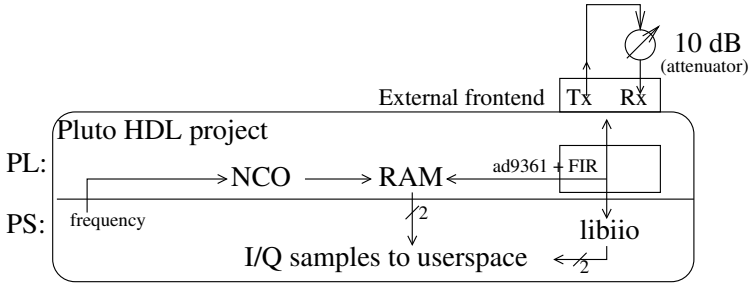
Using ADALM-PLUTO with the OscImpDigital Ecosystem

P.-Y. Bourgeois, G. Goavec-Merou, J.-M Friedt
{pyb2,gwenhael.goavec,jmfriedt}_at_femto-st.fr
June 16, 2019

This tutorial is intended to help you to handle the ADALM-PLUTO (PlutoSDR) board within the OscImpDigital framework. It is focused on

- Providing general guidelines to set up the environment
- Performing a PlutoSDR's harware independant check (NCO→RAM, check the data)
- Connecting the PlutoSDR data stream into the RAM and recovering the data



① nco_counter → dataComplex_to_ram

② ad9361 → fir_decimator → axiStreamToComplex → dataComplex_to_ram
fir_decimator → axi_ad9361_adc_dma

Figure 1: Objectives of this tutorial and raw schematics of the processing chains.

1 Objectives / Experimental set-up

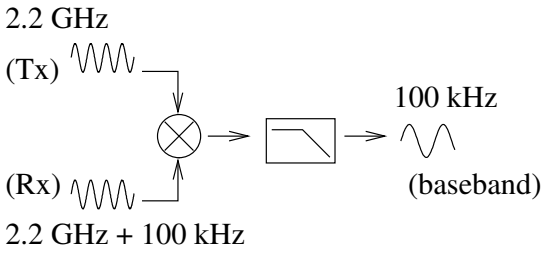
Right out of the box, the ADALM-PLUTO board, when fired up, is able to be fully configurable and flawlessly accessible thanks to the *iio* contexts of **libiio**. Analog Devices Inc. (ADI) provides the library but also distributes the HDL code. It becomes then interresting to test the compatibility of the original PlutoSDR HDL project with the OscImpDigital ecosystem and benefit from extra features by pre-processing the radiofrequency datastream on the high bandwith Programmable Logic (PL) FPGA of the Zynq processor before transferring the lower bandwith dataset to the Processing System (PS).

For this tutorial, we want to recover I/Q samples of a 100 kHz beatnote. Achieving this result is first assessed by fetching samples from the output of a 100 kHz NCO operating independently of the AXI Stream provided by ADI. Having checked that the samples are properly generated and recovered, we mix the output of the AXI Stream including the I/Q coefficients from the AD9363 and check that the result is consistent. Thus:

- the first part aims at checking the correct bitstream synthesis of the project provided by ADI using an additional NCO and RAM transfer from the OscImpDigital ecosystem. In this setup (called sanity check), the NCO acts as a fake 100 kHz beatnote generator, the added IPs having no relationship with the initial firmware,

- in the second part, we connect custom blocks to the AXI Stream from the AD9363 and demonstrate how custom processing can be added to the original ADI processing chain.

In order to create a "beatnote", one can use the following setup:



The **Tx** port of the PlutoSDR board is set to transmit at 2.2 GHz. When connecting the PlutoSDR Sink (select an input level lower than 1 to avoid saturation and unwanted signal beyond the carrier), the Tx transmits the unmodulated carrier at 2.2 GHz (i.e. with no signal of interrest nor message). The receiver part is slightly detuned at 2.2 GHz+100 kHz. After demodulation (RF-hardware) and decimating filter (`fir_decimator` block within the PL-side of the FPGA), a remaining "beatnote" at $f_b = 100$ kHz occurs. Once we know in advance the frequency of this beatnote, when the sampling rate is set up at $fs = 2$ MHz (as defined in the application to be discussed in section 3.3), we also know that a window of $N = 2048$ samples will present $N \times fb/fs \simeq 102$ periods of the beatnote. This "beatnote" trick is routinely used when developping RF applications, to check that collected data are sound.

On the photography, a splitter has been added (before the 10 dB attenuator) to also send the **Tx** port output to a spectrum analyzer which allows to check if the right carrier frequency has been set up through *iio*.

2 Setting up the environment

Up to date versions of the following repositories must be fetched:

- The BR2_EXTERNAL framework for Analog Device's PlutoSDR Zynq (https://github.com/oscimp/PlutoSDR) and compile the Buildroot environment following the instructions provided in the github repository `README.md` file,
- The HDL project from Analog Devices (see further below). In order to build the HDL project for the ADALM-PLUTO board, only the **hdl** repository (https://github.com/analogdevicesinc/hdl) is needed.
- OscImpDigital (https://github.com/oscimp/oscimpDigital). Remember to recursively clone the repository
  `git clone --recursive https://github.com/oscimp/oscimpDigital`
  and set the `BOARD_NAME` to `plutosdr` in the `settings.sh` script. Although the PlutoSDR is fitted with the same Zynq model than the Redpitaya, this variable will allow us to execute commands such as generating the DFU image or flashing the embedded board. Furthermore, we will have to inform where the ADI HDL repository is located by defining the `ADI_HDL_DIR` variable in addition to `BR_DIR` indicating where the BuildRoot repository is located.

