

# Base designs

S. Denis, B. Maréchal G. Goavec-Mérou, J.-M Friedt

October 25, 2019

This document aims at making a breve description of basic RF functions that can be set-up, using Vivado and the fpga\_ip repository. It assumes acquired the a priori knowledge on the OscillatorIMP ecosystem, otherwise refer to : <https://github.com/oscimp/oscimpDigital>. The points discussed, listed below, are wrapped up in the example of a control loop design, with modulation and demodulation. A summary table of the IP blocks that can be used to build a numerical RF setup is given in the next pages.

- |                                |  |
|--------------------------------|--|
| 1. Reminder on signal dynamics | 7. Frequency and phase modulation of a NCO |
| 2. Webserver                   | 8. Demodulation                            |
| 3. Double voltage source       | 9. Filtering                               |
| 4. Double DDS                  | 10. Monitoring                             |
| 5. Amplitude modulation V1     | 11. Example to a control loop              |
| 6. Amplitude modulation V2     | 12. FAQ                                    |

## 1 Reminder on signal dynamics

Regardless of the presented functions, it's obviously better to optimize the dynamic of a signal to the range of data available. This first minimizes the part of noise of the electronics with respect to the signal. Secondly if the signal dynamic exceeds the range available, there is an overflow. For instance 14 *bits* signed data represent a range of  $\pm 13$  *bits* ie. from  $-8192$  to  $8191$  samples. Above  $8191$  samples, there is an overflow and the signal returns to the beginning of the range, ie.  $-8192$ . Representation in Fig.1.

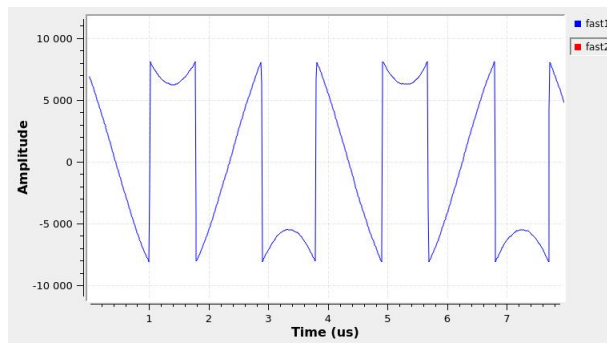
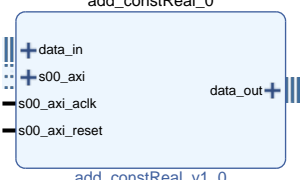
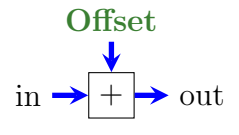
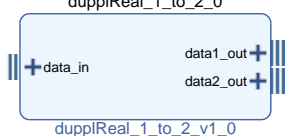
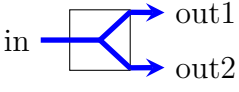
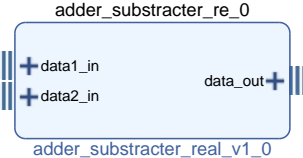
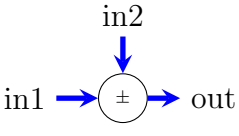
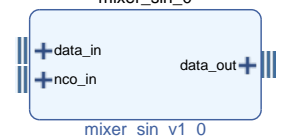
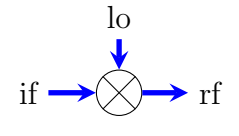
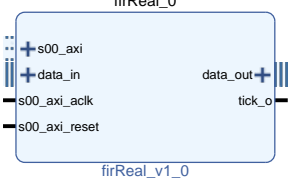
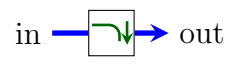
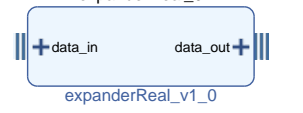
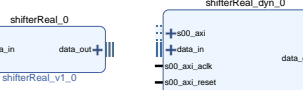
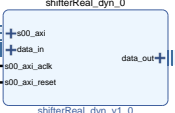
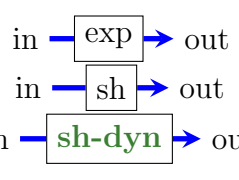
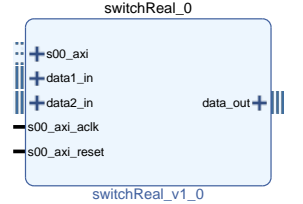
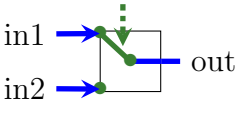
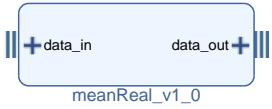
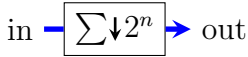
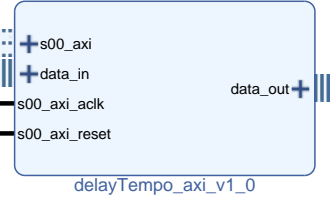

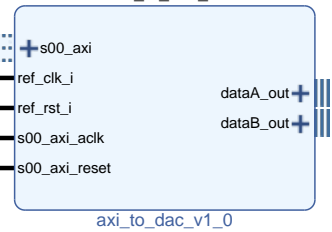
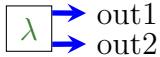
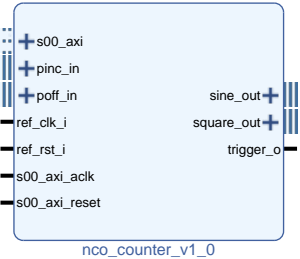
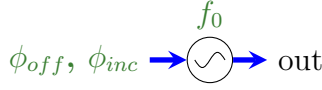


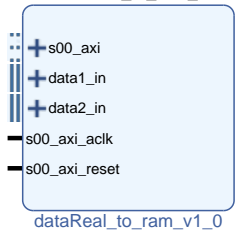



Figure 1: Overflow on the top and bottom of a sine.

