

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

G. Goavec-Mérou

November 21, 2018

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).

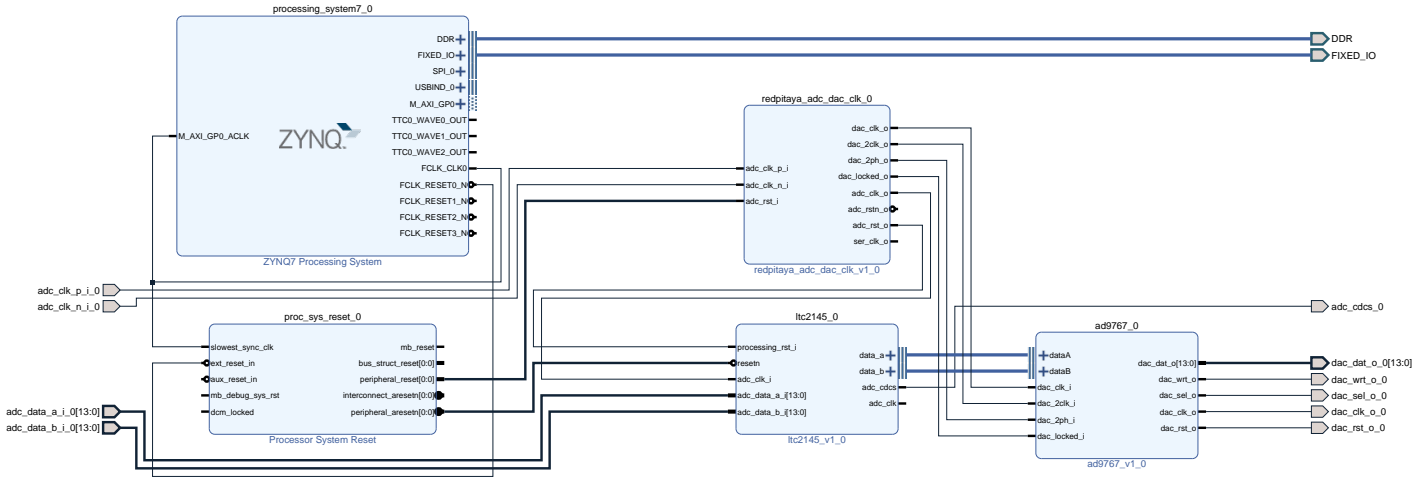
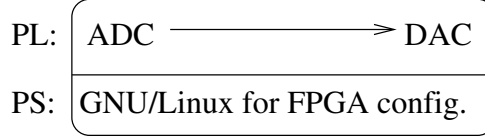


Figure 1: Objective of the tutorial (top) and block design (bottom) including the processor, the reset signals, and ADC and a DAC, as well as the clock generation block for the latters.

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).

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name:

Project location:

☒ Create project subdirectory

Project will be created at: /home/gwe/projects/enseignement/redpitaya/project_1

Figure 2: Selecting the project name and storage location

Project Type
Specify the type of project to create.

☒ **RTL Project**
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
☒ Do not specify sources at this time

☐ **Post-synthesis Project:** You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time

☐ **I/O Planning Project**
Do not specify design sources. You will be able to view part/package resources.

☐ **Imported Project**
Create a Vivado project from a Synplify, XST or ISE Project File.

☐ **Example Project**
Create a new Vivado project from a predefined template.

Figure 3: Selecting the project type.

Default Part
Choose a default Xilinx part or board for your project. This can be changed later.

Select: ☒ Parts ☐ Boards

Filter

Product category: Speed grade:

Family: Temp grade:

Package:

Search:

Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GTPE2 Transceivers	GTXE2 Transceivers	Gb Transceivers	Availa IOBs
xc7z010clg400-1	400	60	80	35200	0	0	0	100
xc7z020clg400-1	400	140	220	106400	0	0	0	125

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: Fenêtre récapitulative.

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` 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`.

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.

4 Inserting new blocks in Vivado

Three processing blocks must be included to transfer the ADC input to the DAC output:

- the ADC description `1tc2145`
- the `ad9767` DAC,
- the `redpitaya_adc_clk` block handling clocks for the ADC and DAC

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**).

5 Connecting blocks to the FPGA pins

The three blocks 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**.

