RadioGaGa

Radio Gammas Gerbe Atmosphériques

Thomas BERTHET

Table of contents

1. Welcome to RadioGaGa documentation

This documentation uses mkdocs. To avoid to install all dependencies needed, a dockerfile is set in the doc directory. You need to have docker installed on your machine.
You can download it here

For ubuntu :

```
sudo apt-get update

sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

echo \
"deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

1.1 Commands

- `./build_docker.sh` - Build the docker image
- `./changelog.sh` - Update docs/changelog.md
- `./doc-build` - Build the doc and generate the PDF.
- `./doc-serve` - Start the live-reloading docs server

2. Changelog

r9801 | berthet | 2021-06-14 17:26:02 +0200 (lun. 14 juin 2021) | 1 ligne

New doc, generated with mkdoc.

3. Firmware

3.1 Signal Traitment

3.1.1 IPD Module

Introduction

This page will confirm the fonctionnement of the IPD module. We will see what it expexted in simulation. And confirm the fonctionnement with signaltap in the FPGA

Presentation

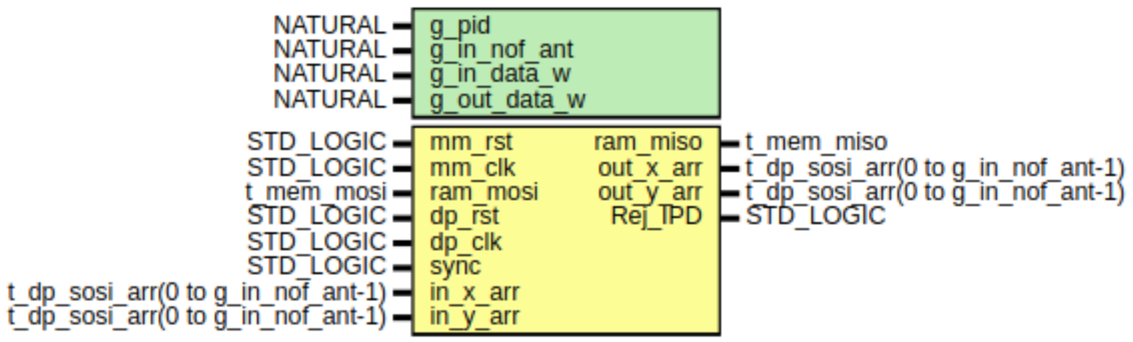


Figure 1 : IPD entity

The figure 1 shows ports of the module. The input data bus is directly connected to the output of ADC. Input data bus is size of 8x14 bits and same in output. This module is a delay line, it allows to to correct time travel of signal above the instrument. To do this the module allows to set a delay from 0 to 255 sample on each channel. Those delay are configurable in the register interface The module has the address offset of `16#1340000#`.
16#1340000# = c_processing_address_offset + c_radiogaga_address_offset + c_IPD_address_offset

Name	Address	Read	Write	Width	Comment
c_ipd_delay_x_offset	16#00#	x	x	4	Add delay to antennas on X polar
c_ipd_delay_y_offset	16#10#	x	x	4	Add delay to antennas on X polar

Simulation

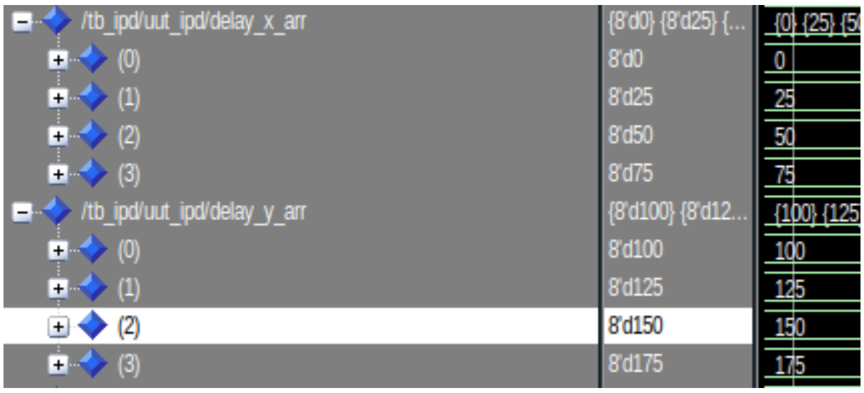
SHIFT TEST

IPD has 2 writable registers, *delay_x* and *delay_y*. So we can apply delay to both polarisations of each antenna. Each delay can be set on the range [0:255]. In simulation, I set delay incrementally by 25, like this we can see a shift of 25 sample on each channels. For this I wrote in the following registers :

# X polar
16#1340000# = 0
16#1340001# = 25

16#1340002# = 50
16#1340003# = 75

# Y polar
16#1340010# = 100
16#1340011# = 125
16#1340012# = 150
16#1340013# = 175



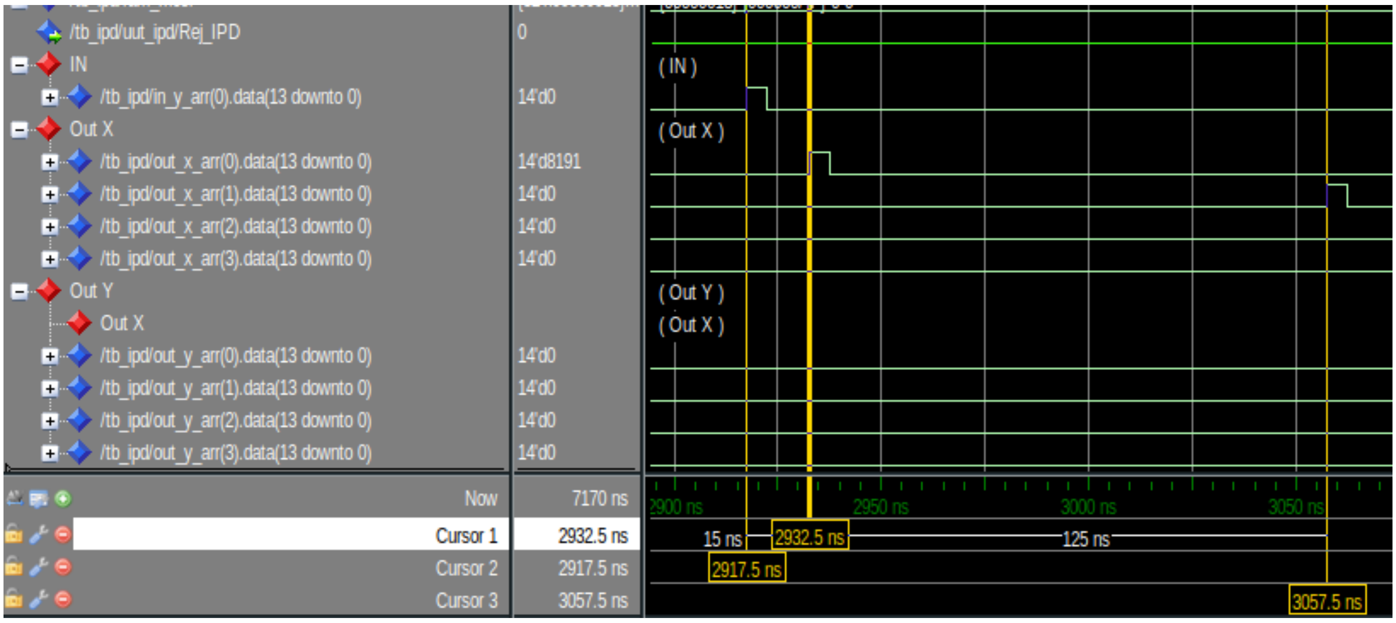
```
/tb_ipd/uut_ipd/delay_x_arr   {8'd0} {8'd25} {...   {0} {25} {5
  (0)                         8'd0                 0
  (1)                         8'd25                25
  (2)                         8'd50                50
  (3)                         8'd75                75
/tb_ipd/uut_ipd/delay_y_arr   {8'd100} {8'd12...   {100} {125
  (0)                         8'd100               100
  (1)                         8'd125               125
  (2)                         8'd150               150
  (3)                         8'd175               175
```

Figure 2 : Delays configuration

For simulation I sent the same pulse on each channels as we can see on `in_y_arr(0)`.
First of all, we can see a shift between `in_y_arr(0)` and `out_x_arr(0)`. The delay of it was set to *0* but there is a shift of *15 ns.* This is due by signal latching in VHDL. This delay is equivalent as 3 clock shots. This delay is not really important because it is applied on each `OUT` channel, so if we mesure shift between them, it will be correct.

Now we can see on the figure 3 a shift of 125 ns between the `out_x_arr(0)` and `out_x_arr(1)`.

$25 = 125/5ns$. Our shift is correct.



```
/tb_ipd/uut_ipd/Rej_IPD                    0
IN                                                     ( IN )
  /tb_ipd/in_y_arr(0).data(13 downto 0)    14'd0
Out X                                                  ( Out X )
  /tb_ipd/out_x_arr(0).data(13 downto 0)   14'd8191
  /tb_ipd/out_x_arr(1).data(13 downto 0)   14'd0
  /tb_ipd/out_x_arr(2).data(13 downto 0)   14'd0
  /tb_ipd/out_x_arr(3).data(13 downto 0)   14'd0
Out Y                                                  ( Out Y )
  Out X                                                ( Out X )
  /tb_ipd/out_y_arr(0).data(13 downto 0)   14'd0
  /tb_ipd/out_y_arr(1).data(13 downto 0)   14'd0
  /tb_ipd/out_y_arr(2).data(13 downto 0)   14'd0
  /tb_ipd/out_y_arr(3).data(13 downto 0)   14'd0