At the time of writing (June 16, 2019), the following versions are functional:

- **adi_hdl** branch **hdl_2018_r2** (401395cdd1980827fd1f7043ce1a10770f666c64)

- Vivado 2018.3

3 Sanity check (NCO→RAM→UserSpace)

This tutorial allows us to use the OscimpDigital blocks with the embedded Zynq of the PlutoSDR board. It is seen as a first demonstration of compatibility.

3.1 Design

To build the HDL project for the PlutoSDR board, `source` the `settings64.sh` script of Vivado 2018.2, the OscImpDigital `settings.sh`. Go to the first tutorial directory located in `oscimpDigital/doc/tutorials/plutosdr/1-adalmPluto_within_OscimpDigital` and from the `project1/design` sub-directory of this tutorial, run `make xpr`.

A successful build of the HDL project is returned by the console message:
`Building` **pluto** `project [[...]/project1/design/pluto_vivado.log] ...` **OK**

Once the pluto HDL project is successfully generated, open *pluto.xpr* within Vivado (`vivado pluto.xpr`).

Check if the *oscimpDigital/fpga_ip* repository is present in the list of IP repository (Icon "Settings" → "Project Settings" → "IP" → "Repository" →).

Open the block design, right-click, "Add IP", then add the *dataComplex_to_ram* and *nco_counter*. Run the "Connection Automation": the NCO counter and dataComplex_to_ram should connect to the AXI bus and have addresses assigned to their registers: dataComplex_to_ram base address should be 0x43C1_0000 and nco_counter base address should be 0x43C0_0000. Refer to `Address Editor` to verify, and if adresses mismatch, keep them in mind as you will need it for the Device Tree overlay.

The next step is to configure the added IPs: we set (double click on each block icon)

- the NCO block `Data Size` to 16 bits, `Counter Size` to 28 bits, and `LUT Size` to 10 bit,
- the Data Complex to RAM block with `Data Size` to 16 bits, `Number of Inputs` to 1, and `Number of Samples` to 1024.

Additional connections (Fig. 2) for control signals:

- NCO
  - `ref_rst_i` → `s00_axi_reset` on the same block
  - `ref_clk_i` → `s00_axi_aclk` which refers to the PS7's FCLK_CLK0 at 100 MHz. We might as well clock and reset all blocks on the AD936x signals (AD936x clock frequency depends on sampling frequency configured through IIO API or GNURadio flowgraph).
- NCO → RAM
  - `sine_out` → `data1_in`

At last, click on "Generate Bitstream".

Bistream generation is quite and lengthy task (depending on the host CPU). This time is best spent compiling OscImpDigital libraries and drivers as well as creating application for the PS as described in the next section.

3.2 Preparing all OscimpDigital lib/driver/tools mandatories

The NCO needs to be configured: rather than directly interacting with low level registers, the simpliest way is to benefit from the functions provided by the OscImpDigital library to do this task.

To build the library, go to `$OSCIMP_DIGITAL_LIB` and use the command:

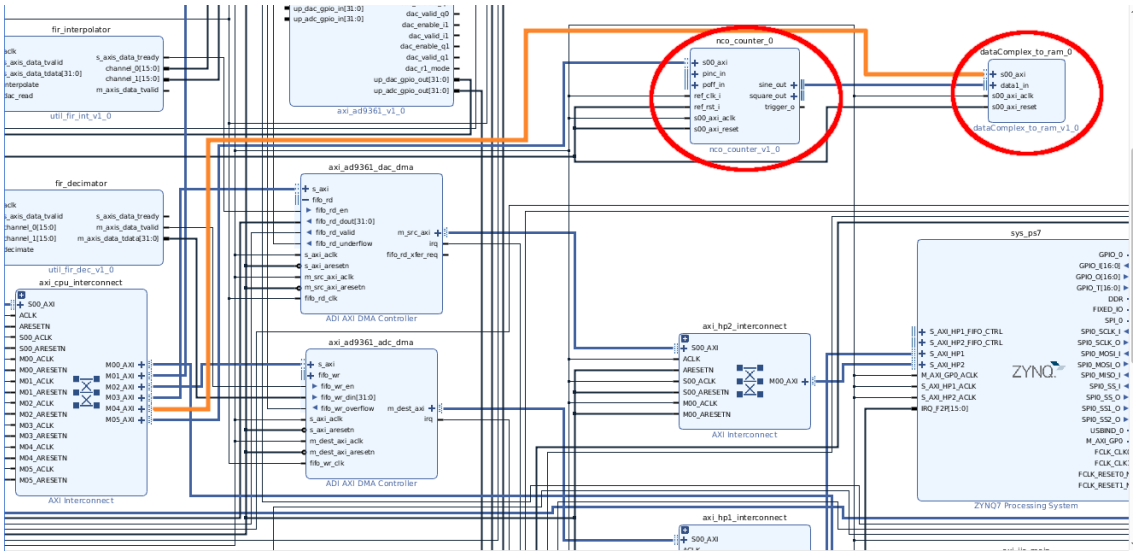


Figure 2: Connection of the custom blocks to the PlutoSDR firmware.

```
make
```

After that the current directory must contain two files: `liboscimp_fpga.so` and `liboscimp_fpga_static.a`: the latter can be linked with an application by defining at `USE_STATIC_LIB=1` option as will be discussed below.