The following signal must be exported:

- for the ADC:
 - `adc_data_a_i`;
 - `adc_data_b_i`;
 - `adc_cdcs`.
- for the DAC:
 - `dac_dat_o`;
 - `dac_wrt_o`;
 - `dac_sel_o`;
 - `dac_clk_o`;
 - `dac_rst_o`.
- for the clock signal handling block:
 - `adc_clk_p_i`;
 - `adc_clk_n_i`;

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 three IPs we have used in this design, these files are provided in their respective sub-directories 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
 - `ad9767.xdc`;
 - `ltc2145-redpy.xdc`;
 - `redpitaya_clk_pin.xdc`

located in the IP directories of the `oscampDigital/fpga_ip` 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).

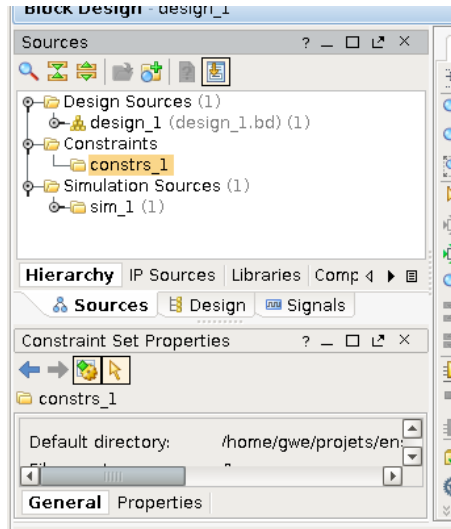


Figure 6: Adding constraints for mapping signals to FPGA pins.

6 Connecting clock and reset signals

The ltc2145 and ad9767 blocks require clock and reset signal to operate properly:

For the ADC:

- `processing_rst_i` must be connected to the `adc_rst_o` signal of the `redpitaya_adc_clk`;
- `resethn` must be connected to the `peripheral_aresethn` signal of the Processor system Reset;
- `adc_clk_i` must be connected to the `adc_clk_o` signal of `redpitaya_adc_clk`.

For the DAC:

- `dac_clk_i` is connected to `dac_clk_o` of `redpitaya_adc_clk`;
- `dac_2clk_i` is connected to `dac_2clk_o` of `redpitaya_adc_clk`;
- `dac_2ph_i` is connected to `dac_2ph_o` of `redpitaya_adc_clk`;
- `dac_locked_i` is connected to `dac_locked_o` of `redpitaya_adc_clk`.

For the clock signal generator:

- `adc_rst_i` de `redpitaya_adc_dac_clk` is connected to `peripheral_reset` of `proc_sys_reset`

Connect the output of the ADC to the input of the DAC:

- `data_a` of the ltc2145 is connected to `dataA` of ad9767
- `data_b` of the ltc2145 is connected to `dataB` of ad9767

7 Other connections

Designs communicating between PL and PS through the AXI bus allow for automated routing of some of the signals. Lacking such functionalities here, we must connect manually:

- the `FCLK_CLK0` signal of `processing_system7_0` is connected to

- M_AXI_GPO_ACLK signal of the same block,
- slowest_sync_clk signal of the rst_processing_system7_0_125M block.
- the signal FCLK_RESET0_N of processing_system7_0 is connected to the ext_reset_in signal of the rst_processing_system7_0_125M block.

8 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.

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.

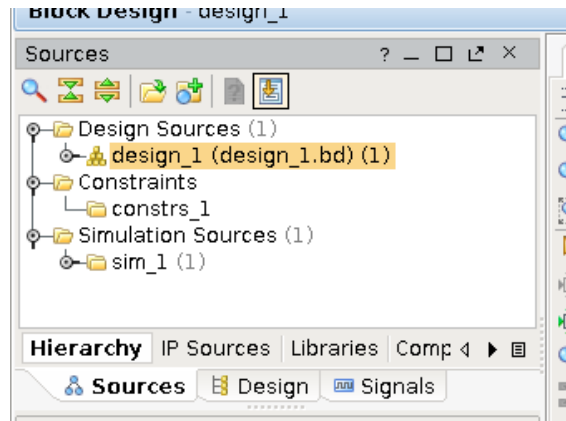


Figure 7: Creating the wrapper (top of the design) needed to generate the bitstream

9 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`

9.1 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

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

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

9.2 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.

9.3 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
```