Now        7170 ns     2900 ns   2950 ns   3000 ns   3050 ns
Cursor 1   2932.5 ns   15 ns  2932.5 ns   125 ns
Cursor 2   2917.5 ns   2917.5 ns
Cursor 3   3057.5 ns                                 3057.5 ns
```

Figure 3 : Delays measurements

An other example on the figure 4 between the first channel `in_x_arr(0)` with delay set at *0* and the last sample `in_y_arr(3)` with delay set at $175 = 875/5ns$

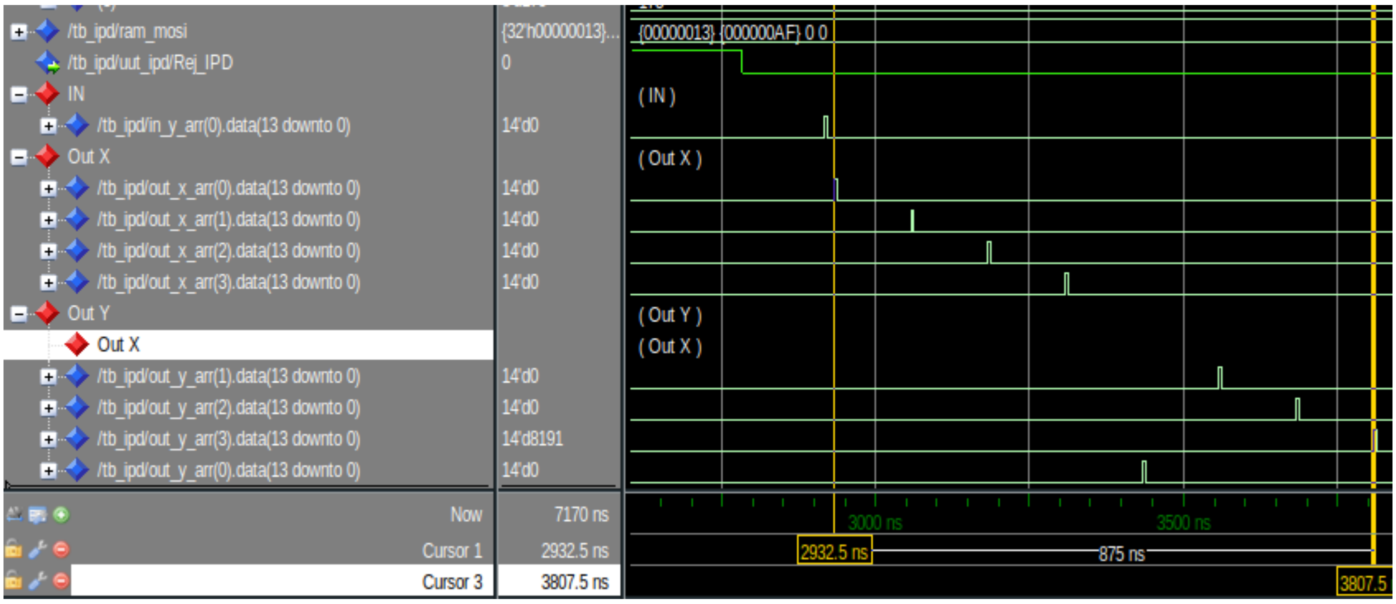

/tb_ipd/ram_mosi
/tb_ipd/uut_ipd/Rej_IPD
IN
/tb_ipd/in_y_arr(0).data(13 downto 0)
Out X
/tb_ipd/out_x_arr(0).data(13 downto 0)
/tb_ipd/out_x_arr(1).data(13 downto 0)
/tb_ipd/out_x_arr(2).data(13 downto 0)
/tb_ipd/out_x_arr(3).data(13 downto 0)
Out Y
Out X
/tb_ipd/out_y_arr(1).data(13 downto 0)
/tb_ipd/out_y_arr(2).data(13 downto 0)
/tb_ipd/out_y_arr(3).data(13 downto 0)
/tb_ipd/out_y_arr(0).data(13 downto 0)
{32'h00000013}...
0
14'd0
14'd8191
{00000013} {000000AF} 0 0
(IN)
(Out X)
(Out Y)
Now 7170 ns
Cursor 1 2932.5 ns
Cursor 3 3807.5 ns
3000 ns
3500 ns
2932.5 ns
875 ns
3807.5

Figure 4 : Delays measurements 2

REJ_IPD TEST

When delays are set by the register interface, signals can be unstable during the max length of the delay-line. So after reset or reconfiguration the signal Rej_IPD is pulled high. Such as the out signal is invalid during this period.

- At the beginning of the simulation Rej_IPD is pulled high because the FPGA start up
- On the first yellow marker, delays are set by interface register, Rej_IPD is pulled high again during $1280 = 256 * 5ns$
- On the third yellow marker we can see an another writing configuration by register interface.

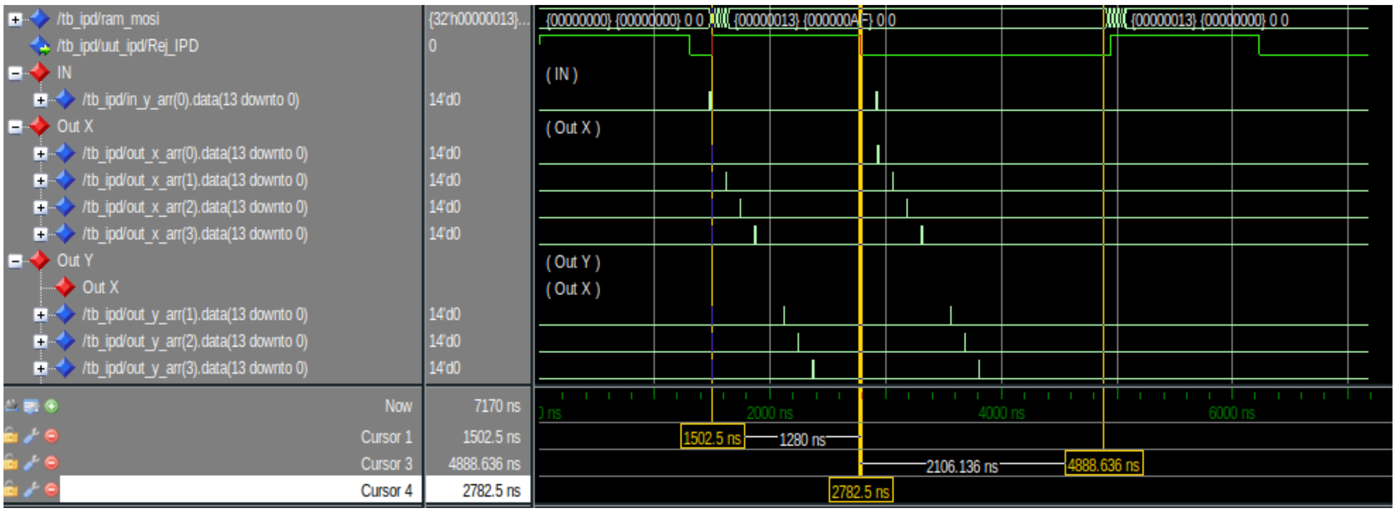

/tb_ipd/ram_mosi
/tb_ipd/uut_ipd/Rej_IPD
IN
/tb_ipd/in_y_arr(0).data(13 downto 0)
Out X
/tb_ipd/out_x_arr(0).data(13 downto 0)
/tb_ipd/out_x_arr(1).data(13 downto 0)
/tb_ipd/out_x_arr(2).data(13 downto 0)
/tb_ipd/out_x_arr(3).data(13 downto 0)
Out Y
Out X
/tb_ipd/out_y_arr(1).data(13 downto 0)
/tb_ipd/out_y_arr(2).data(13 downto 0)
/tb_ipd/out_y_arr(3).data(13 downto 0)
{32'h00000013}...
0
14'd0
{00000000} {00000000} 0 0
{00000013} {000000AF} 0 0
{00000013} {00000000} 0 0
(IN)
(Out X)
(Out Y)
Now 7170 ns
Cursor 1 1502.5 ns
Cursor 3 4888.636 ns
Cursor 4 2782.5 ns
2000 ns
4000 ns
6000 ns
1502.5 ns
1280 ns
2106.136 ns
4888.636 ns
2782.5 ns

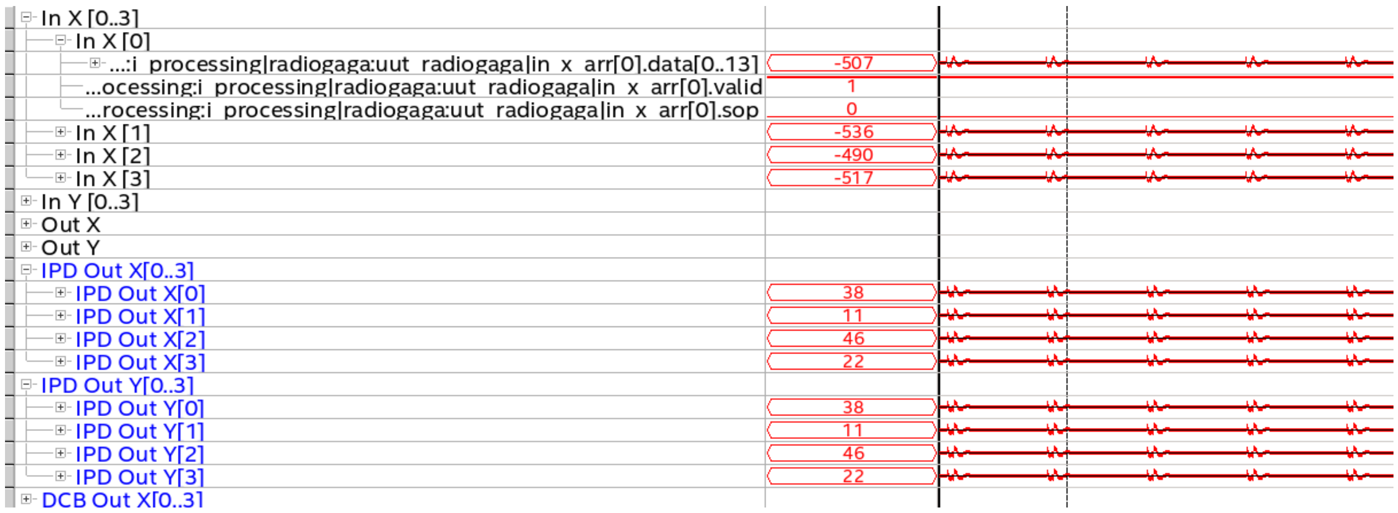
Figure 5 : Rej_IPD measurement

Signal TAP

SHIFT TEST

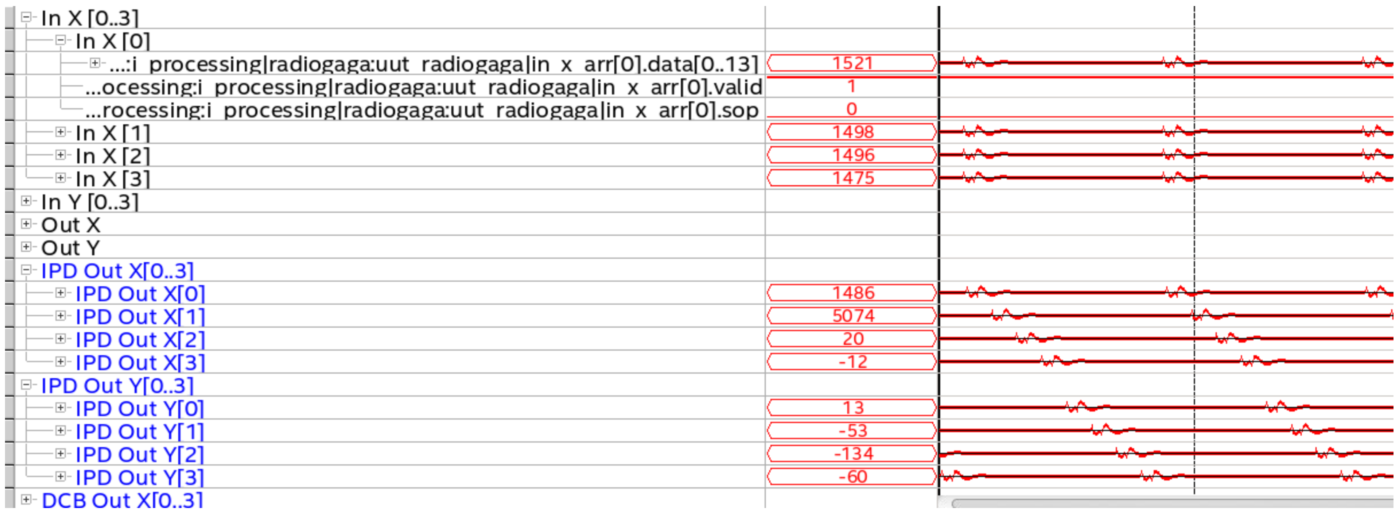
To test the FPGA in real condition, I set a signal tap with some interesting signal. On input of the card. I set a pulse sent every milliseconds.

On the figure 6 we can see a test with delay set to *0* and the figure 7 incrementally by *25* such as in simulation



```
In X [0..3]
  In X [0]
    ...:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].data[0..13]   -507
    ...ocessing:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].valid   1
    ...rocessing:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].sop    0
  In X [1]      -536
  In X [2]      -490
  In X [3]      -517
In Y [0..3]
Out X
Out Y
IPD Out X[0..3]
  IPD Out X[0]   38
  IPD Out X[1]   11
  IPD Out X[2]   46
  IPD Out X[3]   22
IPD Out Y[0..3]
  IPD Out Y[0]   38
  IPD Out Y[1]   11
  IPD Out Y[2]   46
  IPD Out Y[3]   22
DCB Out X[0..3]
```

Figure 6 : Delay set to 0

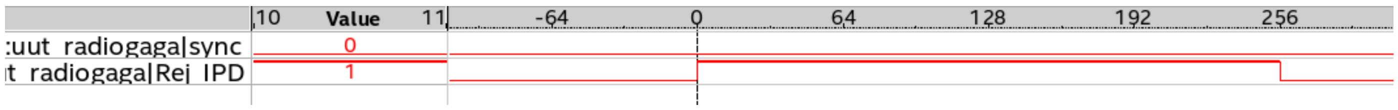


```
In X [0..3]
  In X [0]
    ...:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].data[0..13]   1521
    ...ocessing:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].valid   1
    ...rocessing:i_processing|radiogaga:uut_radiogaga|in_x_arr[0].sop    0
  In X [1]      1498
  In X [2]      1496
  In X [3]      1475
In Y [0..3]
Out X
Out Y
IPD Out X[0..3]
  IPD Out X[0]   1486
  IPD Out X[1]   5074
  IPD Out X[2]   20
  IPD Out X[3]   -12
IPD Out Y[0..3]
  IPD Out Y[0]   13
  IPD Out Y[1]   -53
  IPD Out Y[2]   -134
  IPD Out Y[3]   -60
DCB Out X[0..3]
```

Figure 7 : Delay start 0 to 175 by 25

REJ_IPD TEST

To test it, i just put a trigger on Rej_IPD, reconfigure the delays and we see the signal is pulled HIGH during 255 sample.