Communications between PS userspace applications and IPs in the PL are achieved thanks to Linux drivers. To build mandatory drivers go to `$OSCIMP_DIGITAL_DRIVER` and use

```
make
```

as this command will iterate on all subdirectories and build each driver.

The last element to build is `module_generator`. This tool will be used to build a skeleton for applications based on an XML description (see bellow). Go to `$OSCIMP_DIGITAL_APP/tools/module_generator` and again use

```
make
```

3.3 Userspace application

`module_generator` (https://github.com/oscimp/app/tree/master/tools/module_generator) is used to generates a skeleton for the application:

- `Makefile` to cross-compile and install application and devicetree overlay;
- devicetree source `dts` file to inform the linux kernel about drivers to load and base memory area to communicate with IP in PL;
- script to prepare everything (load drivers and devicetree) before using the binary application.

This tool uses a XML file which describes the list of IPs and its base address. This file may contain options (if the OscImpDigital library is statically or dynamically linked or not used, if the application must be linked with others libraries, ...).

The configuration file (`project1/module_generator.xml`) for this first application looks like

```
<?xml version="1.0" encoding="utf-8"?>
<project name="project1" version="1.0">
  <options>
    <option target="makefile" name="USE_STATIC_LIB">1</option>
    <option target="makefile" name="LDFLAGS">-liio</option>
```

```
  </options>
  <ips>
    <ip name="dataComplex_to_ram" >
      <instance name="data00" id="0" base_addr="0x43c10000" addr_size="0xffff" />
    </ip>
    <ip name="nco_counter" >
      <instance name="nco00" id="0" base_addr="0x43c00000" addr_size="0xffff" />
    </ip>
  </ips>
</project>
```

where `data00` and `nco00` are the `/dev` entries and the base addresses should match those provided by Vivado. The `USE_STATIC_LIB` option is used to add a constant in the `Makefile` to specify that the static library should be linked with the application and LDFLAGS specify a dynamic link to the libiio since we will use IIO API to configure RF frontends (see bellow).

The command:
`$OSCIMP_DIGITAL_APP/tools/module_generator/module_generator module_generator.xml`
will create a new directory called `app` and fill it with the `Makefile`, a script called `project1_us.sh` and the `dts` file `project1.dts`.

The final task is to write the application behaviour `project1/app/main.c`

The application-side makes use of *libiio* as well as *liboscimp*. For the former, one can find examples from M. Hennerich presentation at GNU Radio Conference 2018 [1] or following guidelines written in the ADI wiki website [2] [3] to tweek the configuration.

For this part, we have just changed Hennerich's snippet code to the desired configuration of TX and RX carriers, sampling rate, while the processing part only consists in writing the RAM buffer into a data file.

```
1  [...]
2  #include <iio.h> /* IIO API */
3  #include <nco_conf.h> /* NCO function */
4
5  #define ELEMENT_SIZE 1024 // Nb Sample
6
7  #define CLK_FREQ 2000000
8  #define MOD_FREQ 100000
9  #define NCO_ACCUM_SIZE 28
10
11 int main()
12 {
13   int16_t *rawData;
14   int ramfd = 0, i;
15
16   struct iio_device *dev, *phy;
17   struct iio_context *ctx;
18   struct iio_channel *rx0_i, *rx0_q;
19
20   rawData = (int16_t *) malloc(2 * ELEMENT_SIZE * sizeof(int16_t));
21
22   ctx = iio_create_local_context();
23
24   dev = iio_context_find_device(ctx, "cf-ad9361-lpc");
25   phy = iio_context_find_device(ctx, "ad9361-phy");
26
27   iio_channel_attr_write_longlong(iio_device_find_channel(phy, "altvoltage1", true),
28     "frequency", 2200000000); /* TX LO frequency 2.4GHz */
29
30   iio_channel_attr_write_longlong(iio_device_find_channel(phy, "altvoltage0", true),
31     "frequency", 2200100000); /* RX LO frequency 2.4GHz + 100 kHz */
32
```

[1] www.gnuradio.org/grcon/grcon18/presentations/plutosdr/
[2] wiki.analog.com/resources/tools-software/linux-drivers/iio-transceiver/ad9361
[3] wiki.analog.com/university/tools/pluto/controlling_the_transceiver_and_transferring_data

```
33   iio_channel_attr_write_longlong(iio_device_find_channel(phy, "voltage0", false),
34     "sampling_frequency", CLK_FREQ); /* RX baseband rate 2 MSPS */
35
36   rx0_i = iio_device_find_channel(dev, "voltage0", 0);
37   rx0_q = iio_device_find_channel(dev, "voltage1", 0);
38
39   iio_channel_enable(rx0_i);
40   iio_channel_enable(rx0_q);
41
42   /* NCO configuration */
43   nco_counter_send_conf("/dev/nco00", CLK_FREQ, MOD_FREQ,
44               NCO_ACCUM_SIZE, 0, 1, 1); // 0, 1, 1 => offset, PINC HW/SF, POFF HW/SF
45
46   ramfd = open("/dev/data00", O_RDONLY);
47   if (ramfd < 0) {
48     perror("ram open error\n");
49     return EXIT_FAILURE;
50   }
51   read(ramfd, rawData, 2 * ELEMENT_SIZE * sizeof(int16_t));
52   FILE *fd = fopen("data.dat", "w");
53   for (i = 0; i < 2 * ELEMENT_SIZE; i+=2)
54     fprintf(fd, "%d %d\n", rawData[i], rawData[i+1]);
55   fclose(fd);
56   close(ramfd);
57   iio_context_destroy(ctx);
58   free(rawData);
59   return EXIT_SUCCESS;
60 }
```