IP	Equivalent RF function or numeric function	Equivalent scheme with tuneable entries
 <p>add_constReal_0</p> <p>add_constReal_v1_0</p>	Tuneable amplitude offset, bias.	
 <p>duplReal_1_to_2_0</p> <p>duplReal_1_to_2_v1_0</p>	Splitter	
 <p>adder_substracter_re_0</p> <p>adder_substracter_real_v1_0</p>	Combiner. Add or subtract signals.	
 <p>mixer_sin_0</p> <p>mixer_sin_v1_0</p>	Mixer	
 <p>firReal_0</p> <p>firReal_v1_0</p>	Tuneable filter. FIR with decimation option.	
 <p>expanderReal_0</p> <p>expanderReal_v1_0</p>  <p>shifterReal_0</p> <p>shifterReal_v1_0</p>  <p>shifterReal_dyn_0</p> <p>shifterReal_dyn_v1_0</p>	Can be assimilated to $2^m$ amplifiers or attenuators. Are used to adapt the data size between blocks, or to select the range of the numeric signal. Expander: crop end of world, expand beginning of word. Shift: crop beginning of world, expand end of word.	
 <p>switchReal_0</p> <p>switchReal_v1_0</p>	Switch	

IP	Equivalent RF function or numeric function	Equivalent scheme with tuneable entries
	Moving average. Decimation of $2^n$ with averaging: slows the data flow.	
	Tuneable delay line ie. cables.	
	Tuneable voltage source. Controllable states/constants.	
	DDS NCO	
	PID	
	Monitoring: oscilloscope, spectrum analyzer... Can also be used to process the signal in the CPU. Up to 12 channels.	

## 2 Webserver

The tuneable parameters of the IPs are controlled through C coded functions, visible in the /oscimpDigital/lib/my\_lib.h library files. Example of functions with a generic driver "fpgagen":

```
fpgagen_send_conf(char *filename, int reg, int value);  
fpgagen_rcv_conf(char *filename, int reg, int *value);
```

Those functions can be implemented in a graphic interface to constitute a user friendly control of the IPs. Here we show an example of webserver using RemI<sup>1</sup>, a cross platform remote gui for python. The wrapper liboscimp\_fpga.py makes the intermediary between the webserver and the libraries. It takes the following form:

```
import ctypes  
from ctypes import *  
lib = ctypes.CDLL('/usr/lib/liboscimp_fpga.so')  
  
def fpgagen_send_conf(filename, reg, value):  
    file = ctypes.create_string_buffer(str.encode(filename))  
    my_val = int(value)  
    lib.add_const_set_offset(file, reg, my_val)  
  
def fpgagen_rcv_conf(filename, reg):  
    file = ctypes.create_string_buffer(str.encode(filename))  
    my_val = c_int()  
    ret_val = lib.fpgagen_rcv_conf(file, reg, byref(my_val))  
    return (ret_val, my_val.value)
```

Then the example of functions implemented in the webserver to configure the IPs is:

```
import liboscimp_fpga  
liboscimp_fpga.fpgagen_send_conf("/dev/my_file", my_reg, my_value)
```

In the webserver, values sent to the IPs can either take the form of slider, a spinbox, a checkbox a button... In our case, a simple actuator will be represented by a checkbox, and any other controllable value by both a slider and a spinbox. The structure of the webserver ins as fallows:

```
#!/usr/bin/env python  
  
import liboscimp_fpga  
import ctypes  
import remi.gui as gui  
from remi import start, App  
  
class MyApp(App):
```

---

<sup>1</sup>Download and Faq : <https://www.remigui.com/>

```

def __init__(self, *args):
    super(MyApp, self).__init__(*args)

def main(self):
    self.w = gui.VBox()

    #Create the slider and the spinbox, whose value is restricted to -8192 to 8191 (no overflow)
    self.hbox_MY_VALUE = gui.HBox(margin="10px")
    self.lb_MY_VALUE = gui.Label("/dev/MY_VALUE_FILE", width="20%", margin="50px")
    self.sd_MY_VALUE = gui.Slider(0, -8192, 8191, 1, width="60%", margin="10px")
    self.sd_MY_VALUE.set_oninput_listener(self.sd_MY_VALUE.changed)
    self.sb_MY_VALUE = gui.SpinBox(0, -8192, 8191, 1, width="20%", margin="10px")
    self.sb_MY_VALUE.set_on_change_listener(self.sb_MY_VALUE.changed)
    self.sd_MY_VALUE.changed(self.sd_MY_VALUE, self.sd_MY_VALUE.get_value())
    self.hbox_MY_VALUE.append(self.lb_MY_VALUE)
    self.hbox_MY_VALUE.append(self.sd_MY_VALUE)
    self.hbox_MY_VALUE.append(self.sb_MY_VALUE)
    self.w.append(self.hbox_MY_VALUE)

    #Create the checkbox
    self.hbox_MY_ACTUATOR = gui.HBox(margin="10px")
    self.lb_MY_ACTUATOR = gui.Label("/dev/MY_ACTUATOR_FILE", width="20%", margin="50→
        ↪px")
    self.cb_MY_ACTUATOR = gui.CheckBox(True, width="5%", margin="10px")
    self.cb_MY_ACTUATOR.set_on_change_listener(self.cb_MY_ACTUATOR.changed)
    self.hbox_MY_ACTUATOR.append(self.lb_MY_ACTUATOR)
    self.hbox_MY_ACTUATOR.append(self.cb_MY_ACTUATOR)
    self.w.append(self.hbox_MY_ACTUATOR)

    return self.w

#Function called by the slider
def sd_MY_VALUE_changed(self, widget, value):
    print("/dev/MY_VALUE_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_VALUE_FILE", MY_REG, int(value))
    self.sb_MY_VALUE.set_value(int(value))

#Function called by the spinbox
def sb_MY_VALUE_changed(self, widget, value):
    print("/dev/MY_VALUE_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_VALUE_FILE", MY_REG, int(value))
    self.sd_MY_VALUE.set_value(int(float(value)))

#Function called by the checkbox
def sb_MY_ACTUATOR_changed(self, widget, value):
    print("/dev/MY_ACTUATOR_FILE", MY_REG, int(value))
    liboscimp_fpga.fpgagen_send_conf("/dev/MY_ACTUATOR_FILE", MY_REG2, int(value))
    self.sd_adc1_offset.set_value(int(float(value)))

#Launch oh the webserver on the local machine
start(MyApp, address="0.0.0.0", port=80, title="My_super_webserver")