```
              ,10   Value   11,  -64   0   64   128   192   256
:uut_radiogaga|sync    0
t_radiogaga|Rej_IPD    1
```

Figure 8 : REJ_IPD test

3.1.2 DCB

Introduction

This document will present and confirm the fonctionnement of DCB module. We will see what it expected in simulation. And confirm the fonctionnement with Signal Tap in the FPGA

Presentation

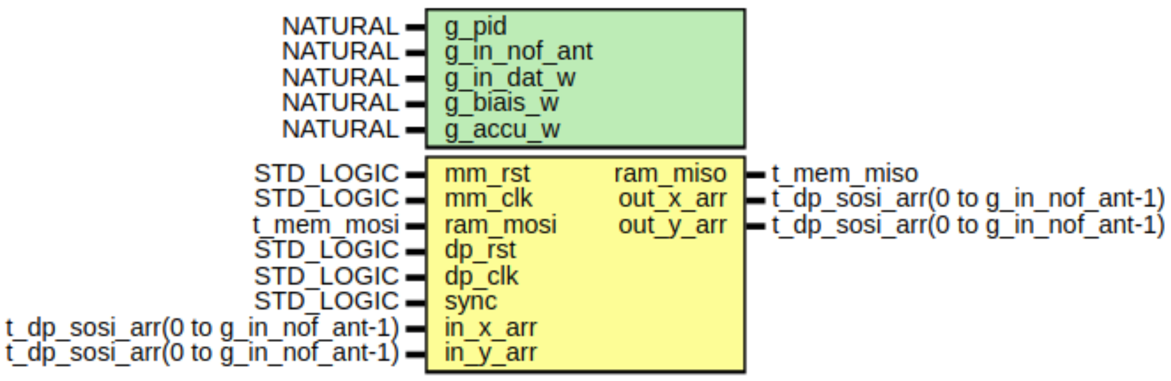


Figure 1 : DCB entity

The figure 1 show the port of the module. The input is directly connected to the output of IPD module. Input data bus is size of 8x14 bits and 8x16 bits in output. The goal of this module is to add biais in order to delete DC offset from the ADC. To do this the module allow to estimate the average of the input. To read average and configure biais the register interface is set like below.
The module has the address offset of `16#1350000#`.
16#1350000# = c_processing_address_offset + c_radiogaga_address_offset + c_DCB_address_offset

Name	Address	Read	Write	Width	Comment
c_DCB_biais_x_offset	16#00#	x	x	4	Delete DC offset on ADC
c_DCB_biais_y_offset	16#10#	x	x	4	Delete DC offset on ADC
c_DCB_accu_x_low_offset	16#20#	x		4	Lower bits of accumulator
c_DCB_accu_y_low_offset	16#30#	x		4	Lower bits of accumulator
c_DCB_accu_x_high_offset	16#40#	x		4	Higher bits of accumulator
c_DCB_accu_y_high_offset	16#50#	x		4	Higher bits of accumulator

Simulation

BIAIS ADD

DCB has 2 writable registers, `biais_x` and `biais_y`. Those biais has a width of 16 bit, 14 for integer part and 2 for decimal part. If we apply a biais of 10, these are :

$$(1010)_2 = (10)_{10}$$

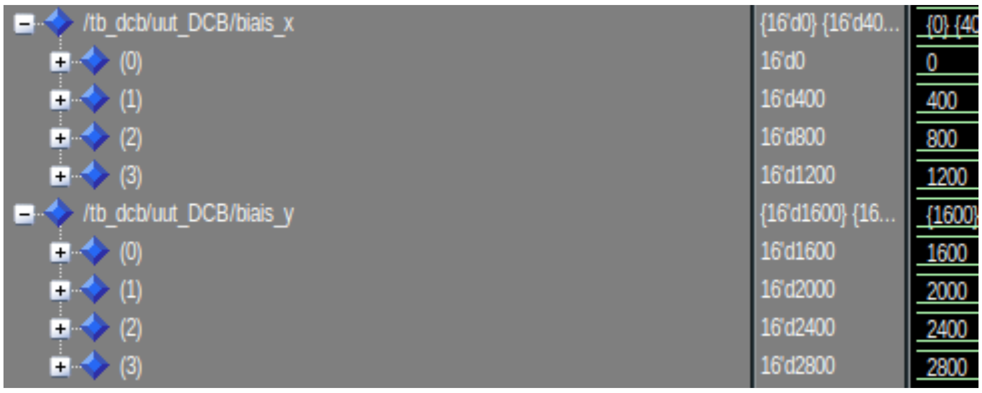
$(1010 + 00)_2 = (40)_{10}$

So we can apply biais to both polarization of each antenna. In simulation, I set biais incrementally by 100, like this we can see + 100 on each channels.

To set biais we have to write in following registers :

# X polar
16#1350000# = 0
16#1350001# = 400
16#1350002# = 800
16#1350003# = 1200

# Y polar
16#1350010# = 1600
16#1350011# = 2000
16#1350012# = 2400
16#1350013# = 2800

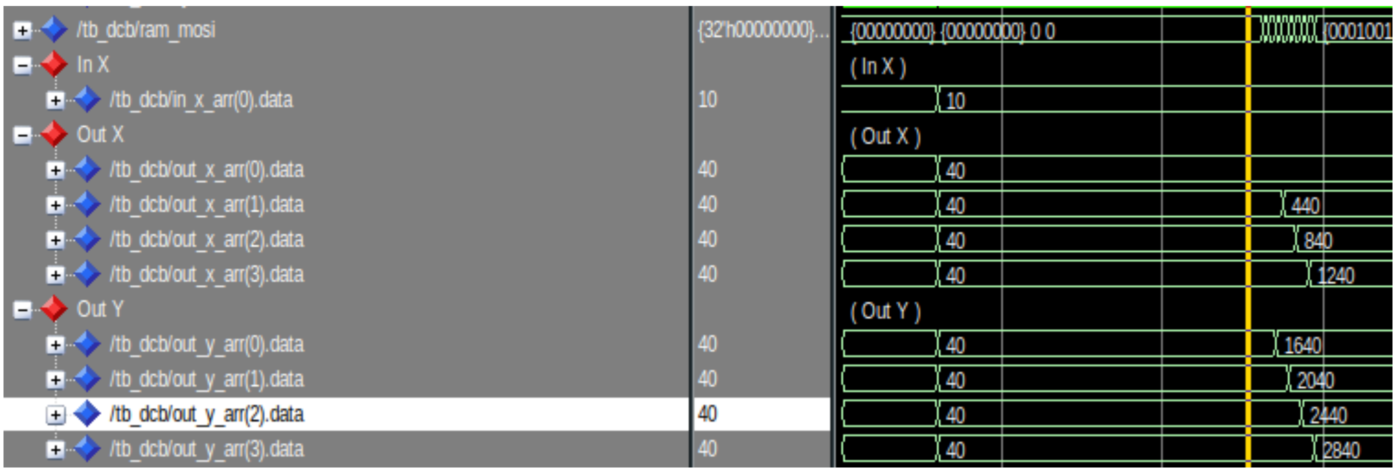


```
/tb_dcb/uut_DCB/biais_x   {16'd0} {16'd40...   {0} {40
  (0)                     16'd0                0
  (1)                     16'd400              400
  (2)                     16'd800              800
  (3)                     16'd1200             1200
/tb_dcb/uut_DCB/biais_y   {16'd1600} {16...    {1600}
  (0)                     16'd1600             1600
  (1)                     16'd2000             2000
  (2)                     16'd2400             2400
  (3)                     16'd2800             2800
```

Figure 2 : Biais configuration

The value are multiply by 4 caused by 2 bit added for the decimal part.

So now we apply a signal of 10 on all the input :



```
/tb_dcb/ram_mosi            {32'h00000000}...  {00000000} {00000000} 0 0   {0001001
In X                                           ( In X )
  /tb_dcb/in_x_arr(0).data  10                 10
Out X                                          ( Out X )
  /tb_dcb/out_x_arr(0).data 40                 40
  /tb_dcb/out_x_arr(1).data 40                 40      440
  /tb_dcb/out_x_arr(2).data 40                 40      840
  /tb_dcb/out_x_arr(3).data 40                 40      1240
Out Y                                          ( Out Y )
  /tb_dcb/out_y_arr(0).data 40                 40      1640
  /tb_dcb/out_y_arr(1).data 40                 40      2040
  /tb_dcb/out_y_arr(2).data 40                 40      2440
  /tb_dcb/out_y_arr(3).data 40                 40      2840
```

Figure 3 : Biais test

AVERAGE ESTIMATION

To set biais we need to estimate the average of the input. To calcul it, the FPGA accumulate the input during 200M of sample (1 sec). Is difficult to divide in vhdl, so the module return the accumulation not normalized. The division by 200M will be done by the soft on server.

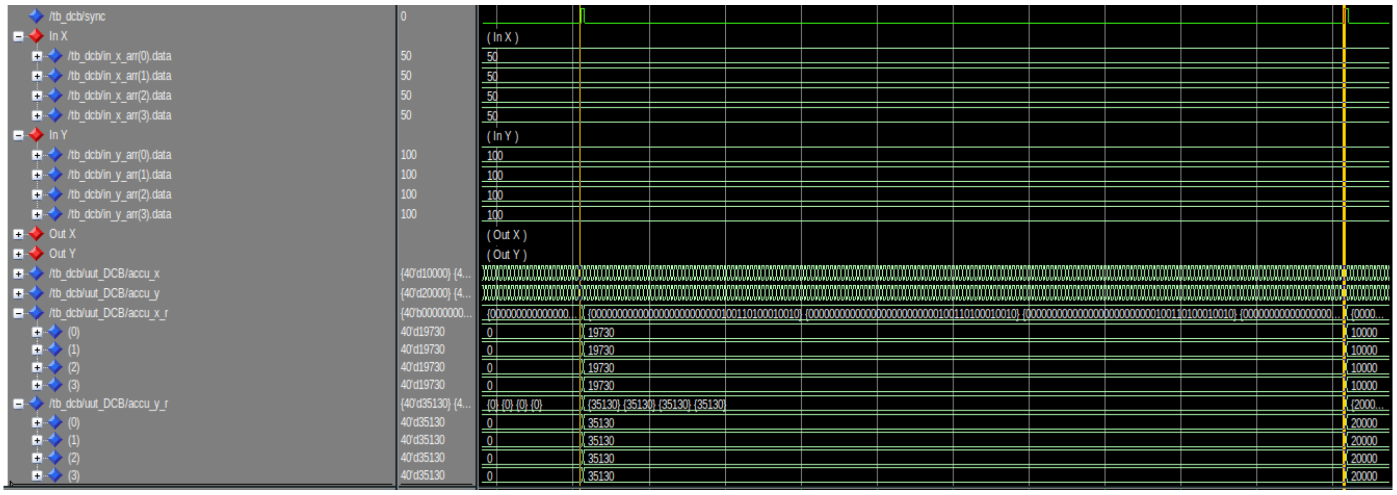


Figure 4 : Average test

On the figure 4 we can see the signal accumulate on a period of 1 µs (1 sec in reality). On the right bottom corner, the accumulation is latched on a rising edge of `sync` and we can see the value of the accumulation during 1µs. So on the X input it gives :

In FPGA

n_sample = 1 000ns/5ns
n_sample = 200
accu_x_r = 50 * 200
accu_x_r = 10 000

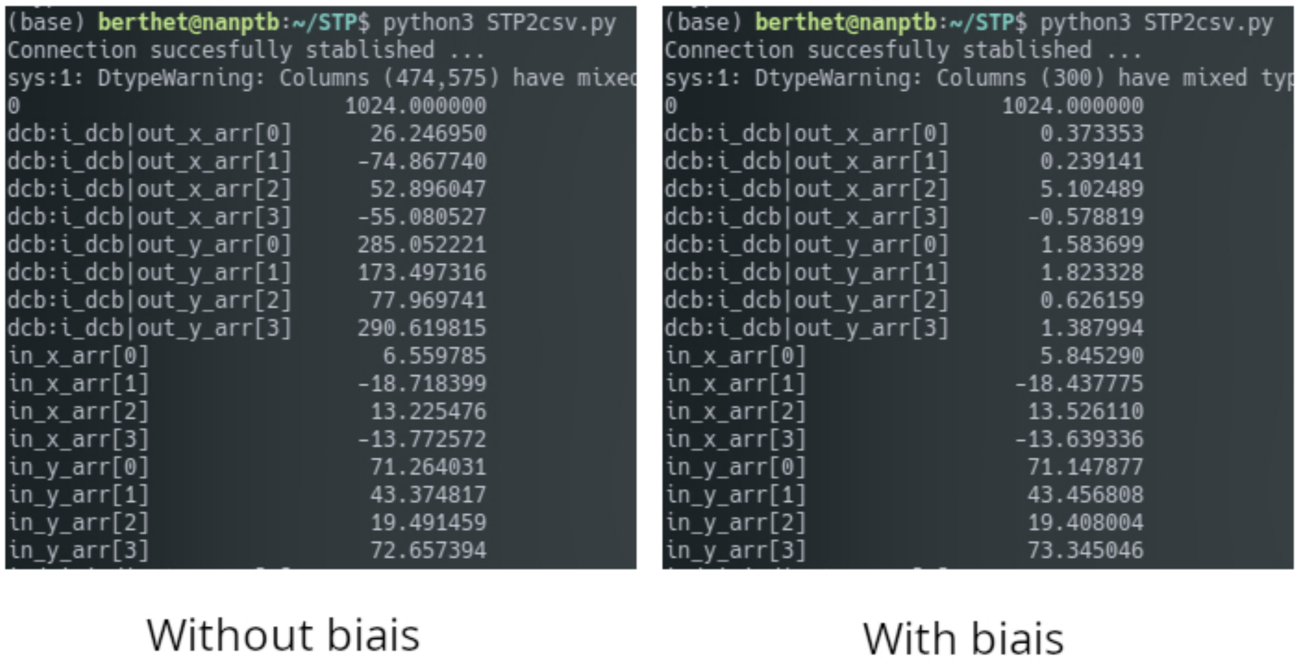
In soft

moyenne = 10 000 / 200
moyenne = 50

Signal TAP

BIAIS ADD

To verify the fonctionnement of this module on the FPGA, the input of the card are still open. To estimate the average on each input I realized a little python program to parse the signal TAP data. With and without biais configuration it gives :