The most important lines for this first project where we only fetch the data stream produce by the NCO are:

- lines 33-34: set the sample frequency (here 2 MS/s for consistency, although sampling rates up to 64 MS/s can be achieved between the AD9363 and the Zynq). This frequency corresponds to the `l_clk` output from `axi_ad9361` IP and is used by the NCO though `ref_clk` input for clocking frequency.
- lines 43-44: `nco_counter_send_conf` is the OscImpDigital library function to configure the NCO IP. We found:
  - a `/dev/nco00` according to the dev name provided in the XML file;
  - the same constant used to configure the sampling frequency and here specifying the IP clocking frequency to compute phase increment work;
  - the output, requested, frequency corresponding to the beatnote between TX LO frequency and RX LO frequency;
  - the phase accumulator size again used to compute the phase increment;
  - and, but out of the scope of this tutorial, a phase offset, and a mux configuration to driver phase increment and offset from AXI or from NCO IP inputs
- finally lines 46-51: the `/dev/data00` is opened and a simple read is used to fetch samples from data stream. we read 16bits $\times 1024 \times 2$ (the $\times 2$ is due to the complex I/Q samples)

`make` will produce both `project1.dtbo` and `project1_us`.

3.4 Update PlutoSDR firmware and install application

Now we have a `.bit` for the PL and application for the PS.

First a new firmware image must be produced containing the custom bitstream. In the `design` directory the command `make dfu_frm` will produce both `.dfu` and `.frm` images.

Two firmware flashing strategies can be followed:

- using the `.frm`, we copy the file to the PlutoSDR mass storage and reboot by:

- `sudo mount /dev/sdx /mnt/sd` (see `dmesg` for corresponding device)
- `cp image/pluto.frm /mnt/sd`
- `sudo eject /dev/sdx`

You should see the **blue LED1** flashing quickly: **do not disconnect nor touch anything until the PlutoSDR has rebooted** or you may brick the board at this stage.

- using the `.dfu:` use the command `make flash_dfu_frm` to reboot the board in dfu mode (password will be asked: analog) and to flash the `pluto.dfu` image. This option is prefered for large images which sometimes do not fit in the remaining mass storage space.

  When `dfu-util` displays:

```
_firmware
Copying data from PC to DFU device
Download        [=========================] 100%     30669815 bytes
Download done.
state(7) = dfuMANIFEST, status(0) = No error condition is present
state(2) = dfuIDLE, status(0) = No error condition is present
Done!
```

  unplug and replug the PlutoSDR.

Now that the firmware including the updated bitstream has been flashed, the PlutoSDR is ready to be tested.

In the `app` directory use:

```
make install_ssh
```

Again, the PlutoSDR root password (`analog`) is required.

To test the application:

```
ssh root@192.168.2.1
```

and go to the directory in which the applications were stored, namely a sub-directory of `/tmp`:

```
cd /tmp/project1/bin/
```

3.5 Testing the application

Before using the application, dtbo overides must be applied and drivers loaded: this is done by executing `./project1_us.sh`. The command `dmesg` must display something like:

```
[...]
data_to_ram_core: loading out-of-tree module taints kernel.
probing data00 with dts
name: data00 4 6
dataToRam 43c00000.data00: data00: Add the device to the kernel, connecting cde
dataToRam 43c00000.data00: 6data00 loaded
probing nco00
name: nco00 4 5
nco_counter 43c10000.nco00: nco00: Add the device to the kernel, connecting cde
nco_counter 43c10000.nco00: 6nco00 loaded
```

with the first two lines exhibiting kernel messages resulting from the bitstream loading, then the `data_to_ram` driver and the `nco` driver. The two `/dev/nco00` and `/dev/data00` defined in the devicetree source should have been created.

Now launch `./project1_us`: the `/dev/data00` device is opened, the output of the NCO fed to this block read from the PS, and stored in a file. Plotting the resulting dataset yields charts similar to those shown in Fig. 3.

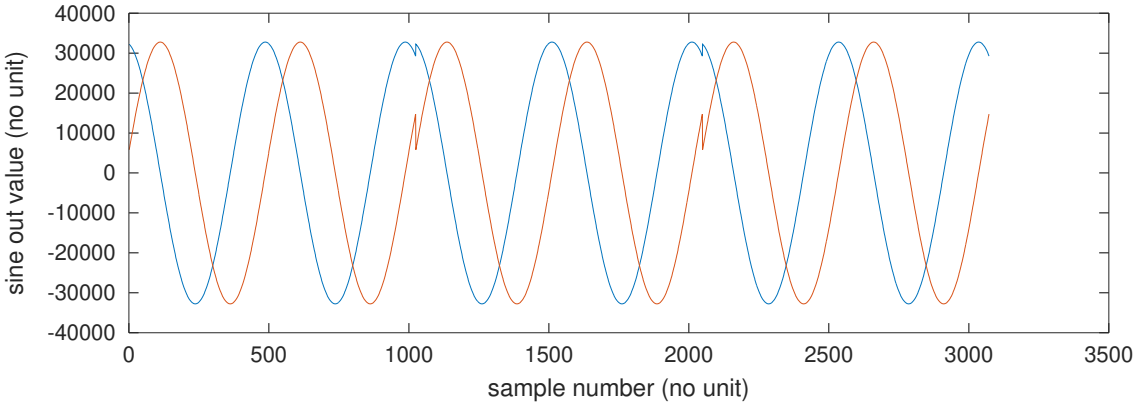


Figure 3: Collected I/Q datasets. The 200 samples/period match the expected sampling rate of 2 MS/s as a 100 kHz sine wave is generated by the NCO. Three successive (discontinuous) datasets were collected.