```

Preview of the webserver created:

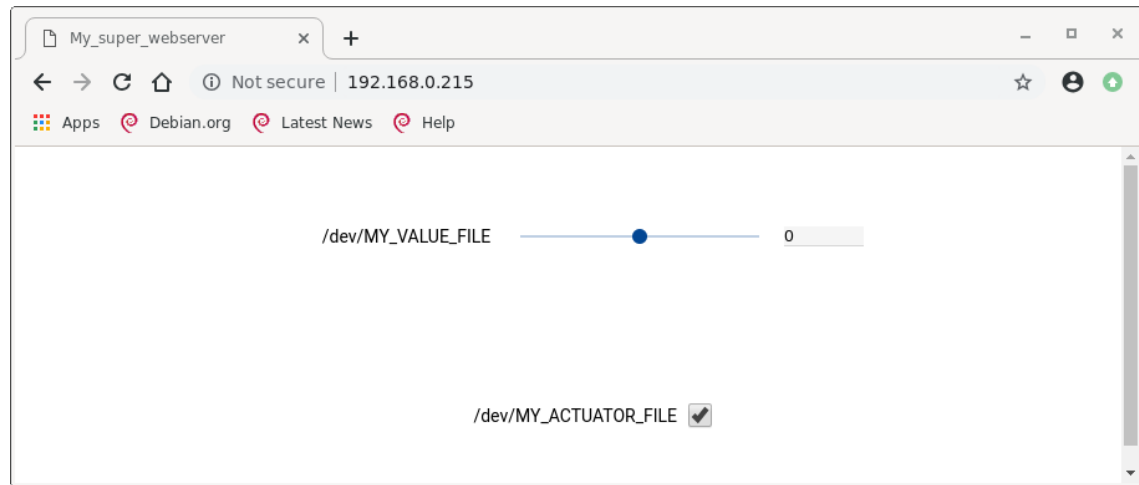


Figure 2: Example of webserver with MY\_VALUE\_FILE represented by a slider and a spinbox, and MY\_ACTUATOR\_FILE represented by a checkbox.

Here the generic driver fpgagen is used as an example, however the use of a different driver can lead to various requirements: arguments, data type, or several functions. Some specific cases will be treated in the next sections. In the other cases, refer to the /oscimpDigital/lib/my.lib.h library files, or the oscimpDigital documentation: <https://github.com/oscimp/oscimpDigital/tree/master/doc>

### 3 Double voltage source

A double tuneable voltage source can be set up si using the axi\_to\_dac IP. However it's only one of the many functions that can be imagined with this IP.

See [https://github.com/oscimp/oscimpDigital/blob/master/doc/IP/axi\\_to\\_dac.md](https://github.com/oscimp/oscimpDigital/blob/master/doc/IP/axi_to_dac.md). The block diagram is as follows :

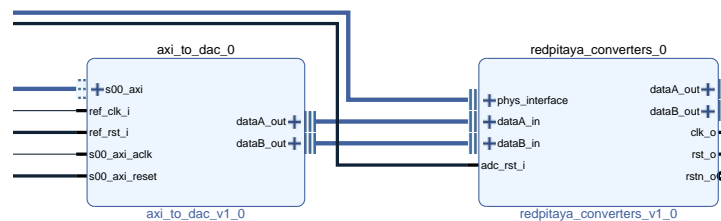
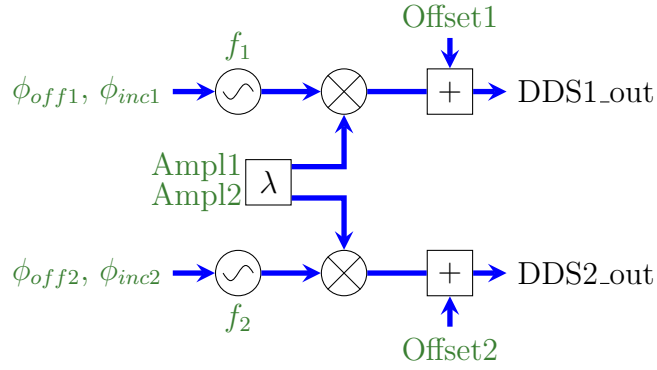


Figure 3: Part of the block diagram for the tuneable voltage source.

In this configuration and with data on 14 *bits*, all the parameters of the `axi_to_dac` block keep their default value. The tuning of the output is performed with the webserver (see section 2). With the Redpitaya board, the maximum voltage is  $\pm 1$  V per output.

## 4 Double DDS

The schematic configuration of the double DDS we propose here is shown below:



It corresponds to two DDS with adjustable frequency, amplitude, output offset, and referenced on the same clock. Frequency/phase and amplitude modulation are not represented here, but can be added using sections ?? and ??. The block diagram associated with this scheme is as follows:

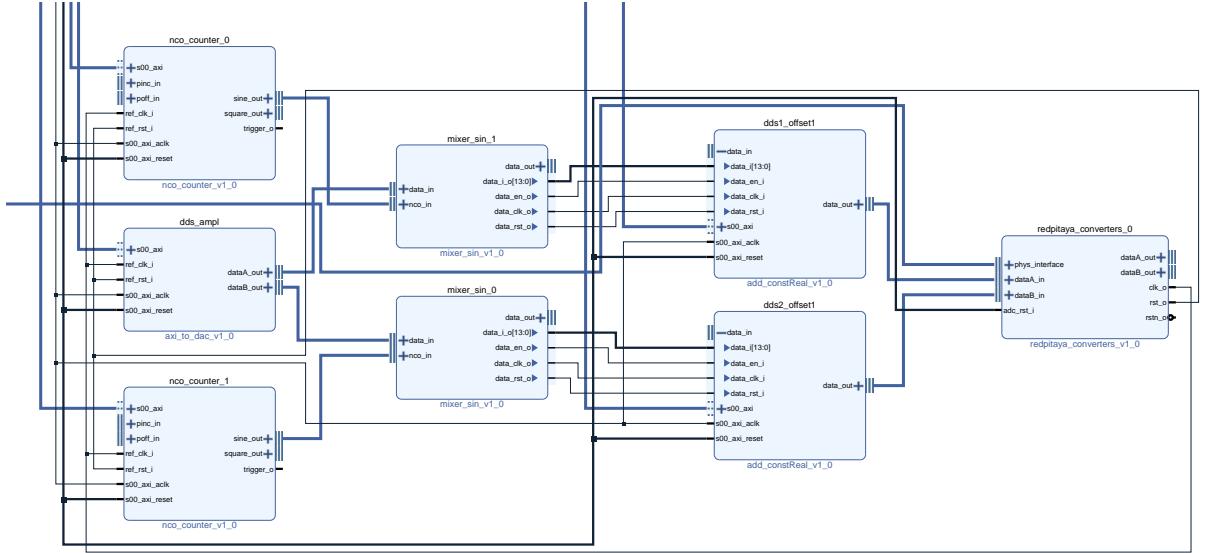


Figure 4: Part of the block diagram for the double DDS.

## 4.1 IP configuration

The IPs configuration may change depending on the board/application, however in this example we used the following configurations:

IP	Configuration
nco_counter	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
axi_to_dac	Data size: 14 <i>bits</i>
mixer_sin	Data size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
add_constReal	Data in size: 14 <i>bits</i> Data out size: 14 <i>bits</i> Signed

## 4.2 Webserver configuration

For the DDS offsets and amplitudes blocks we only use one value to be controlled, ie. one slider/spinbox. However the NCO block offers several options as the phase offset and increment can be internal or external to the block. Therefore we keep a default construction for the NCO in the webserver, including all these possibilities:

- 1<sup>st</sup> slider+spinbox: the frequency control, up to the half clock frequency (Hz)
- 2<sup>nd</sup> slider+spinbox: the phase offset control
- pinc checkbox: internal or external phase increment
- poff checkbox: internal or external phase increment

In the present case there is no external connections for the frequency and phase increments, thus the pinc and poff checkbox will remain checked. A preview of the webserver for the double DDS design is given in fig.5.



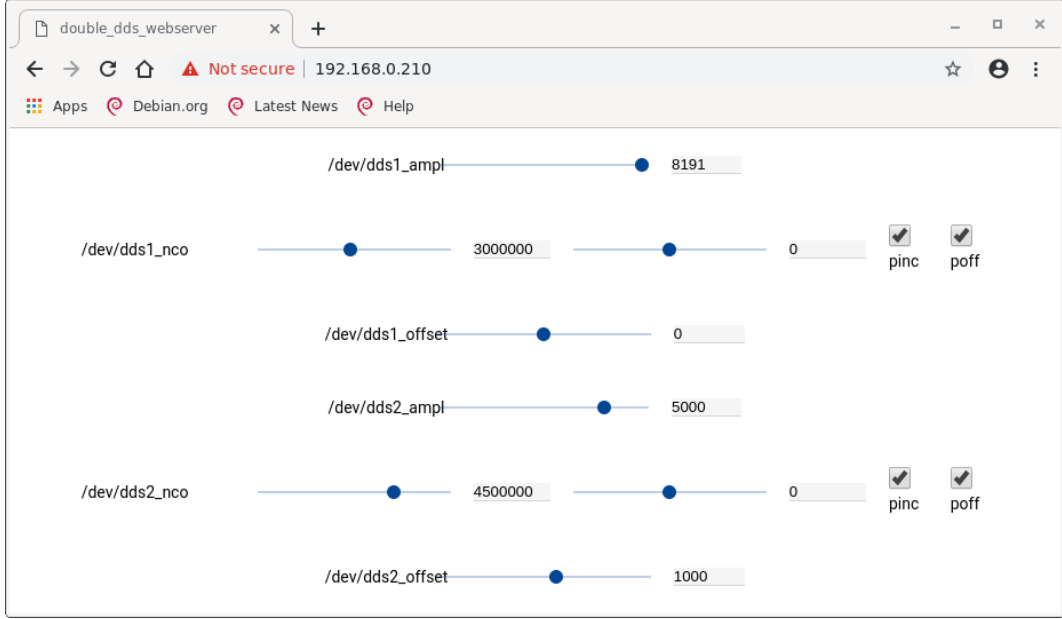


Figure 5: Screenshot of the double DDS webserver.

### 4.3 Expected output

We show in fig.6 an example of two signals generated.

Ch1:  $f_0 = 30 \text{ MHz}$ ,  $dds1\_ampl = 8191 \text{ samples}$ ,  $dds1\_offset = 0 \text{ sample}$ ,  $\phi_{off1} = 0 \text{ sample}$ .

Ch2:  $f_0 = 45 \text{ MHz}$ ,  $dds2\_ampl = 3000 \text{ samples}$ ,  $dds2\_offset = 5000 \text{ samples}$ ,  $\phi_{off2} = 0 \text{ sample}$ .