```
(base) berthet@nanptb:~/STP$ python3 STP2csv.py
Connection succesfully stablished ...
sys:1: DtypeWarning: Columns (474,575) have mixe
0                          1024.000000
dcb:i_dcb|out_x_arr[0]       26.246950
dcb:i_dcb|out_x_arr[1]      -74.867740
dcb:i_dcb|out_x_arr[2]       52.896047
dcb:i_dcb|out_x_arr[3]      -55.080527
dcb:i_dcb|out_y_arr[0]      285.052221
dcb:i_dcb|out_y_arr[1]      173.497316
dcb:i_dcb|out_y_arr[2]       77.969741
dcb:i_dcb|out_y_arr[3]      290.619815
in_x_arr[0]                   6.559785
in_x_arr[1]                 -18.718399
in_x_arr[2]                  13.225476
in_x_arr[3]                 -13.772572
in_y_arr[0]                  71.264031
in_y_arr[1]                  43.374817
in_y_arr[2]                  19.491459
in_y_arr[3]                  72.657394
```

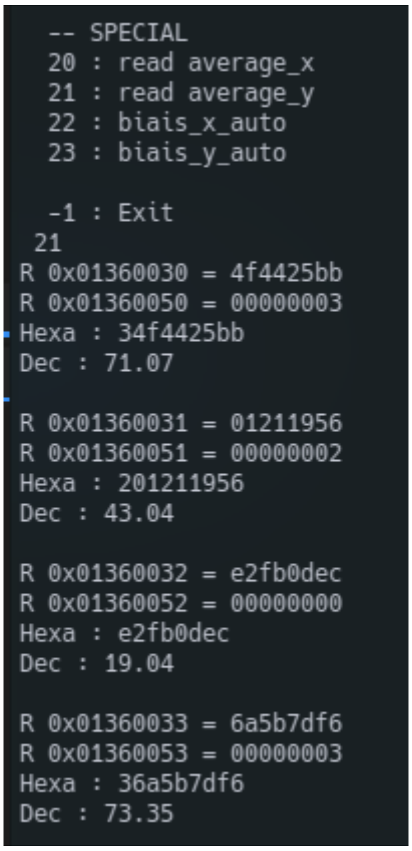
```
(base) berthet@nanptb:~/STP$ python3 STP2csv.py
Connection succesfully stablished ...
sys:1: DtypeWarning: Columns (300) have mixed typ
0                          1024.000000
dcb:i_dcb|out_x_arr[0]        0.373353
dcb:i_dcb|out_x_arr[1]        0.239141
dcb:i_dcb|out_x_arr[2]        5.102489
dcb:i_dcb|out_x_arr[3]       -0.578819
dcb:i_dcb|out_y_arr[0]        1.583699
dcb:i_dcb|out_y_arr[1]        1.823328
dcb:i_dcb|out_y_arr[2]        0.626159
dcb:i_dcb|out_y_arr[3]        1.387994
in_x_arr[0]                   5.845290
in_x_arr[1]                 -18.437775
in_x_arr[2]                  13.526110
in_x_arr[3]                 -13.639336
in_y_arr[0]                  71.147877
in_y_arr[1]                  43.456808
in_y_arr[2]                  19.408004
in_y_arr[3]                  73.345046
```

Figure 5: Average estimation

On the left of figure 5 we can see the average on 1024 sample on each input. DCB out is just the input with 2 bit added for decimal part.On the right of the figure 5 we can see when we applied biais on each entry. To this we apply `-26.2 (4 x 6.55)` on in_x_arr[0] `+74.9` on in_x_arr[1] etc. Values are nears to zero, what is expected.

5.2. Average Estimation

Now to read accumulation from the FPGA it was a bit complicated. The value is accumulate in a 40 bits word. But the register interface accept only until 32 bits word. So i cut the word in 2 part ; low_acc(32 lower bits) and high_acc(8 higher bits). In the soft those value are stock in two variable `uint64_t`. The MSB is duplicated on the whole word to keep the sign. Then the hig_acc part is shifted by 32 bits. Finally the 2 parts are added to be divided by 200M.



```
  -- SPECIAL
  20 : read average_x
  21 : read average_y
  22 : biais_x_auto
  23 : biais_y_auto

  -1 : Exit
 21
R 0x01360030 = 4f4425bb
R 0x01360050 = 00000003
Hexa : 34f4425bb
Dec : 71.07

R 0x01360031 = 01211956
R 0x01360051 = 00000002
Hexa : 201211956
Dec : 43.04

R 0x01360032 = e2fb0dec
R 0x01360052 = 00000000
Hexa : e2fb0dec
Dec : 19.04

R 0x01360033 = 6a5b7df6
R 0x01360053 = 00000003
Hexa : 36a5b7df6
Dec : 73.35
```

Figure 6 : Read average

For Example on the figure 6 we can see the DC offset on the Y polarisation. That's correspond with the figure 5.

3.1.3 Presum

Introduction

This page will confirm the fonctionnement of the presum module. We will see what it expected in simulation. And confirm the fonctionnement with signaltap in the FPGA

Presentation

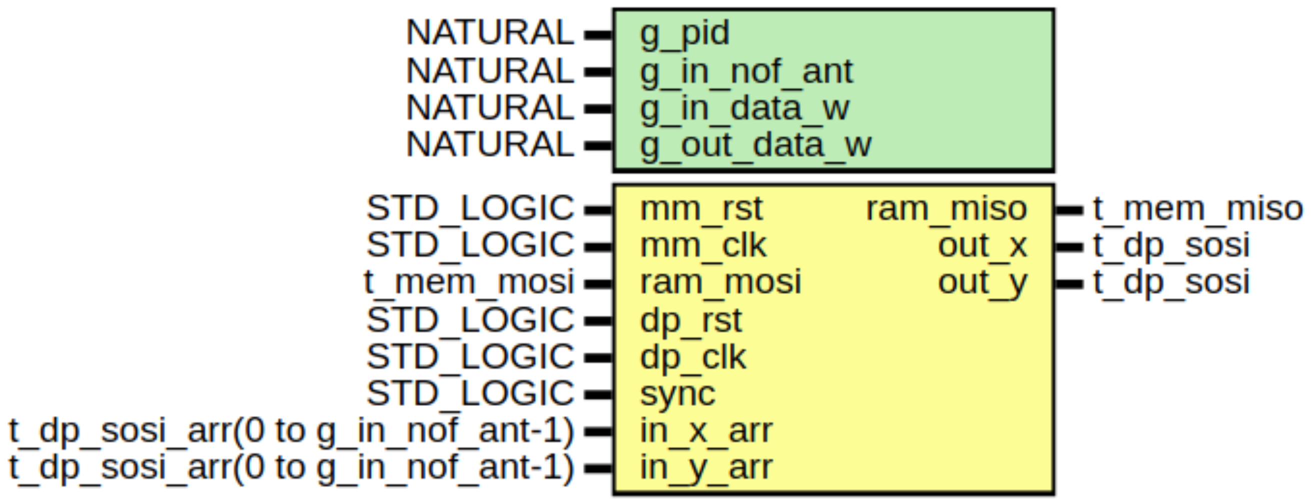


Figure 1 : Presum entity

The figure 1 shows ports of the module. The input is directly connected to the output of DCB. Input data bus is size of 8x16 bits and 2x18 bits in output. This module will just add all antennas on both polarization. In register interface it's allow to disable one or more antenna. The register interface is set like below. The module has the address offset of `16#1360000#`.

16#1360000# = c_processing_address_offset + c_radiogaga_address_offset + c_presum_address_offset

Name	Address	Read	Write	Width	Comment
c_presum_ant_sel_x_offset	16#00#	x	x	4	Disable antennas on x polar
c_presum_ant_sel_y_offset	16#10#	x	x	4	Disable antennas on Y polar

Simulation

ENABLE/DISABLE ANTENNAS

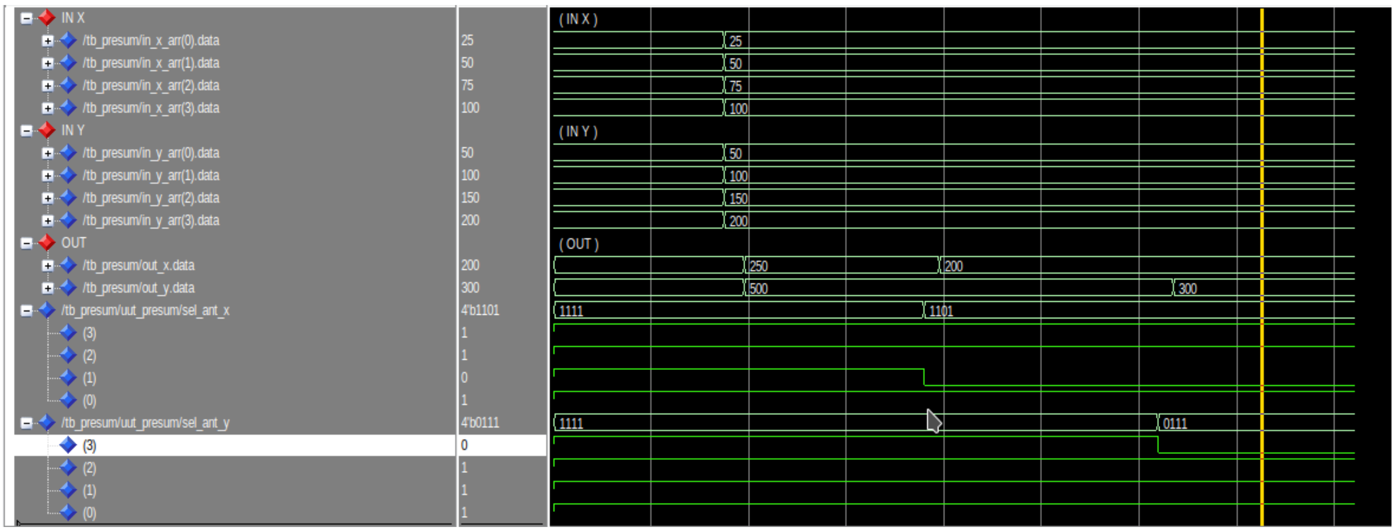
The register ant_sel_x and ant_sel_y are initialized to `'1'` in the FPGA. Like this all inputs are enable. To disable we have to write `'0'` to the antenna which we want want to disable.
For exemple to disable the 2nd and 4th antenna of the X polarization we have to write in the register interface like below :

# X polar
16#1360000# = 1
16#1360001# = 0
16#1360002# = 1
16#1360003# = 0

SUMMATION

Now the FPGA check for each antenna if ant_sel is `'0'` or `'1'` and if is `'1'` datas of the inputs are added, else the input is replaced by zero.

On the figure 2, at the beginning of simulation, all antennas are set to `'1'` so the output is equal to sum of 4 antennas for each polarization. Then the 2nd and the 4th antenna are disable for X and Y polar respectively.



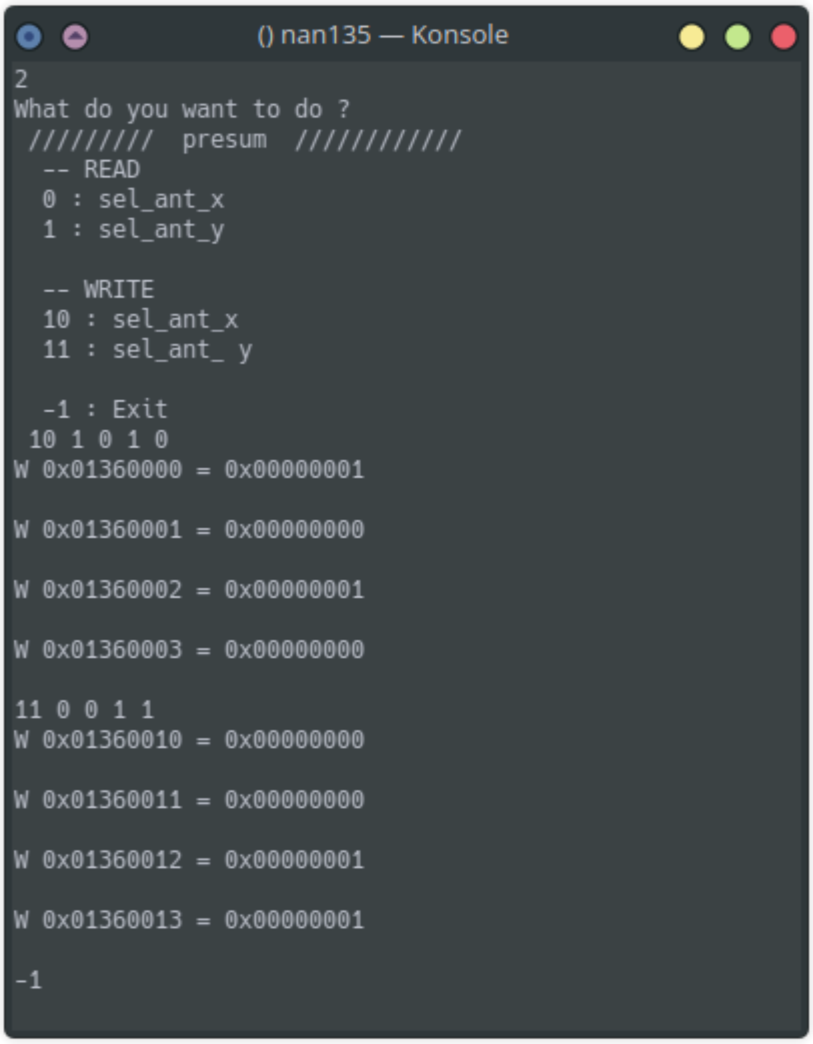
IN X	
/tb_presum/in_x_arr(0).data	25
/tb_presum/in_x_arr(1).data	50
/tb_presum/in_x_arr(2).data	75
/tb_presum/in_x_arr(3).data	100
IN Y	
/tb_presum/in_y_arr(0).data	50
/tb_presum/in_y_arr(1).data	100
/tb_presum/in_y_arr(2).data	150
/tb_presum/in_y_arr(3).data	200
OUT	
/tb_presum/out_x.data	200
/tb_presum/out_y.data	300
/tb_presum/uut_presum/sel_ant_x	4'b1101
(3)	1
(2)	1
(1)	0
(0)	1
/tb_presum/uut_presum/sel_ant_y	4'b0111
(3)	0
(2)	1
(1)	1
(0)	1

Figure 2 : Summation test

Signal TAP

ENABLE/DISABLE ANTENNAS

On the figure 3 we can read and write in both register. For this test we can see I disable the 2nd and the 4th antenna for the X polar. And for the the 1st and 2nd antenna are disable.

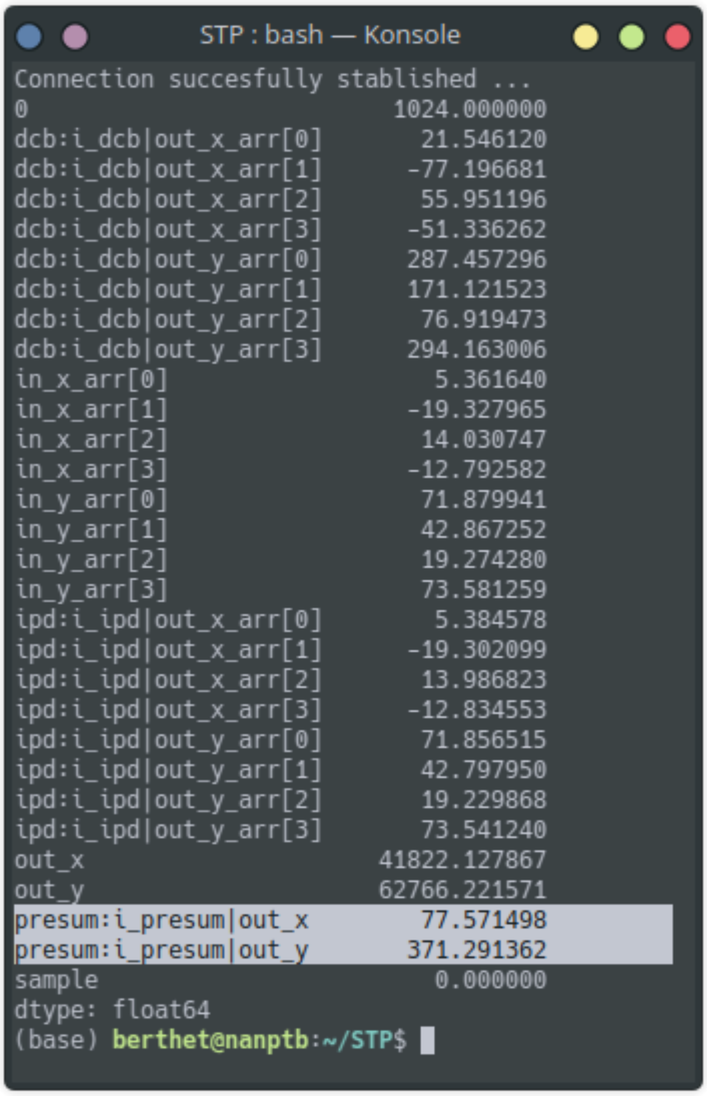


