

Redpitaya: seventh example

G. Goavec-Mérou, J.-M Friedt

January 30, 2019

This tutorial departs from the previous series in its aim of being portable to any platform supported by the OscIMP Digital framework by not requiring dedicated hardware peripheral to generate a data source but will rely on a pseudo-random generator to characterize filter responses. By not requiring the ADC and DAC peripherals of the Redpitaya, this example aims at demonstrating how assembling the blocks provided by this framework is portable to various platforms including the ZC706 board or Altera-SoC based boards.

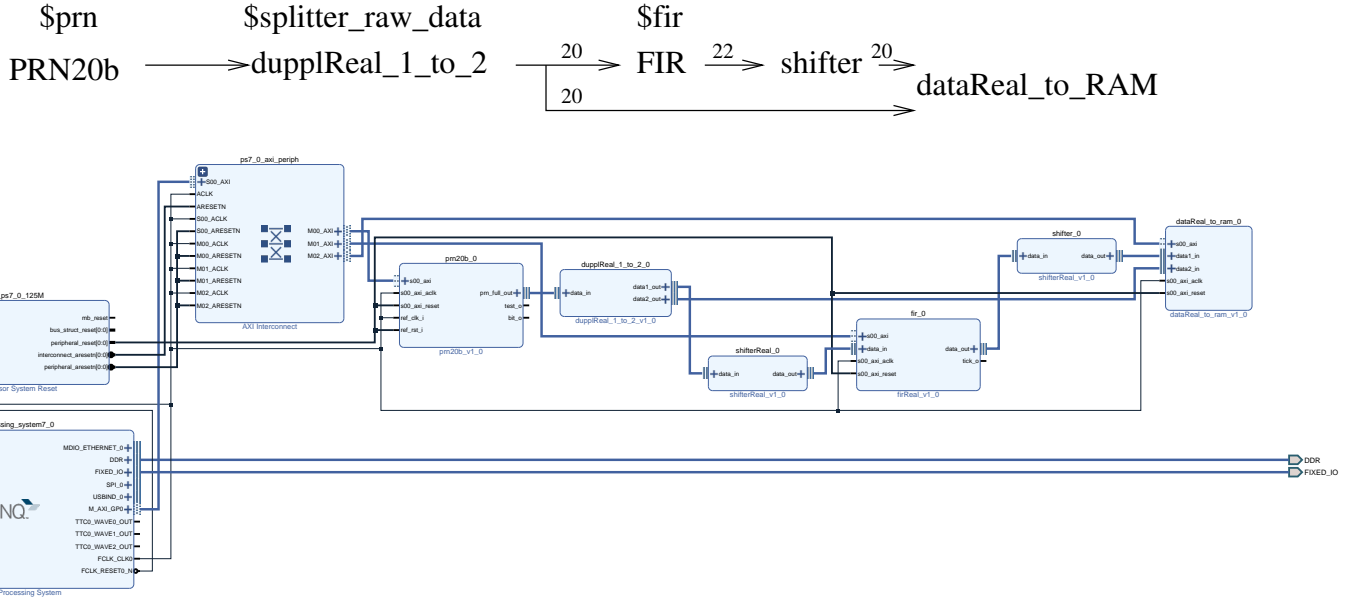


Figure 1: Schematic of the objective and final processing chain described in this document

1 PRN20 block

Characterizing a filter response requires probing with a known source the device under test and recording the response for each frequency. Assuming a linear, time invariant system, each frequency can either be swept sequentially (frequency sweep network analyzer) or probed simultaneously using a broadband noise generator and extracting each frequency contribution using a Fourier transform. The latter approach is selected here, with the broadband noise generator being implemented as a 20-bit long linear feedback shift register (LFSR). With a repetition period of one million samples, 1 Msamples can be collected or a duration of 8 ms if clocking at a rate of 125 MHz.

In the following example, the PRN output is fed to one channel of the Data to RAM block, and the other channel goes through the filter with additional shifting to keep a controlled number of bits before feeding the second channel of the Data to RAM block.

2 PS: Linux kernel driver

The same driver for communicating with the PL will be requested as (`data_to_ram`), in addition to which we wish to communicate with the FIR to configure taps. This time, the `module_generator` XML configuration file should look like

```
<?xml version="1.0" encoding="utf-8"?>
<drivers name="prn_fir" version="1.0">
  <driver name="data_to_ram" >
    <board_driver name="data00" id="0" base_addr="0x43C20000" addr_size="0xffff" />
  </driver>
  <driver name="fir" id="0" >
    <board_driver name="fir00" id="0" base_addr="0x43C10000" addr_size="0xffff" />
  </driver>
</drivers>
```

The name of the appropriate module has been found by looking for a directory with the same name than the IP in the `fpga_driver` directory.






Cell	Slave Interface	Base Name	Offset Address	Range	High Address
✓  processing_system7_0					
✓  Data (32 address bits : 0x40000000 [1G])					
 dataReal_to_ram_0	s00_axi	reg0	0x43C2_0000	64K ▾	0x43C2_FFFF
 fir_0	s00_axi	reg0	0x43C1_0000	64K ▾	0x43C1_FFFF
 prn20b_0	s00_axi	reg0	0x43C0_0000	64K ▾	0x43C0_FFFF

Figure 2: Address range used by the IPs as defined by Vivado.