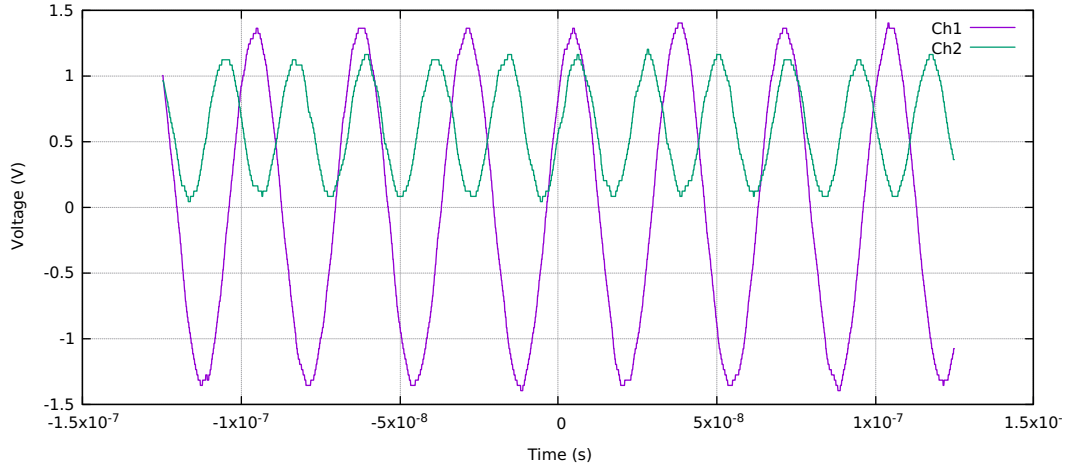


Figure 6: Expected output.

With an internal phase increment, the output signals are generated with an arbitrary

phase. This phase can be adjusted according to the intended application, using the phase offset slider, as represented in fig.7:

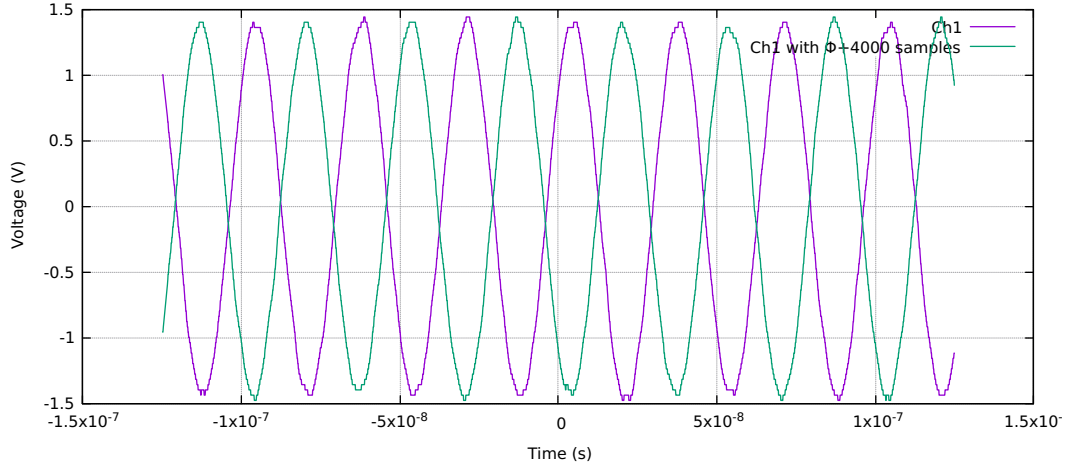


Figure 7: Phase offset.

#### 4.4 Unexpected output

In fig.8 the Ch1 signal is the same, but there is an overflow in Ch2 due to the sum of  $dds1\_ampl$  and  $dds1\_offset$ .

Ch2:  $f_0 = 45 \text{ MHz}$ ,  $dds2\_ampl = 8191 \text{ samples}$ ,  $dds2\_offset = 8191 \text{ samples}$ ,  $\phi_{off2} = 0 \text{ sample}$ .

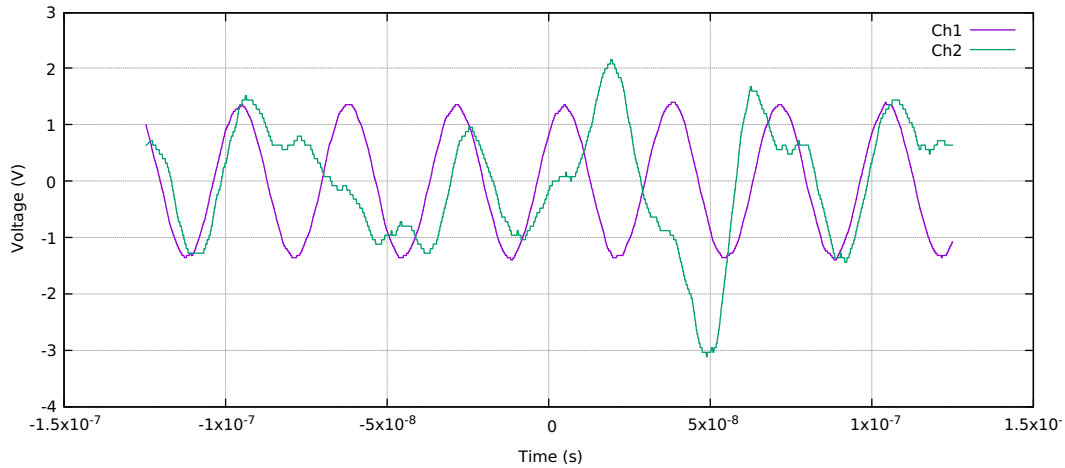
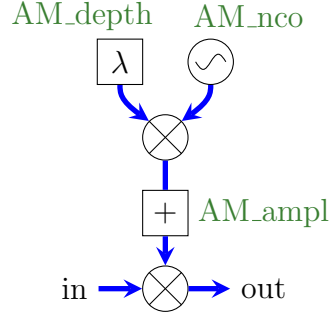


Figure 8: Unexpected output due to overflow in ch2.

Solution: decrease either  $dds1\_ampl$  or  $dds1\_offset$ , such as  $dds1\_ampl + dds1\_offset < 8191$ .