```
2
What do you want to do ?
 /////////  presum  ///////////
  -- READ
  0 : sel_ant_x
  1 : sel_ant_y

  -- WRITE
  10 : sel_ant_x
  11 : sel_ant_ y

  -1 : Exit
 10 1 0 1 0
W 0x01360000 = 0x00000001

W 0x01360001 = 0x00000000

W 0x01360002 = 0x00000001

W 0x01360003 = 0x00000000

11 0 0 1 1
W 0x01360010 = 0x00000000

W 0x01360011 = 0x00000000

W 0x01360012 = 0x00000001

W 0x01360013 = 0x00000001

-1
```

Figure 3 : Disable antennas in test soft

SUMMATION

To verify the good fonctionnement of it, we will use the same python program like we used to test DCB. It export data from signal TAP and we can read the average of each polar in different modules.



```
Connection succesfully stablished ...
0                               1024.000000
dcb:i_dcb|out_x_arr[0]            21.546120
dcb:i_dcb|out_x_arr[1]           -77.196681
dcb:i_dcb|out_x_arr[2]            55.951196
dcb:i_dcb|out_x_arr[3]           -51.336262
dcb:i_dcb|out_y_arr[0]           287.457296
dcb:i_dcb|out_y_arr[1]           171.121523
dcb:i_dcb|out_y_arr[2]            76.919473
dcb:i_dcb|out_y_arr[3]           294.163006
in_x_arr[0]                        5.361640
in_x_arr[1]                      -19.327965
in_x_arr[2]                       14.030747
in_x_arr[3]                      -12.792582
in_y_arr[0]                       71.879941
in_y_arr[1]                       42.867252
in_y_arr[2]                       19.274280
in_y_arr[3]                       73.581259
ipd:i_ipd|out_x_arr[0]             5.384578
ipd:i_ipd|out_x_arr[1]           -19.302099
ipd:i_ipd|out_x_arr[2]            13.986823
ipd:i_ipd|out_x_arr[3]           -12.834553
ipd:i_ipd|out_y_arr[0]            71.856515
ipd:i_ipd|out_y_arr[1]            42.797950
ipd:i_ipd|out_y_arr[2]            19.229868
ipd:i_ipd|out_y_arr[3]            73.541240
out_x                          41822.127867
out_y                          62766.221571
presum:i_presum|out_x             77.571498
presum:i_presum|out_y            371.291362
sample                             0.000000
dtype: float64
(base) berthet@nanptb:~/STP$ 
```

Figure 4 : Verify the output of presum

On the figure 4 we can see the sum on highlight lines. presum_out is equal to the sum of DCB_out_x_arr[n] on each polar.

So if we take the exemple beside for X polar :
presum|out_x = dcb|out_x_arr[0] + dcb|out_x_arr[2] presum|out_x = 21.5 + 56 presum|out_x = 77.5

And for the Y polar :

presum|out_y = dcb|out_x_arr[2] + dcb|out_x_arr[3] presum|out_y = 76.9 + 294.2 presum|out_y = 371.1

3.1.4 Filter

Introduction

This page will confirm the fonctionnement of the filter module. We will see what it expected in simulation. And confirm the fonctionnement with signaltap in the FPGA

Presentation

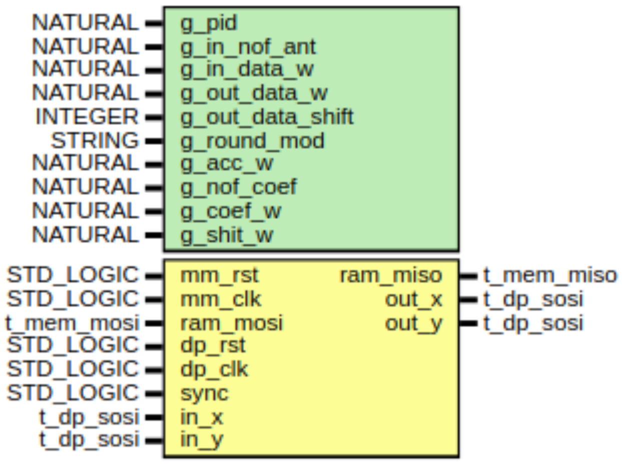


Figure 1 : Filter entiry

The figure 1 shows ports of the module. The input data bus is directly connected to the output of presum. Input data bus is size of 2x18 bits and 2x18 bits in output. This module is a numeric filter. It takes 128 coef and use FIR implementation to filter the signal. It possible to shift the output of the signal to decrease the gain of the filter.
The module has the address offset of `16#1370000#`.

16#1370000# = c_processing_address_offset + c_radiogaga_address_offset + c_filter_address_offset

Name	Address	Read	Write	Width	Comment
c_filter_coef_offset	16#00#	x	x	128	Coefficient of the filter
c_filter_shift_offset	16#FF#	x	x	1	Shift to apply on the output

This module has 2 architectures, **normal** and **transposed**. Difference between both are explained here. Today, is the transposed structure that is used, it allows to use DSP blocks in FPGA, explained here.

This module have important generic port to determine the good fonctionnement.

- **g_round_mod** : This allow to choose the rounding mode for the output. Today only *truncate* is available.
- **g_nof_coef** : Determine the number of coefficient for the implementation. 128 is sufficient for our application.
- **g_out_data_shift** : For the normal architecture only, the shift is not configurable in the register interface so it's declared here.
- **g_acc_w** : $18 + 18 + \frac{\log(128)}{\log(2)} = 44$
- **g_coef_w** : Coefficients are defined on 16 bits.
- **g_shift_w** : The shift value is defined on 6 bits.

Simulation

RAMP COEFFICIENTS

First of all before to test is the filter make the job, we can set coefs from 0 to 128 by 1. To do this we have to write to the following addresses :

16#1370000# = 1
16#1370001# = 2
16#1370002# = 3
...
16#13700126# = 127
16#13700127# = 128

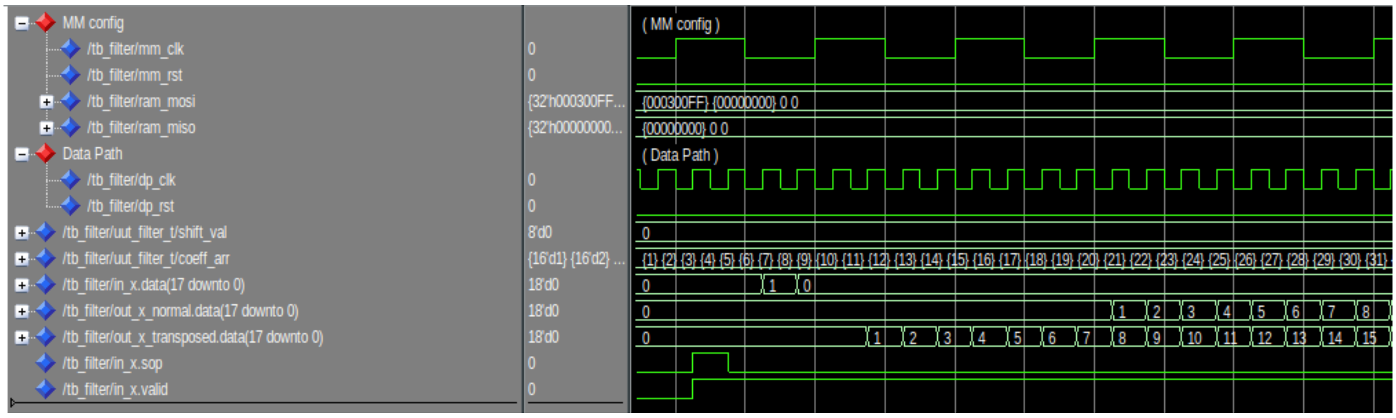


Figure 2 : Coefficient from 0 to 128

On the figure 2 we can see a pulse of **1** is sent on `in_x.data` during only one sample. We have 2 outputs because I instantiate both architecture, normal and transposed. So we can see our pulse multiplied by each coefficient until the 128th.

Info

We can notice the transposed implementation generates less retard than the normal structure.

SHIFT TEST

To compensate the gain involved by the filter. An adjustable shift is implemented. It allow to shift to the right the signal. For exemple :