4 PlutoSDR data stream → RAM → Userspace

4.1 Design update

This second tutorial is a demonstration of the compatibility of the OscImpDigital ecosystem with the use of the existing Pluto HDL. Once complete, you should be able to plug any DSP task to the initial design and perform extra processing chains (Figs. 4 and 5).

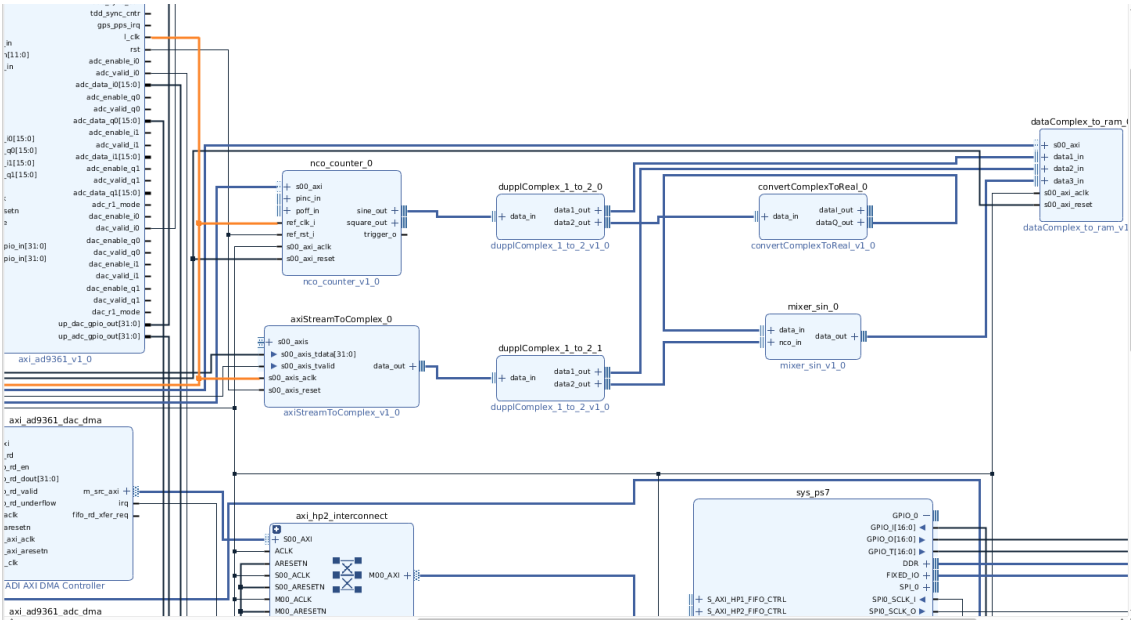


Figure 4: Signal processing chain as defined in the Vivado 2018.3 graphical user interface (added blocks are on the top right and include the nco_counter, axiStreamToComplex, two duppComplex, convertComplextoReal, mixer_sin and finally dataComplex_to_ram).

This project is an update of the previous one. Instead of only fetching data stream generated by the NCO, we will use this data flow to frequency transpose the AD9361 data stream by mixing. A second update is to be done: three datastreams lead to the DataToRAM block: the (complex) NCO output as addressed previously, the raw ADC (complex) data resulting from the

I/Q demodulation within the AD9363 RF frontend, and the output of the mixer of the NCO with the ADC I/Q stream.

This time, since data from the AD9363 will be used in the flow chart, a consistent clocking and signal reset distribution must be used:

- connect `ref_clk_i` of the `nco_counter` to the `axi_ad9361` IP `l_clk` (as is the `s00_axis_aclk` of the `axiStreamToComplex` block)
- connect `ref_rst_i` of the `nco_counter` to the `axi_ad9361` IP `rst` (as is the `s00_axis_reset` of the `axiStreamToComplex` block)
- these clock and reset signals are then propagated to the other processing blocks through their interfaces.

Failing to use these connections will result in timing errors including negative WNS and TNS values related to the fact that the mixer is fed data from two different clock domains – 100 MHz for the AXI bus and 50 MHz for the AD9361.

We must add these IPs:

- *axiStreamToComplex*: since ADI IP uses AXI Stream interfaces and OscImpDigital custom real or complex interfaces, this IP must be used to convert interfaces. According to the RF frontend bus size, this block must be configured with `Data Size = 16`;
- 2× *dupplComplex_1_to_2*: when connection between two IPs is done using interfaces (instead of manually connecting wire/bus contained in an interface) a relation 1 to many is forbidden. The dupplXX IP acts as a splitter to copy inputs to both ouput ports. These two IPs must configured with `Data Size = 16`;
- *mixerComplex_sin*: this IP is used to apply a complex multiplication between the two input port data flow. This IP must be configured with `Data Size = 16` according to the RF frondend size, and `Nco Size = 16` according to the NCO configuration.

Now it is time to connect blocks, but before that the connection between `NCO` and `dataComplex_to_ram` set previously must be suppressed and the `dataComplex_to_ram` configuration must be modified to change `NB input` to 3.