## 5 Amplitude modulation V1

An amplitude modulation can be performed easily, in the same way as with RF components:



This scheme is equivalent to the expression of the amplitude modulation:

$$y(t) = [1 + h \cos(\omega_m t)]z(t)$$

$$\Rightarrow out = [1 + \frac{AM\_depth}{AM\_ampl} AM\_nco] AM\_ampl \times in$$

Then the modulation depth is  $h = \frac{AM\_depth}{AM\_ampl}$ . Thereafter:

- $h = 0$  with  $AM\_depth = 0$
- $h = 0.5$  with  $AM\_depth = 4096$  samples and  $AM\_ampl = 8191$  samples
- $h = 1$  with  $AM\_depth = 8191$  samples and  $AM\_ampl = 8191$  samples
- $h = 2$  with  $AM\_depth = 8191$  samples and  $AM\_ampl = 4096$  samples

The block diagram corresponding to this scheme is as follows:

In this block diagram the input and output are connected to the converters block to make an example with external signal. However this principle can be included to other applications, such as the double dds design to add an amplitude modulation option to the generated signals.

### 5.1 IP configuration

IPs configuration in this example:

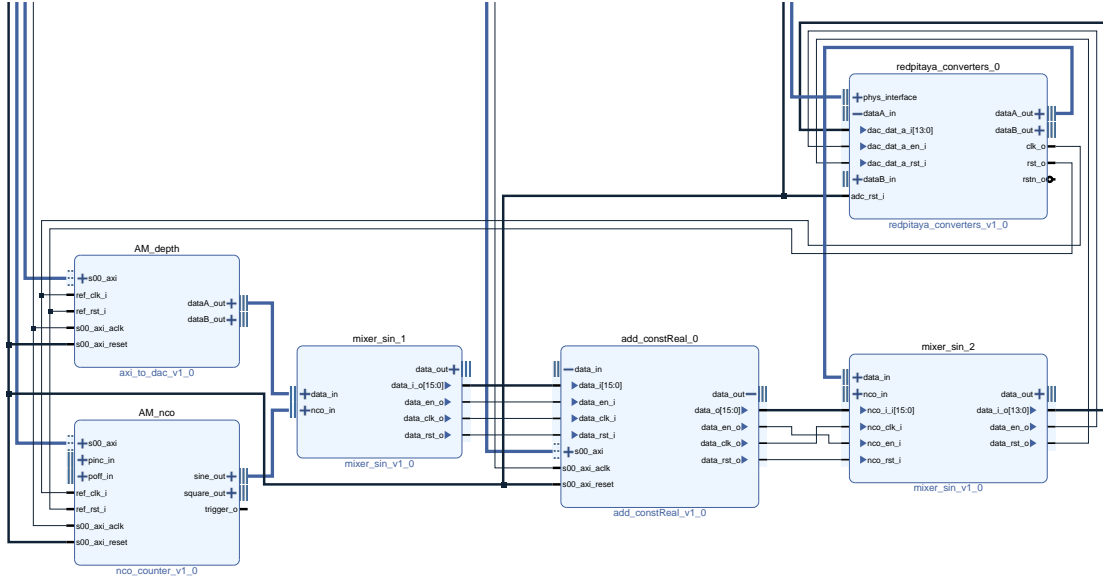


Figure 9: Part of the block diagram for this amplitude modulation.

IP	Configuration
nco_counter	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
axi_to_dac	Data size: 16 <i>bits</i>
mixer_sin_1	Data size: 16 <i>bits</i> Nco_size: 16 <i>bits</i>
mixer_sin_2	Data size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
add_const_real	Data in size: 16 <i>bits</i> Data out size: 14 <i>bits</i> Format: Signed

## 5.2 Webserver configuration

Here the webserver configuration is similar to the double dds one. In case refer to subsection 4.2. preview of the amplitude modulation part of the webserver, with the `mod_nco` controlling the modulation frequency, and `mod_ampl` controlling it's amplitude:

## 5.3 Expected output

To make a small preview of the expected behavior of the amplitude modulation, first we set the `AM_ampl` to 8191 *samples* we use at the input a sine signal of 5 *MHz* and 0 *dBm*. With an amplitude modulation of 50 *MHz* and 1000 *samples*, we expect:

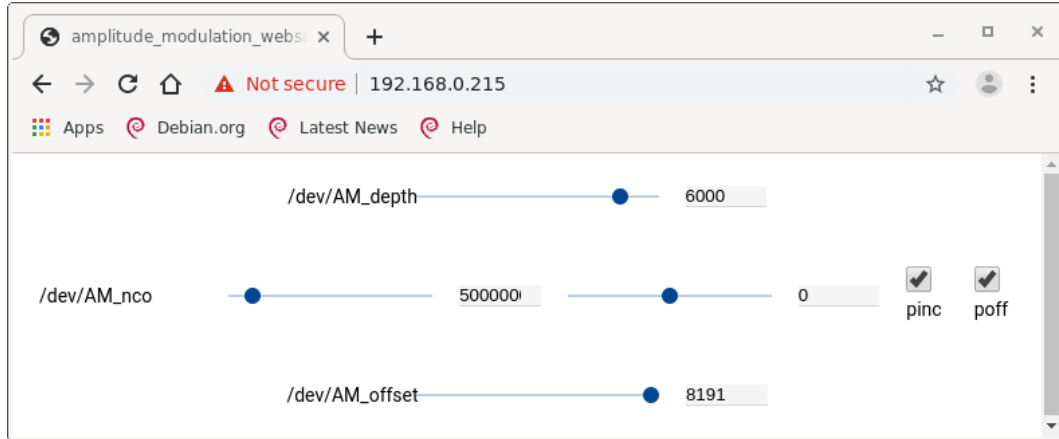
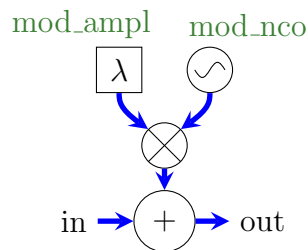


Figure 10: Screenshot of the amplitude modulation part of the webserver.