```
mysignal   <= 80 --(0b1010000)
mysignal   <= shift_right(mysignal, 2)
-- mysignal = 20   (0b10100)
```

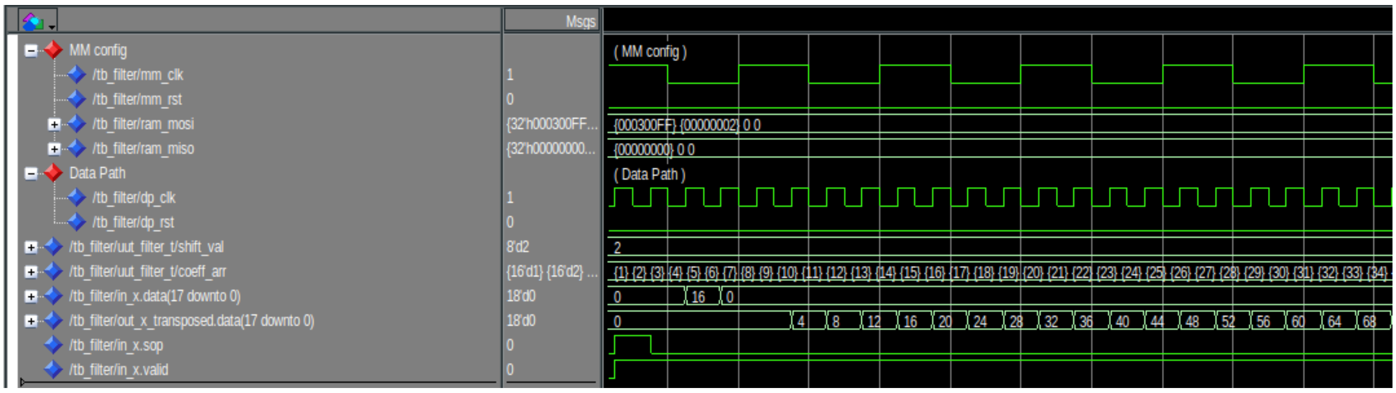


Figure 3 : Shift test

On the figure 3 the pule is now of **16** and the shift of **2**. And we can see on the output the signal is always multiplied by the ramp coefficients but now is divided by $2^2 = 4$

Signal TAP

TEST OF A 36MHZ REJECTOR

To verify the fonctionnement of the filter I directly test it in the FPGA. For it I just generated a sinusoidal with a GBF.

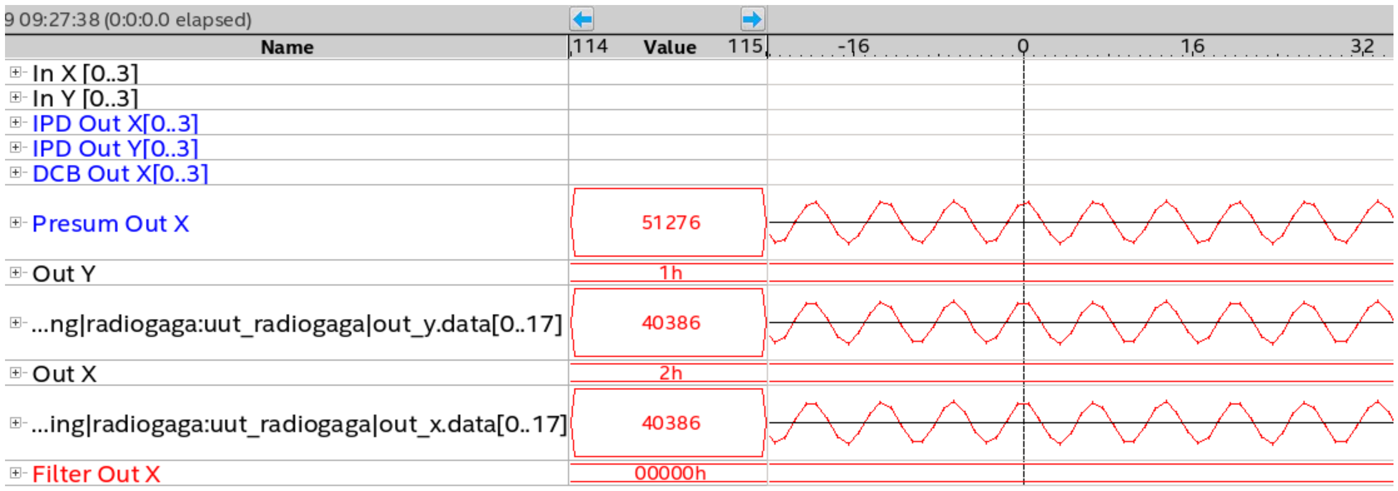
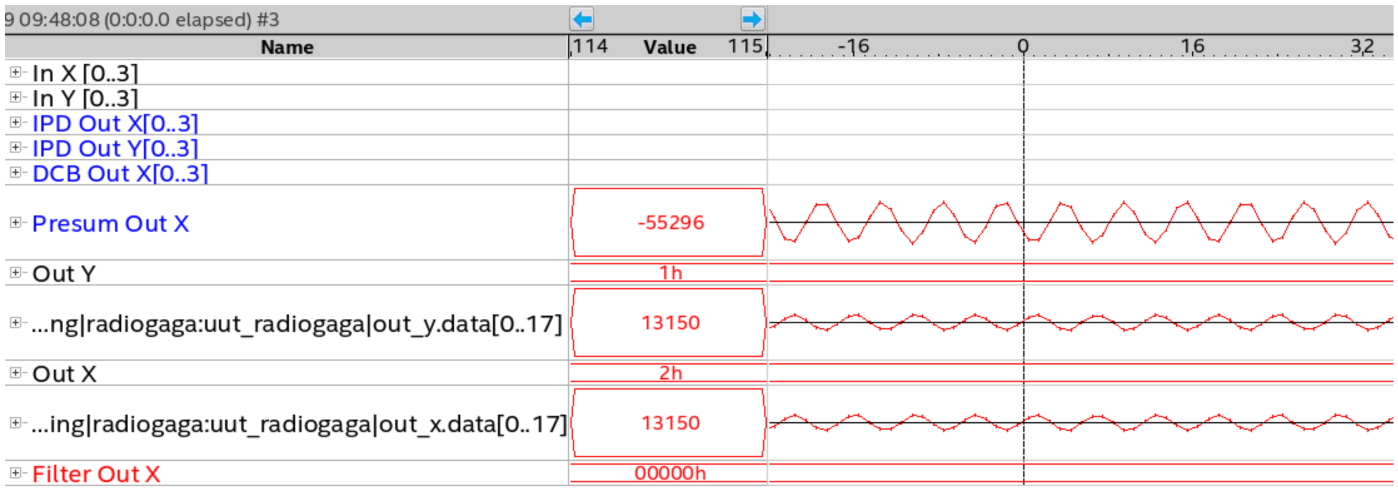
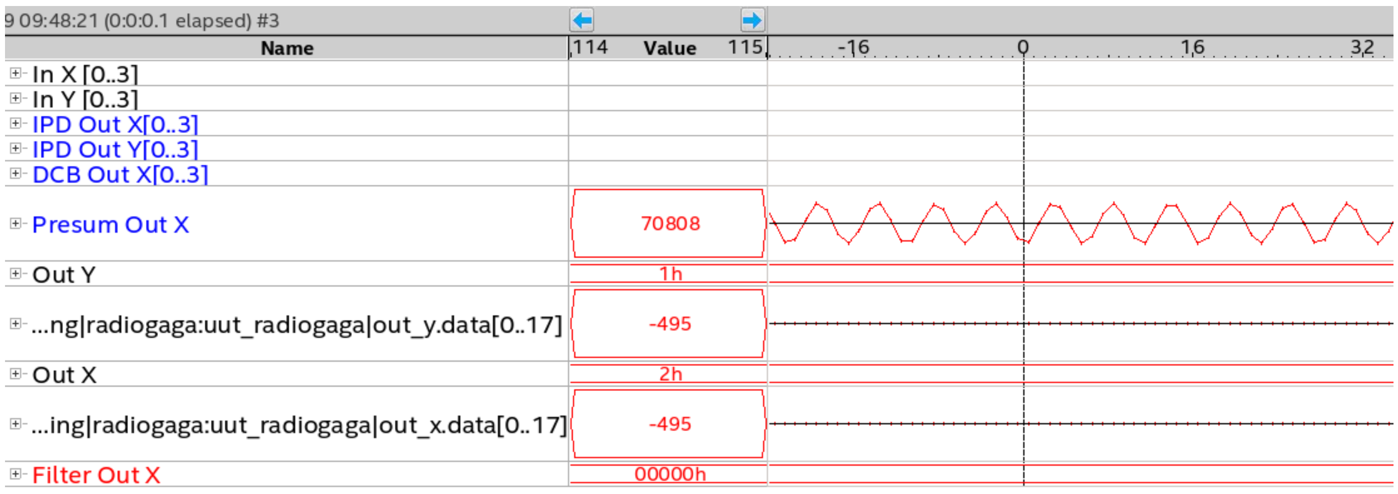


Figure 4 : Signal of 30MHz

On the figure 4 we can see a signal of 30MHz, it pass trough without any attenuation.




9 09:48:21 (0:0:0.1 elapsed) #3

Name	114 Value 115
In X [0..3]	
In Y [0..3]	
IPD Out X[0..3]	
IPD Out Y[0..3]	
DCB Out X[0..3]	
Presum Out X	70808
Out Y	1h
...ng|radiogaga:uut_radiogaga|out_y.data[0..17]	-495
Out X	2h
...ing|radiogaga:uut_radiogaga|out_x.data[0..17]	-495
Filter Out X	00000h

Figure 5 : Signal of 35MHz and 36MHz

Now on the figure 5 we can see the signal is a bit attenuated at 35MHz and completely at 36MHz.
We can also see the spectrum of the output. There is a big reject of 50dB at 36M.

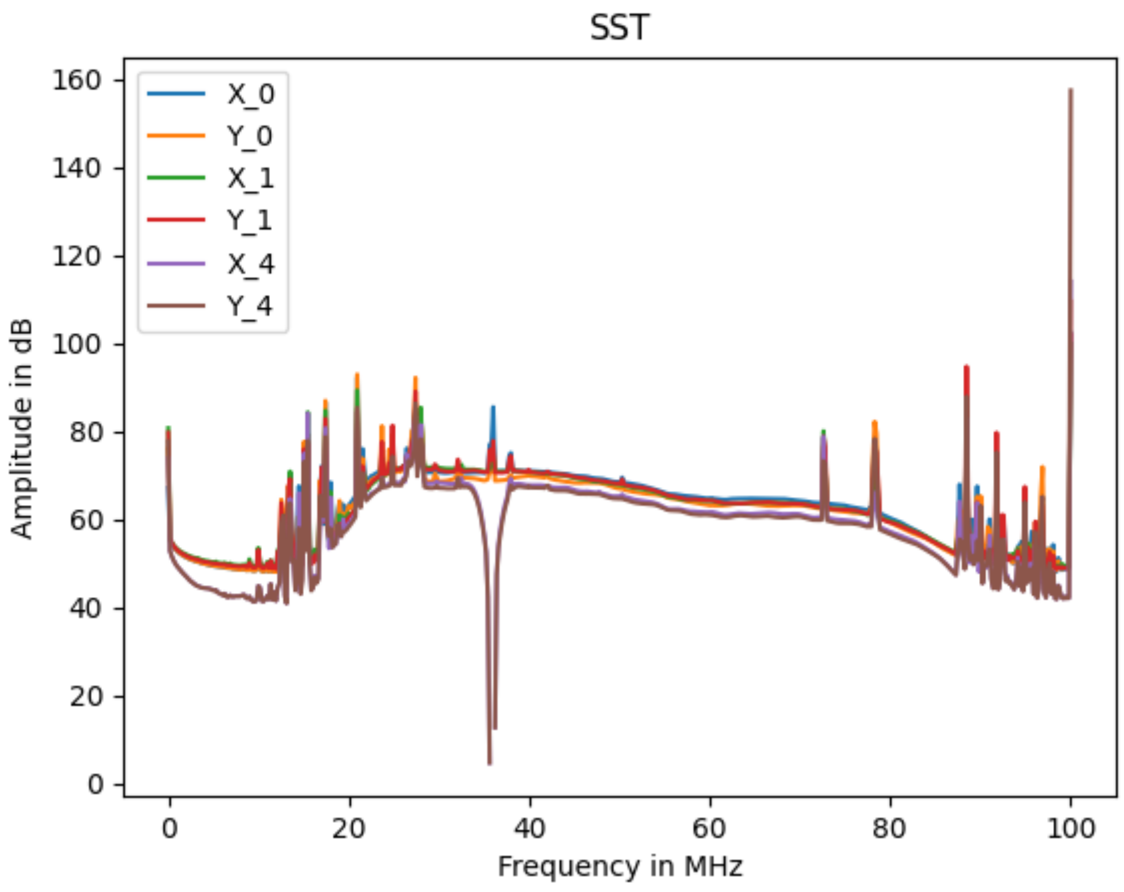


Figure 6 : SST of filter output

Info

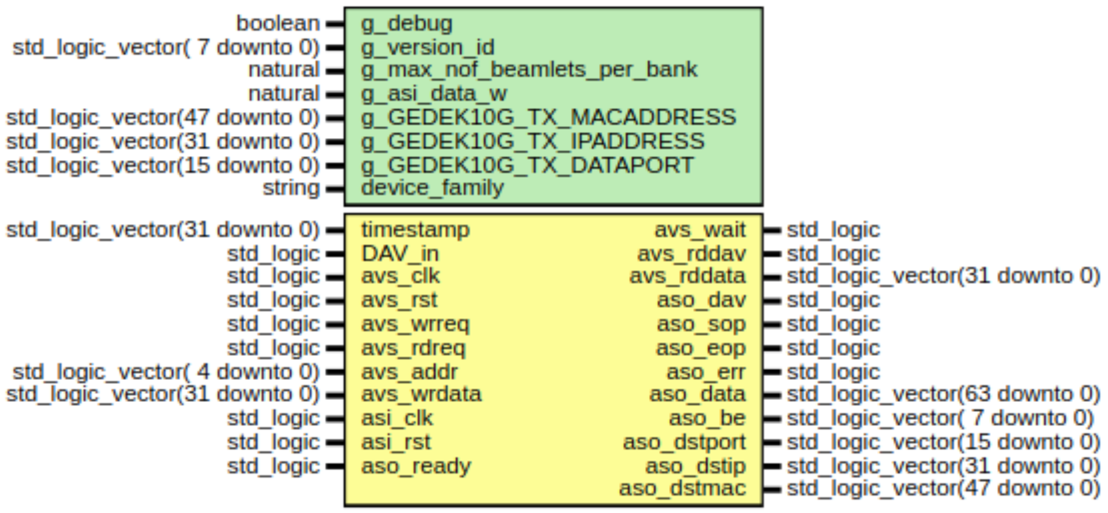
The filter module is instantiate 3 times, so we can apply 3 different filters on the signal. We just have to reconfigure the coefficient in the register interface. To generate coefficient, we used pyFDA explained here : PyFDA doc

3.2 UDP Sender

3.2.1 Introduction

This document will present and confirm the fonctionnement of UDP_sender module. We will see what it expected in simulation. And confirm the fonctionnement with Signal Tap in the FPGA.

3.2.2 Presentation



boolean — g_debug
std_logic_vector(7 downto 0) — g_version_id
natural — g_max_nof_beamlets_per_bank
natural — g_asi_data_w
std_logic_vector(47 downto 0) — g_GEDEK10G_TX_MACADDRESS
std_logic_vector(31 downto 0) — g_GEDEK10G_TX_IPADDRESS
std_logic_vector(15 downto 0) — g_GEDEK10G_TX_DATAPORT
string — device_family
std_logic_vector(31 downto 0) — timestamp
std_logic — DAV_in
std_logic — avs_clk
std_logic — avs_rst
std_logic — avs_wrreq
std_logic — avs_rdreq
std_logic_vector(4 downto 0) — avs_addr
std_logic_vector(31 downto 0) — avs_wrdata
std_logic — asi_clk
std_logic — asi_rst
std_logic — aso_ready
avs_wait — std_logic
avs_rddav — std_logic
avs_rddata — std_logic_vector(31 downto 0)
aso_dav — std_logic
aso_sop — std_logic
aso_eop — std_logic
aso_err — std_logic
aso_data — std_logic_vector(63 downto 0)
aso_be — std_logic_vector(7 downto 0)
aso_dstport — std_logic_vector(15 downto 0)
aso_dstip — std_logic_vector(31 downto 0)
aso_dstmac — std_logic_vector(47 downto 0)

Figure 1 : UDP_sender entity

This has to sent an UDP frame on the network when a cosmic gerbe is detected. The frame will be transmitted by a 10G link. For this on the figure 1 we can see different ports (yellow) of the module.
In the generic interface (green) we can set the default IP configuration of the destination server.

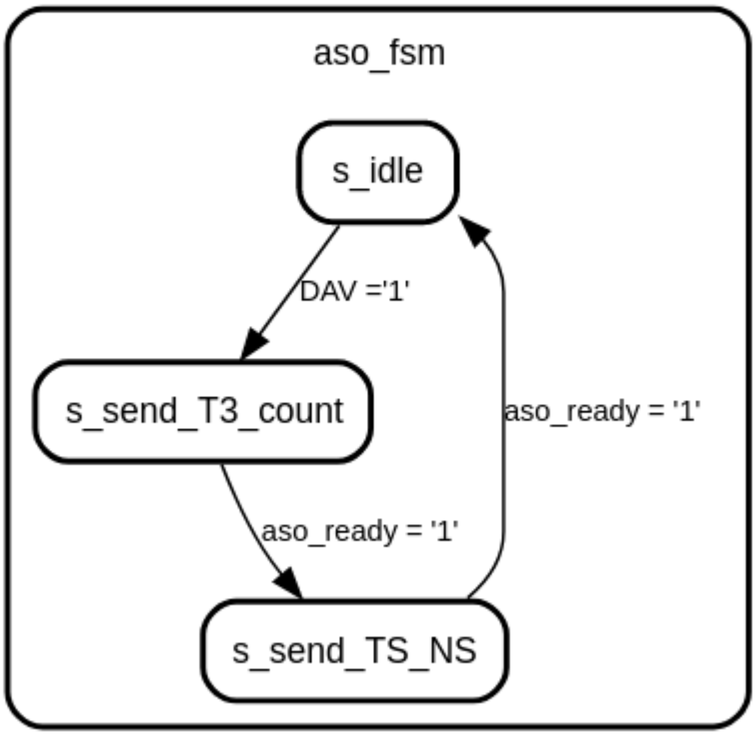

$$\begin{tikzcd}
 & s\_idle \arrow[dl, "DAV ='1'"] & \\
s\_send\_T3\_count \arrow[r, "aso\_ready = '1'"] & s\_send\_TS\_NS \arrow[u, bend right, "aso\_ready = '1'"'] &
\end{tikzcd}$$

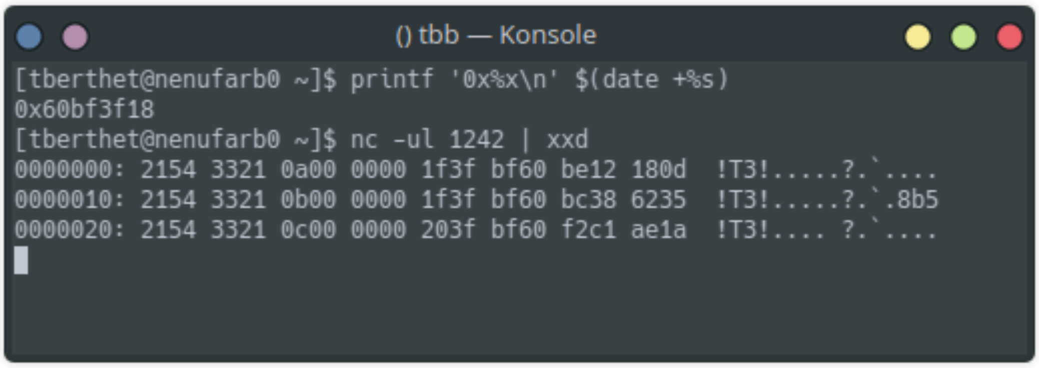
Figure 2 : State machine

On the figure 2 we can see the state machine of the module. By default the module is in **idle** mode. It is waiting a **DAV** from the trigger module.
When a DAV is sent, the module will put 2 packets of 64 bits in the frame :

-- First packet --
!T3! : Header of the frame. 4 ASCII character, so 32 bits width.
count : Number of the frame since the boot of the card.

-- Second packet --
timestamp : Is the timestamp when the trig happened
nanosecond : Is the nanosecond in the timestamp when the trig happened for more accuracy.

So now one the figure 3 we can see reception of a frame with netcat.