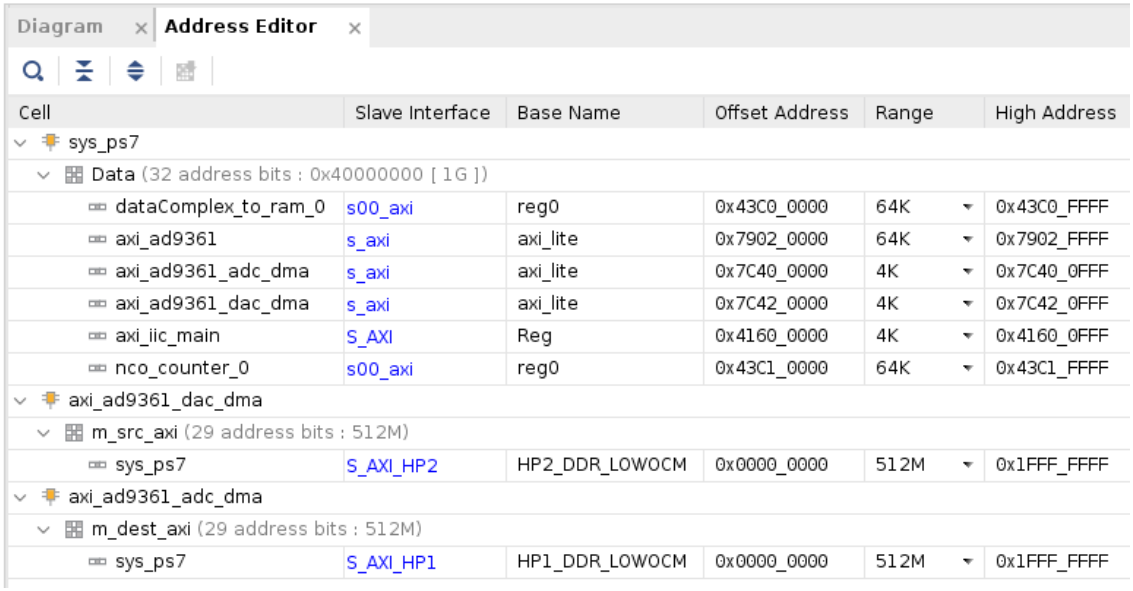
For connecting:

- `fir_decimator` → `axiStreamToComplex`: click on `'+'` before `axiStreamToComplex` `s00_axis` interface to expose all signals included in this interface. Connect:
  - `fir_decimator` `m_axis_data_tdata` → `axiStreamToComplex` `s00_axis_tdata`;
  - `fir_decimator` `m_axis_data_tvalid` → `axiStreamToComplex` `s00_axis_tvalid`
- `axiStreamToComplex` → `dupplComplex_XX_0`: connect `axiStreamToComplex` `data_out` → `dupplComplex_XX_0` `data_in`;
- `nco_counter` → `dupplComplex_XX_1`: connect `nco_counter` `sine_out` → `dupplComplex_XX_1` `data_in`;
- `dupplComplex_XX_0` → `mixerComplex`: connect `dupplComplex_XX_0` `data1_out` → `mixerComplex` `data_in`;
- `dupplComplex_XX_1` → `mixerComplex`: connect `dupplComplex_XX_1` `data1_out` → `mixerComplex` `nco_in`;
- `dupplComplex_XX_0` → `dataComplex_to_ram`: connect `dupplComplex_XX_0` `data2_out` → `dataComplex_to_ram` `data1_in`;
- `dupplComplex_XX_1` → `dataComplex_to_ram`: connect `dupplComplex_XX_1` `data2_out` → `dataComplex_to_ram` `data2_in`;

- `mixerComplex_sin` → `dataComplex_to_ram`: connect `mixerComplex_sin` `data_out` → `dataComplex_to_ram` `data3_in`;

Additional connections for clock and reset signals:

- `axiStreamToComplex` `s00_axis_aclk` → `axi_ad9361` `l_clk`;
- `axiStreamToComplex` `s00_axis_reset` → `axi_ad9361` `rst`;



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
sys_ps7					
Data (32 address bits : 0x40000000 [1G])					
dataComplex_to_ram_0	s00_axi	reg0	0x43C0_0000	64K	0x43C0_FFFF
axi_ad9361	s_axi	axi_lite	0x7902_0000	64K	0x7902_FFFF
axi_ad9361_adc_dma	s_axi	axi_lite	0x7C40_0000	4K	0x7C40_0FFF
axi_ad9361_dac_dma	s_axi	axi_lite	0x7C42_0000	4K	0x7C42_0FFF
axi_iic_main	S_AXI	Reg	0x4160_0000	4K	0x4160_0FFF
nco_counter_0	s00_axi	reg0	0x43C1_0000	64K	0x43C1_FFFF
axi_ad9361_dac_dma					
m_src_axi (29 address bits : 512M)					
sys_ps7	S_AXI_HP2	HP2_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
axi_ad9361_adc_dma					
m_dest_axi (29 address bits : 512M)					
sys_ps7	S_AXI_HP1	HP1_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF

Figure 5: Address map generated by Vivado: the devicetree source file must match this configuration for the Linux drivers to reach registers at the right addresses.

Once these modifications are done, click on `Generate bitstream` and switch to application update while *Vivado* is working.