## 5.4 Unexpected output

# 6 Amplitude modulation V2

Another way to modulate the amplitude is presented here. Although it does not correspond to the known definition of an amplitude modulation, it can be useful to perform a scan acting on the amplitude of a given signal. An example is performing a frequency scan with a laser diode, by acting on its current control. The scheme used is quite similar:



The block diagram corresponding to this scheme is as follows:

## 6.1 IP configuration

IPs configuration in this example:

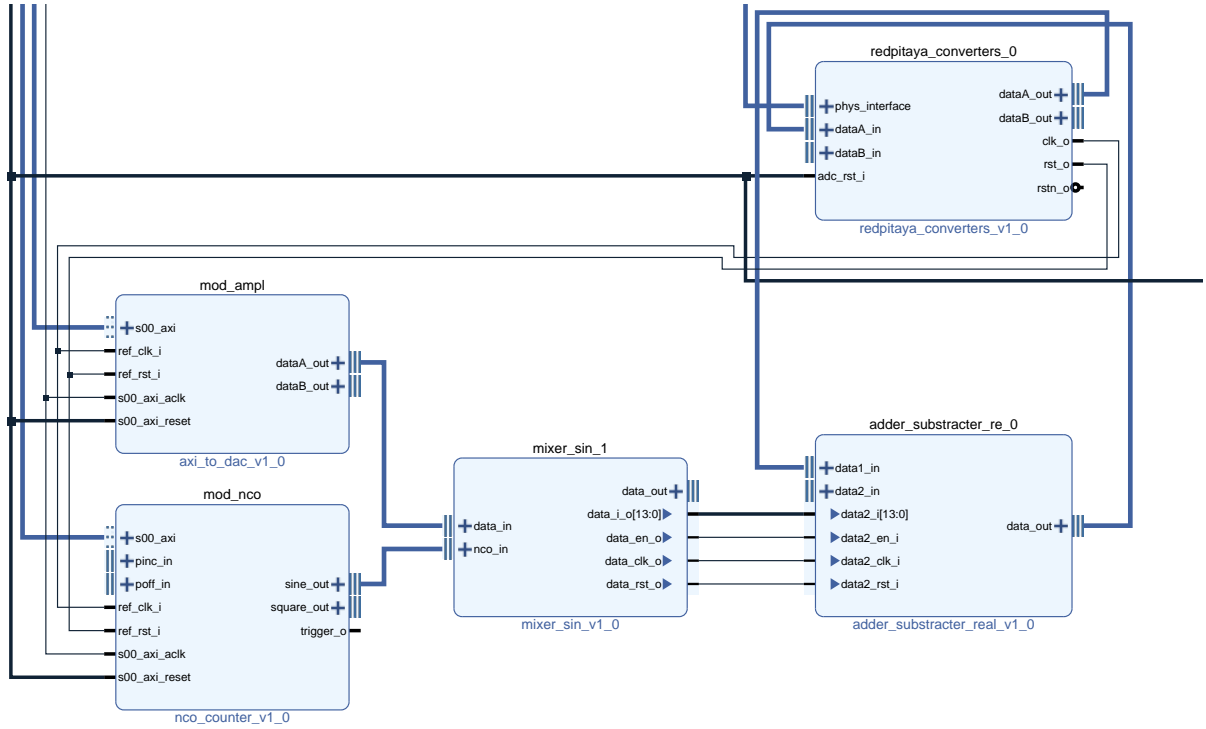


Figure 11: Part of the block diagram for the amplitude modulation.

IP	Configuration
nco_counter	Counter size: 40 <i>bits</i> Data size: 16 <i>bits</i> Lut size: 12 <i>bits</i>
axi_to_dac	Data size: 14 <i>bits</i>
mixer_sin	Data size: 14 <i>bits</i> Nco_size: 16 <i>bits</i>
adder_substracter_real	Data size: 14 <i>bits</i> Operation: add Format: Signed

## 6.2 Expected output

We use at the input a sine signal of 5 *MHz* and 0 *dBm*. With an amplitude modulation of 50 *MHz* and 1000 *samples*, we expect:

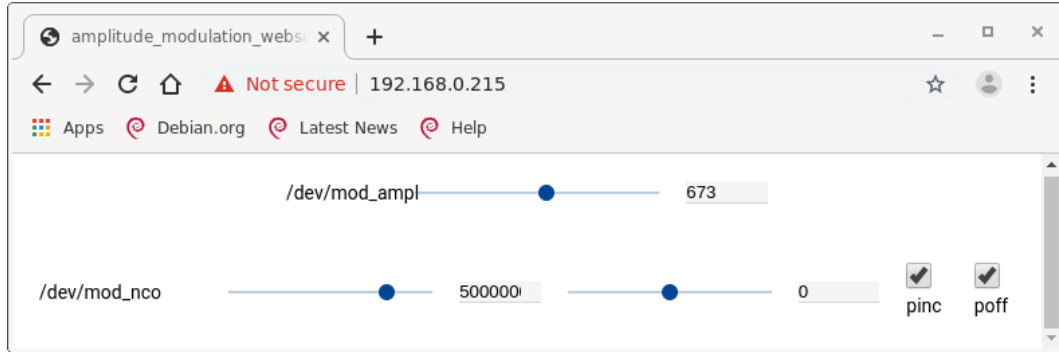


Figure 12: Screenshot of the amplitude modulation part of the webserver.

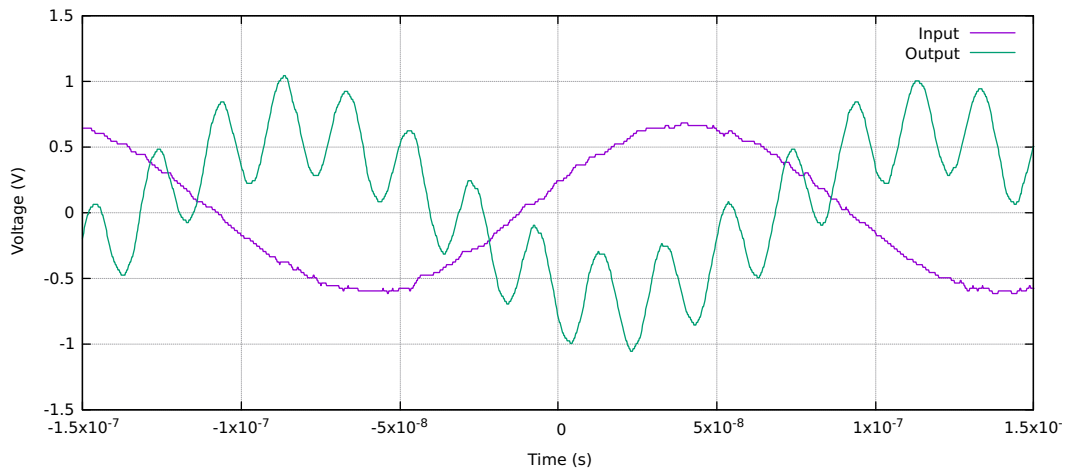


Figure 13: Expected behavior with input sine at 5  $MHz$  and amplitude modulation of 50  $MHz$ .