```
[tberthet@nenufarb0 ~]$ printf '0x%x\n' $(date +%s)
0x60bf3f18
[tberthet@nenufarb0 ~]$ nc -ul 1242 | xxd
0000000: 2154 3321 0a00 0000 1f3f bf60 be12 180d  !T3!.....?.`....
0000010: 2154 3321 0b00 0000 1f3f bf60 bc38 6235  !T3!.....?.`.8b5
0000020: 2154 3321 0c00 0000 203f bf60 f2c1 ae1a  !T3!.... ?.`....
```

Figure 3 : Reception of a frame with netcat

The module has the address 16#980000#. Below they are the different address of the register interface.

Name	Address	Read	Write	Width	Comment
c_UDP_sender_version_id_offset	16#00#	x		1	Version _ id of the framer
c_UDP_sender_lo_32_mac_offset	16#01#	x	x	1	Lower 32 bits of the mac address
c_UDP_sender_hi_16_mac_offset	16#02#	x	x	1	Higher 32 bits of the mac address
c_UDP_sender_ip_offset	16#03#	x	x	1	Ip address
c_UDP_sender_port_offset	16#04#	x	x	1	Port
c_UDP_sender_manual_dav_offset	16#05#		x	1	Allow to send a frame manually (for test)
c_UDP_sender_delay_ns_offset	16#06#	x	x	1	Time in number of sample to delete the traitement time

3.2.3 Simulation

To simulate this module, we have just have to up DAV during one sample and watch the frame on the 10G bus.

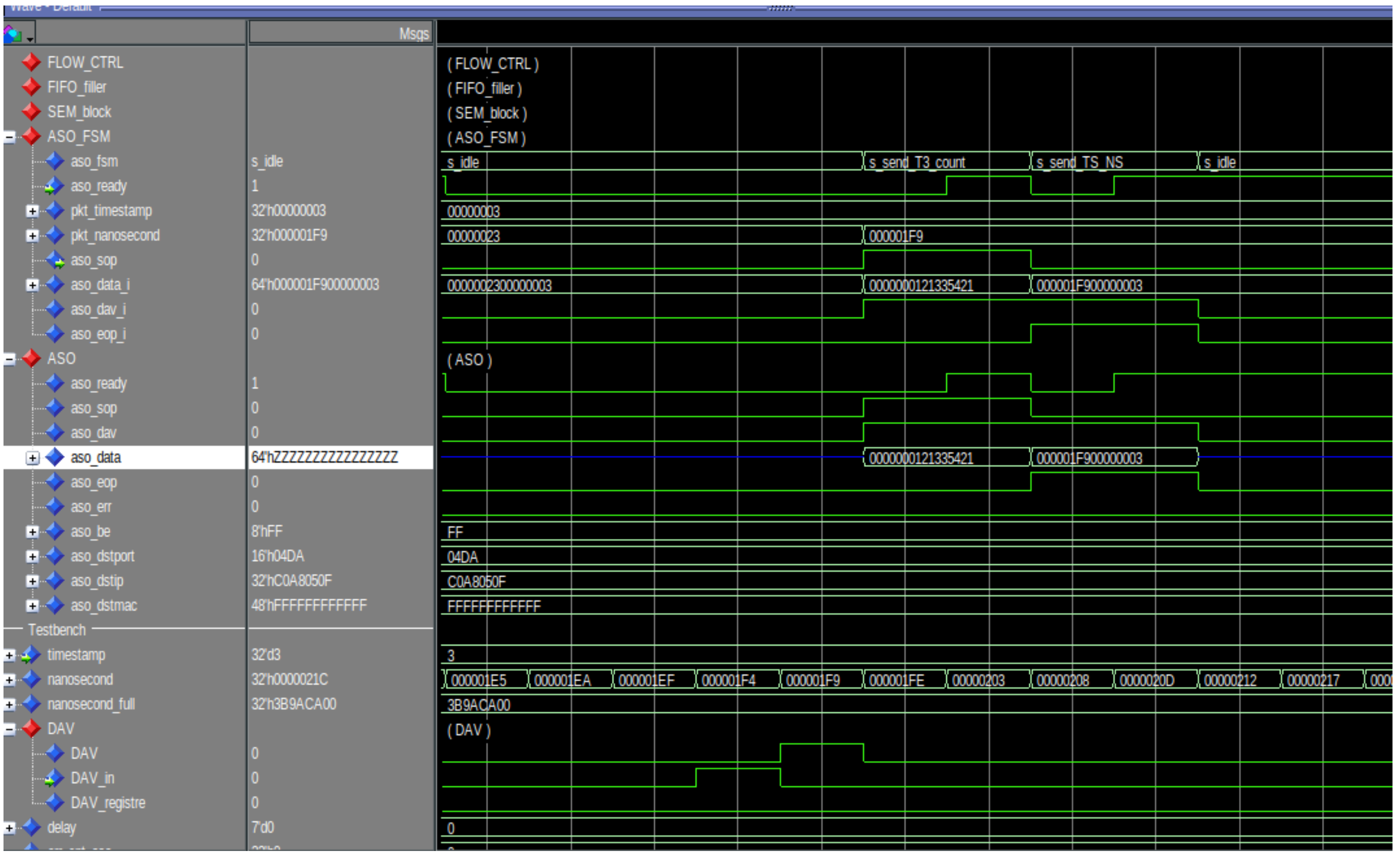


Figure 4 : Test bench of UDP_sender without delay

Here we can see the nanosecond during the pulse is **1F9** and we can see it in the frame on aso_data.

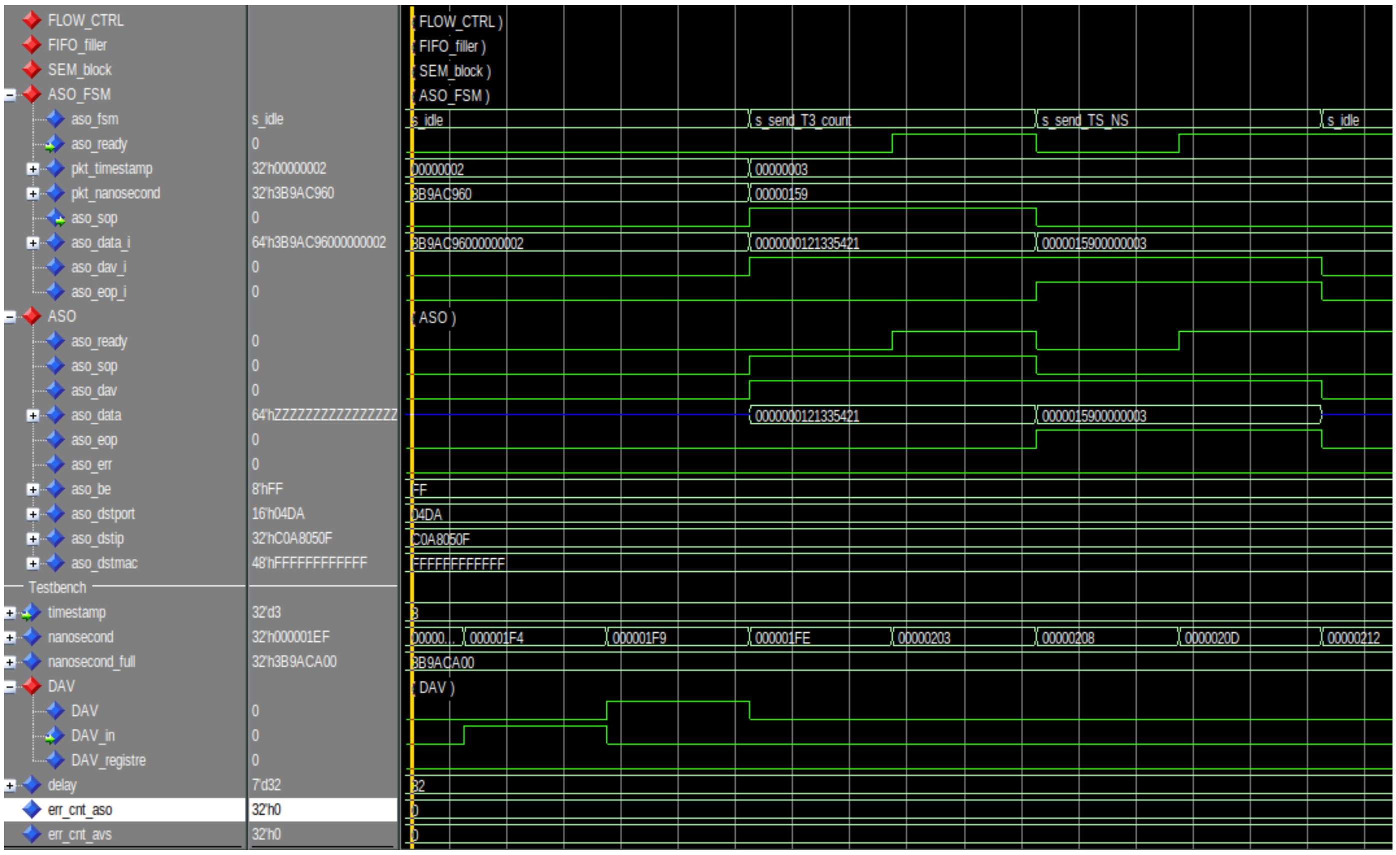
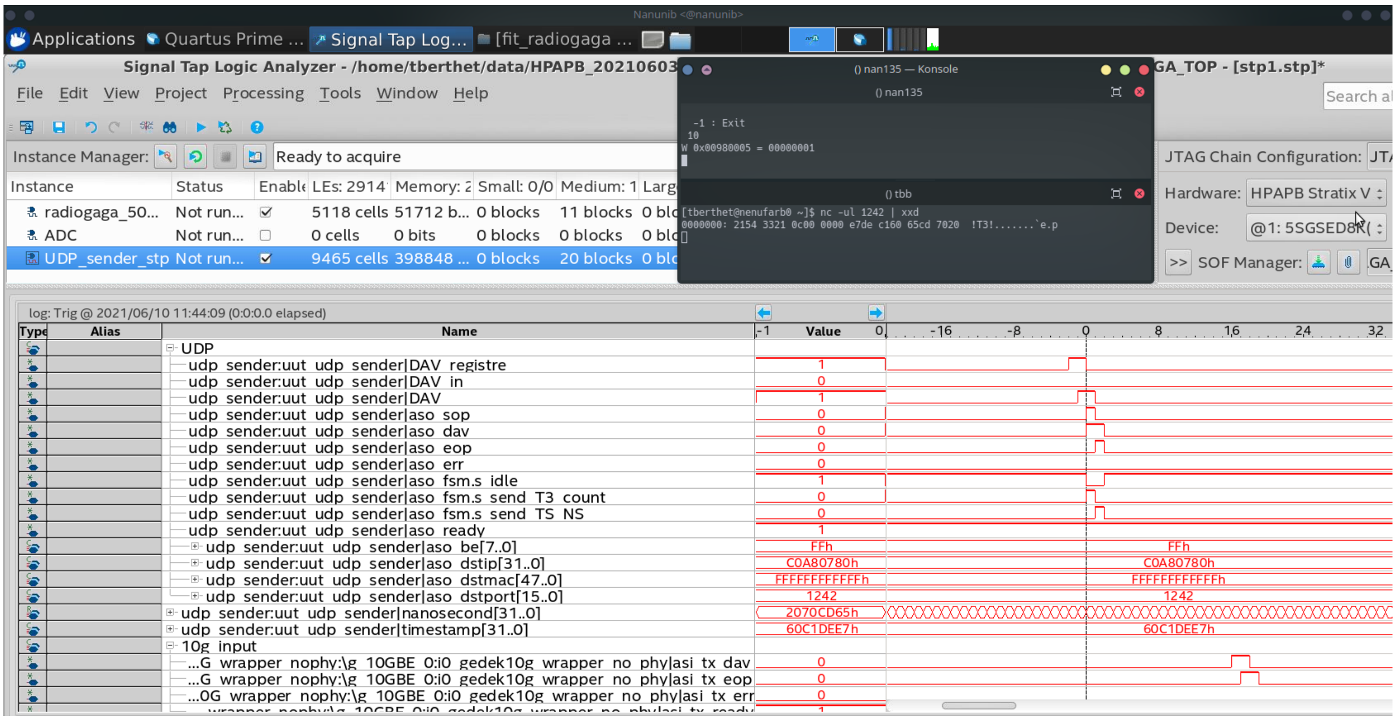


Figure 5 : Test bench of UDP sender with delay

Now the DAV is always at **1F9** but in the frame we see **159**. Because we want to delay of 32 sample.
0x1F9 - 32 * 5 = 0x159

3.2.4 Signal TAP



