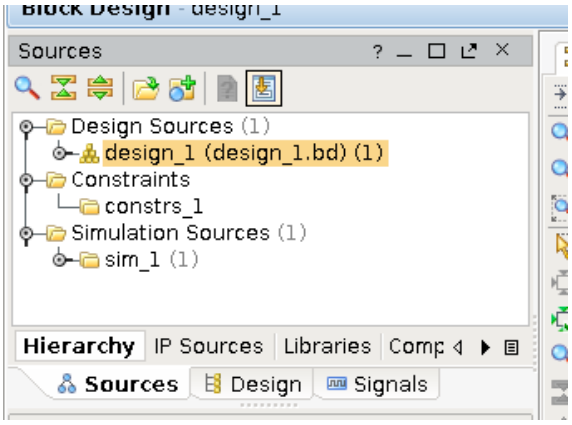

Figure 7: Creating the wrapper (`top` of the design) needed to generate the bitstream

.

7 Signed bitstream and FPGA configuration

The previous steps have ended with the generation of a `.bit` located in the `project_name/project_name.runs/impl_1` directory and called `project_name_wrapper.bit`

7.1 Creating the encrypted bitstream

The default file format of the bitstream generated by Vivado is a `.bit` file. The driver allowing to configure the PL from GNU/Linux requires a specific format including a dedicated header.

Converting from one format to another is achieved by using the `bootgen` tool provided by the Vivado SDK.

This tool expects a configuration file with a `.bif` extension and filled with

```
all:
{
  bitstream_name.bit
}
```

so that the following command is executed in the `project_name.runs/impl_1` subdirectory of the Vivado project

```
bootgen -image bif_file.bif -arch zynq -process_bitstream bin
```

Following this command, a file named `bitstream_name.bit.bin` is generated in the working directory.

7.2 Configuring the PL by using `fpga_manager`

GNU/Linux provides a homogeneous framework for configuring the FPGA of SoC chips: `fpga_manager`. This framework expects the `.bit.bin` file to be located in the `/lib/firmware` of the target platform.

Once the file is in the right location, the driver must be informed that the FPGA must be configured and which bitstream to use:

```
echo "bitstream_name.bit.bin" > /sys/class/fpga_manager/fpga0/firmware
```

which results in

```
fpga_manager fpga0: writing bitstream_name.bit.bin to Xilinx Zynq FPGA Manager
```

being displayed in the console or in `/var/log/syslog` and the LED (blue on the Redpitaya platform) connected to `Prog done` will be lit.

7.3 Using the `devicetree` overlay for PL configuration

The devicetree overlay provides an alternative solution for configuring the FPGA in which all necessary resources – driver name, address space and bitstream name – are referenced in a single file and communicated to the kernel module. For the purpose of this design, this solution is oversized but offer a coherent approach with next tutorials, where Axi based IPs are used.

Similar to the previous method, the bitstream must be located in `/lib/firmware`.

Without getting in the details of the devicetree overlay format, the following code aims at modify `fpga_full` node, defined at board's default devicetree, to provide, through attribute `firmware-name`, the bitstream name.

```
/dts-v1/;
/plugin/;
/ {
    compatible = "xlnx,zynq-7000";
    fragment@0 {
        target = <&fpga_full>;
        #address-cells = <1>;
        #size-cells = <1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;

            firmware-name = "bitstream_name.bit.bin";
        };
    };
};
```

This file is compiled by using the following command

```
/somewhere/buildroot/output/host/bin/dtc -@ -I dts -O dtb -o ${FILENAME}.dtbo ${FILENAME}.dts
```

in which

- `-@` requires generating symbols that will be dynamically linked when loaded,
- `-I dts` defines the format of the input file,

- `-O dtb` defines the format of the output file,
- `-o` the name of the generated file.

Loading this file in memory is achieved in two steps:

1. creating a directory hosting our overlay
   ```
   mkdir /sys/kernel/config/device-tree/overlays/myname
   ```
   will create a directory automatically filled with the files needed to communicate with the driver
   ```
   redpitaya> ls -l /sys/kernel/config/device-tree/overlays/myname/
   total 0
   -rw-r--r-- 1 root root 0 Jan 1 00:04 dtbo
   -rw-r--r-- 1 root root 4096 Jan 1 00:04 path
   -r--r--r-- 1 root root 4096 Jan 1 00:04 status
   ```
2. loading the overlay in the devicetree :
   ```
   cat gpio_red.dtbo > /sys/kernel/config/device-tree/overlays/myname/dtbo
   ```
   will configure the PL by transferring the bitstream, insert, if needed, the associated module driver as defined by the "compatible" field which must be filled with a matching string in the driver.

Returning to a state where the overlay functionalities are removed is achieved by erasing the directory:
```
rmdir /sys/kernel/config/device-tree/overlays/myname
```

8 Demonstration

Connect a signal synthesizer to the ADC analog input and an oscilloscope to the DAC analog output. The output must mirror the input with a delay due to the signal propagation in the FPGA, as shown in Fig. 8.

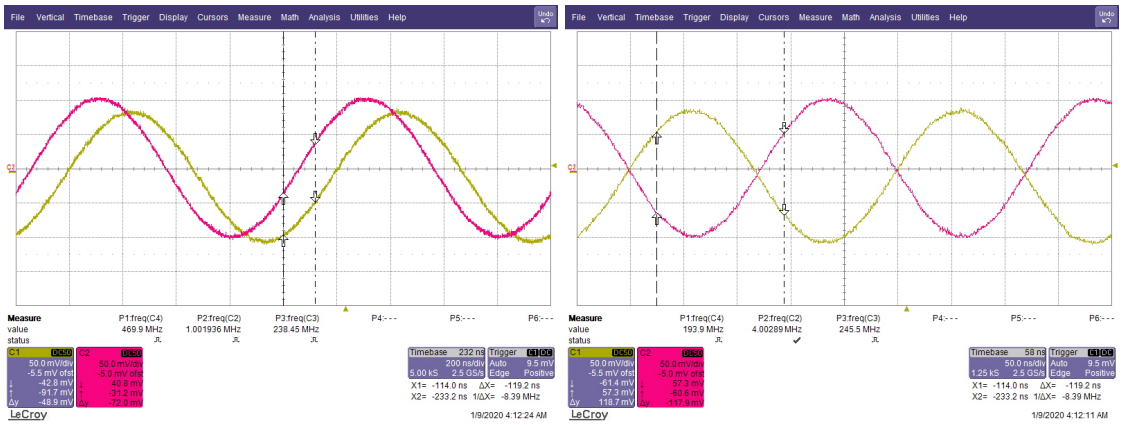

Figure 8: Left: 1-MHz signal mirrored from ADC to DAC. Right: 4 MHz signal, with a 180° phase rotation indicative of the propagation delay.