## 6.4 Unexpected output

An unexpected situation can be the output signal visible in figure 14:

This situation is very similar to the fig.8 in subsection 4.4: this output is a result of an overflow happening during the signal processing. This situation is due to an input signal too powerful with respect to the modulation depth. Solution: decrease the modulation depth or the power of the input signal.

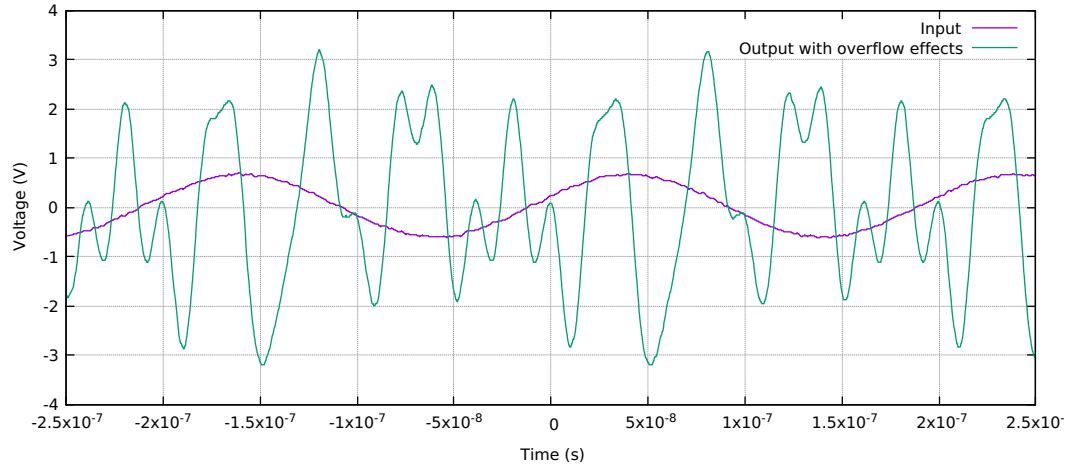
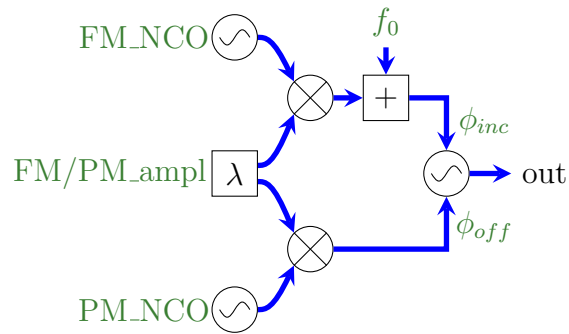


Figure 14: Unexpected behavior with input sine at 5  $MHz$  and amplitude modulation of 50  $MHz$ .

## 7 Frequency and phase modulation of a NCO





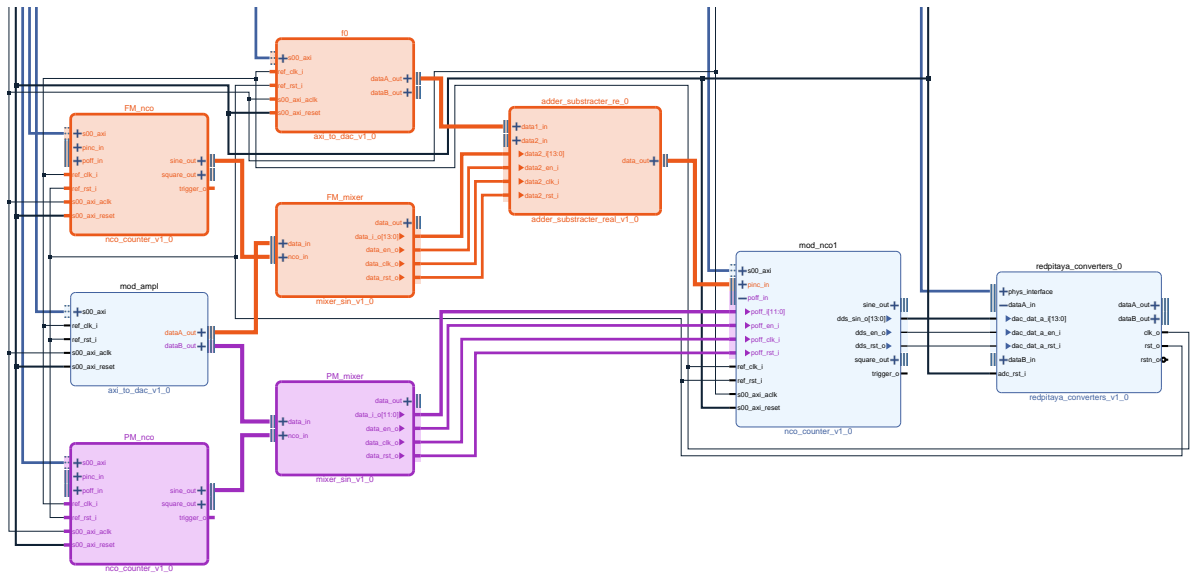


Figure 15: Part of the block diagram for frequency (orange) and phase (purple) modulation of a NCO.