4.2 Application update

Compared to the design, updates to apply to the PS part is small:

- since we want to feedback TX RF signal to the RX RF input, we will use GNU Radio on the host computer to handle TX stream and to configure the *AD936x* so we must remove all lines related to the RF frontend configuration though IIO in the C source code.
- using USB communication is a bottleneck, which is the reason why since the beginning of this tutorial we have selected a sampling frequency if 2 MS/s. This setting must be applied to the GNU Radio flowchart `samp_rate`. Check that this value is selected as NCO clock frequency (`#define CLK_FREQ`);
- in the previous project, `dataComplex_to_ram` stored and transmitted 1 channel, 1024 × 16 bit complex integers. In this design update we have 3 input channels. All references to buffer sizes must be updated accordingly.

4.3 Testing

Flashing firmware and application install to the PlutoSDR are exactly the same as previously, so please refer to this section.

The resulting datasets are plotted on Fig. 6, left with the PlutoSDR disconnected from its input (noise measurement on the ADC) and right with the RF output connected to the input through a 10 dB attenuator.

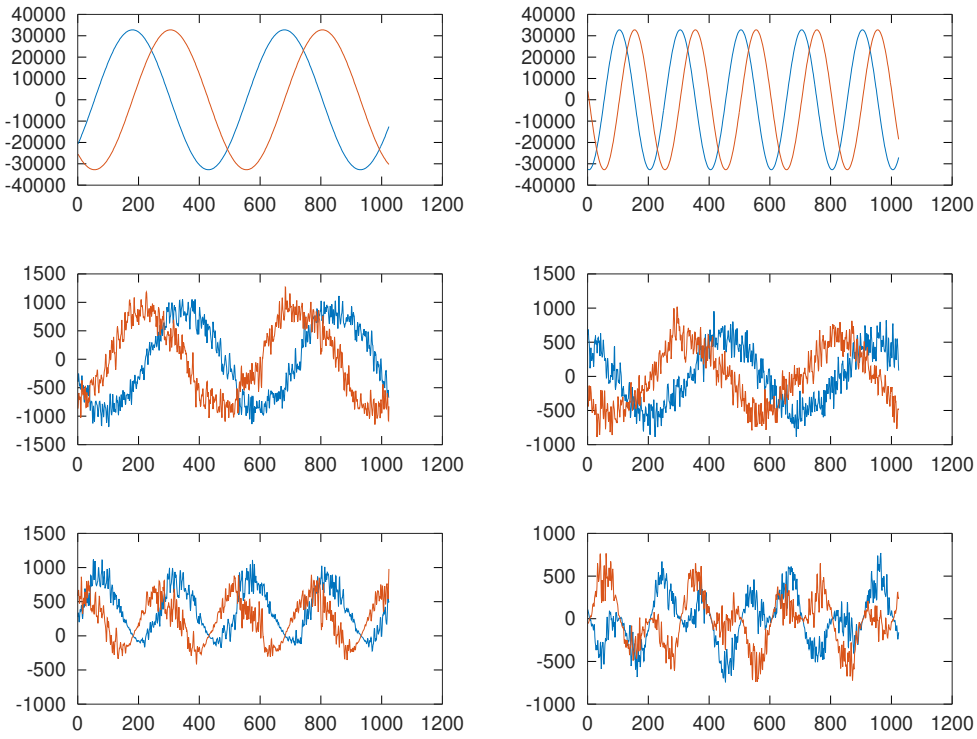


Figure 6: Measurements results with a 100 kHz local oscillator (left) and with (right) a 250 kHz local oscillator, while in both cases the output is offset by 100 kHz from the input. Top to bottom: NCO, measured I/Q streams, and mixer output.

The original PlutoSDR used with GNU Radio remains functional despite the updated bitstream. The same result than the one demonstrated from the OscimpDigital framework but running on the host computer is shown in Fig. 7.

5 TCL update of the original ADI PL configuration

Rather than bothering with the graphical user interface, adding new functionalitites brought by the OscimpDigital framework to the original ADI PL configuration file is most efficiently achieved by updating the TCL script.

A default design for PlutoSDR, with OscimpDigital support, is available in the tutorial subdirectory `project2/design` or alternatively in `$OSCIMP_DIGITAL_APP/plutosdr/plutosdr_template`.

The TCL script must be updated with the additional blocks:

```
# axiStreamToComplex
ad_ip_instance axiStreamToComplex axis2Complex
ad_ip_parameter axis2Complex CONFIG.DATA_SIZE 16

ad_connect axi_ad9361/rst axis2Complex/s00_axis_reset
```