In addition to displaying the graphical output of Vivado to find the address space used by each block, we provide in `design/addr.tcl` a Vivado TCL script for extracting such information from the synthesized bitstream. Such a script is most useful when remotely working on a server running Vivado without graphical display output. The output should look like

```
$ vivado -mode batch -source addr.tcl | grep -A1 \ SEG
NAME      string  false      SEG_dataReal_to_ram_0_reg0
OFFSET    string  false      0x43C20000
--
NAME      string  false      SEG_fir_0_reg0
OFFSET    string  false      0x43C10000
--
NAME      string  false      SEG_prn20b_0_reg0
OFFSET    string  false      0x43C00000
```

3 On the Redpitaya ...

The userspace software is limited to configuring the FIR and collecting the raw data streamed from the PRN generator as well as the filtered datastream.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "fir_conf.h" // library for communicating with the FIR

int main()
{int k;
 char c[16384];
 int fi,fo;
 fi=open("/dev/data00",O_RDWR);
 fo=open("/tmp/data.bin",O_WRONLY|O_CREAT,S_IRWXU);
 fir_send_conf("/dev/fir00", "coefs.txt", 32);

 for (k=1;k<5;k++)
 {read(fi,c,16384);
  write(fo,c,16384);
 }
 close(fi);
 close(fo);
 return EXIT_SUCCESS;
}
```

The dataset is processed using the following GNU/Octave script

```
f=fopen('data15fois16.bin');d=fread(f,inf,'int32');x=d(1:2:end);
z=zeros(1,length(x)); z(1:15)=1;
u=(abs(fft(d(1:2:end)))); plot(u/max(u),'b'); hold on
u=(abs(fft(z-mean(z)))); plot(u/max(u),'r');
```

illustrating how the data are interleaved in the file ($x=d(1:2:end);$) as well as the consistency of the filter shape with the expected transfer function, here computed for the very simple rectangular window.

The results of the FIR filter implementation are illustrated in Figs. 3 and 4). Fig. 3 demonstrates the impact of the FIR by displaying the power spectrum of the unfiltered data (blue, output of the PRN generator) and filtered data (red) in the case of a simple rectangular window filter. Fig. 4 demonstrates the consistency of the result with theory by overlaying the theoretical spectral response of various rectangular window filters (red) with the experimental results (blue).

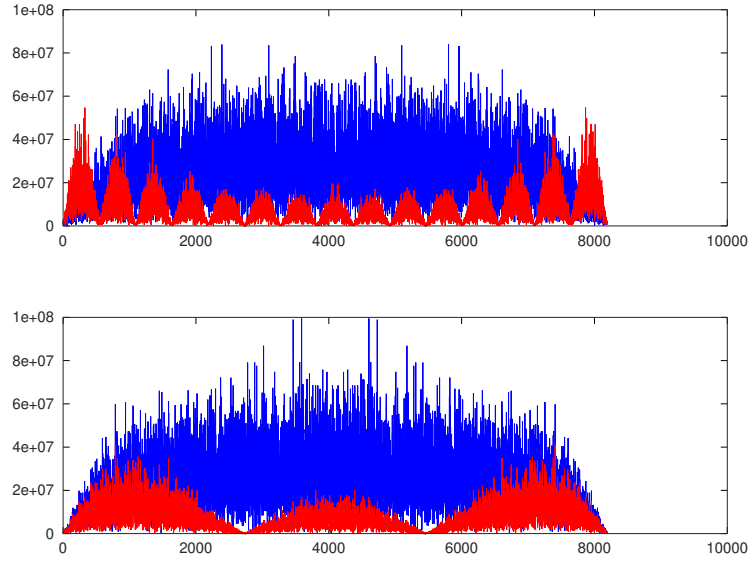


Figure 3: Comparison of the FIR input (pseudo random generator spectrum – blue) and the filtered output (red), here in the case of simple rectangular window filters.

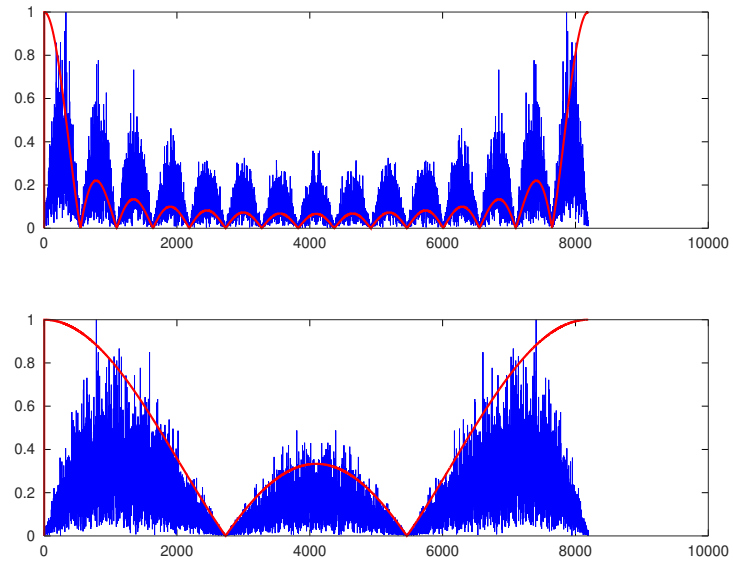


Figure 4: Comparison of the FIR filtered output (blue) and the expected filter response (red), here in the case of simple rectangular window filters.