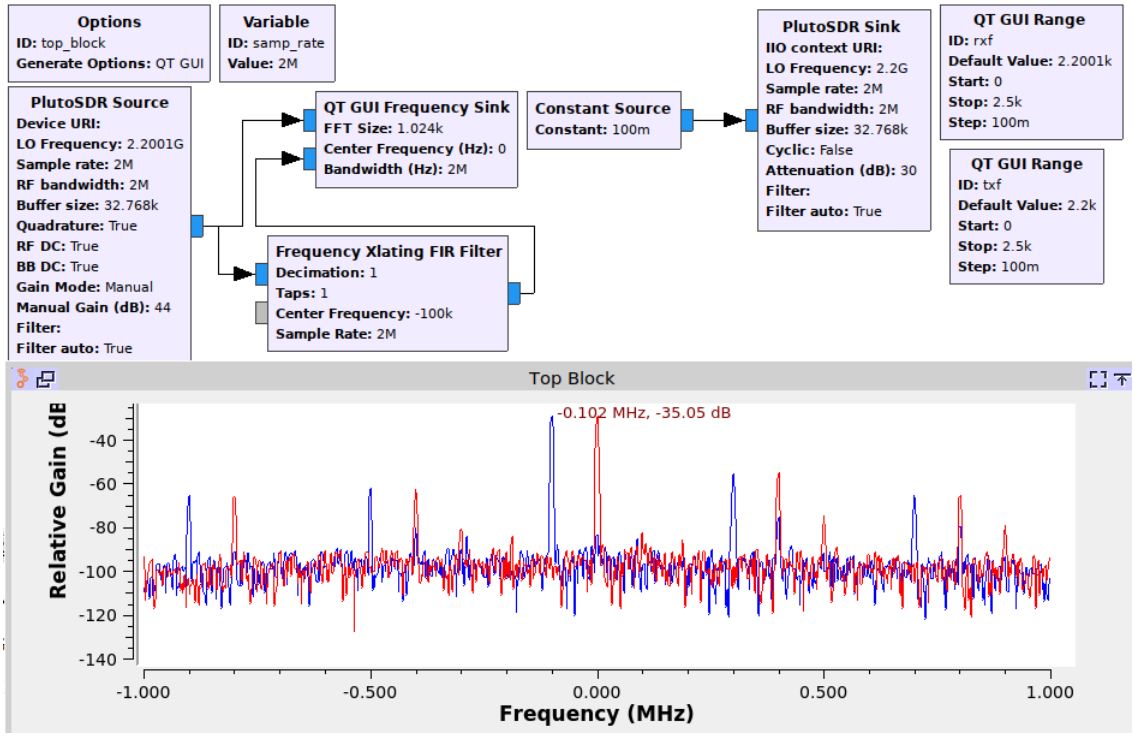


Figure 7: GNU Radio flowchart accessing the PlutoSDR as source and sink with the updated bitstream configuring the PL.

```
ad_connect axi_ad9361/l_clk axis2Complex/s00_axis_aclk

ad_connect fir_decimator/m_axis_data_tdata axis2Complex/s00_axis_tdata
ad_connect fir_decimator/m_axis_data_tvalid axis2Complex/s00_axis_tvalid

#nco
ad_ip_instance nco_counter nco
ad_ip_parameter nco CONFIG.DATA_SIZE 16
ad_ip_parameter nco CONFIG.COUNTER_SIZE 28
ad_ip_parameter nco CONFIG.LUT_SIZE 10

ad_connect axi_ad9361/rst nco/ref_rst_i
ad_connect axi_ad9361/l_clk nco/ref_clk_i

ad_connect sys_rstgen/peripheral_reset nco/s00_axi_reset
ad_cpu_interconnect 0x43C00000 nco

# duppl raw data -> mixer and dataComplex_to_ram
ad_ip_instance dupplComplex_1_to_2 duppl_0
ad_ip_parameter duppl_0 CONFIG.DATA_SIZE 16

ad_connect axis2Complex/data_out duppl_0/data_in

# duppl NCO -> mixer and dataComplex_to_ram
ad_ip_instance dupplComplex_1_to_2 duppl_1
ad_ip_parameter duppl_1 CONFIG.DATA_SIZE 16
```

```
ad_connect nco/sine_out duppl_1/data_in

# mixer
ad_ip_instance mixerComplex_sin mixer
ad_ip_parameter mixer CONFIG.NCO_SIZE 16
ad_ip_parameter mixer CONFIG.DATA_SIZE 16

ad_connect duppl_0/data1_out mixer/data_in
ad_connect duppl_1/data1_out mixer/nco_in

#dataComplex
ad_ip_instance dataComplex_to_ram data_to_ram
ad_ip_parameter data_to_ram CONFIG.NB_INPUT 3
ad_ip_parameter data_to_ram CONFIG.DATA_SIZE 16
ad_ip_parameter data_to_ram CONFIG.NB_SAMPLE 1024

ad_connect duppl_0/data2_out data_to_ram/data1_in
ad_connect duppl_1/data2_out data_to_ram/data2_in
ad_connect mixer/data_out data_to_ram/data3_in

ad_connect sys_rstgen/peripheral_reset data_to_ram/s00_axi_reset
ad_cpu_interconnect 0x43C10000 data_to_ram
```

TCL functions starting with *ad_* are from ADI's `hdl` repository:

- `ad_ip_instance` is used to add an instance of an IP. The first parameter is the name of the IP (not the *VLNV* (*Vendor Library Name Version*) and the second is the instance name;
- `ad_ip_parameter` configures, for one instance whose name is given as first argument, one parameter (second argument) with name *CONFIG.XXX* with the value passed as third argument;
- `ad_connect` to connect signals/interfaces between IPs with format *instance_name/interface_name*;
- `ad_cpu_interconnect` to connect the `AXI Lite` interface for an instance (second argument) to the Zynq PS, with base address provided as first parameter.

Note: ADI `ad_cpu_interconnect` automatically connects AXI clock and AXI reset to the CPU, if this was not done previously. However, this function always considers reset as active low. When the reset signal is active high, as for most of OscImpDigital IPs, this signal must be **explictly connected** before using this function.

The most difficult aspect with this approach is to know interface names and parameters: see `$OSCIMP_DIGITAL_IP/README.md` for a detailed description of each IP.

Following these updates, `make` will generates `pluto.xpr` and then synthesize the bistream which can be used as described earlier in the text to generate the `.frm` and `.dfu` images to reflash the PlutoSDR.