```
 -1 : Exit
10
W 0x00980005 = 00000001
```

```
[tberthet@nenufarb0 ~]$ nc -ul 1242 | xxd
0000000: 2154 3321 0c00 0000 e7de c160 65cd 7020  !T3!.......`e.p
```

log: Trig @ 2021/06/10 11:44:09 (0:0:0.0 elapsed)

Name	Value
UDP	
udp sender:uut udp sender|DAV registre	1
udp sender:uut udp sender|DAV in	0
udp sender:uut udp sender|DAV	1
udp sender:uut udp sender|aso sop	0
udp sender:uut udp sender|aso dav	0
udp sender:uut udp sender|aso eop	0
udp sender:uut udp sender|aso err	0
udp sender:uut udp sender|aso fsm.s idle	1
udp sender:uut udp sender|aso fsm.s send T3 count	0
udp sender:uut udp sender|aso fsm.s send TS NS	0
udp sender:uut udp sender|aso ready	1
udp sender:uut udp sender|aso be[7..0]	FFh
udp sender:uut udp sender|aso dstip[31..0]	C0A80780h
udp sender:uut udp sender|aso dstmac[47..0]	FFFFFFFFFFFFh
udp sender:uut udp sender|aso dstport[15..0]	1242
udp sender:uut udp sender|nanosecond[31..0]	2070CD65h
udp sender:uut udp sender|timestamp[31..0]	60C1DEE7h
10g input	
...G wrapper nophy:\g 10GBE 0:i0 gedek10g wrapper no phy|asi tx dav	0
...G wrapper nophy:\g 10GBE 0:i0 gedek10g wrapper no phy|asi tx eop	0
...0G wrapper nophy:\g 10GBE 0:i0 gedek10g wrapper no phy|asi tx err	0

Figure 6 : Signal TAP of UDP sender

In the register interface of UDP sender is possible to send a DAV manually. Is very useful to test if the 10G configuration is working well.
So on the figure 6 we see in the terminal we write **'1'** in at the address 16#980005#. So after this the frame is built and put on the 10G interface. By inverting the bytes 2 by 2 we can read timestamp and nanosecond correctly.

3.3 Trigger

TODO

4. Software

4.1 Configuration files for RadioGaga

4.1.1 Presentation of files

This page will describe how configuration files works for RadioGaGa.
There are 2 types of files:

boot_configuration_board_X.conf

Contains all command to configure radiogaga module at the boot of the service. This files should load stable commands to enable the same boot each times.

> ⚠ **Warning**
>
> Need to generate one file by board

This include :

- Load filter coefficients and configure shift.
- Apply a value of K and a shift for the trigger, not so low for prevent too much detection.
- Configure the register in SumX, `'0'` for the last board, `'1'` for others.
- UDP_sender :
  - enable or not to send frames to TBB
  - set the delay time on TS to delete the traitment time.
  - Configure IP, mac address and port of TBB

recursive_configuration_board_X.conf

Those files allow to reconfigure specifics registers at each new pointing .

> ⚠ **Warning**
>
> Need to generate one file by board

This include :

- Change coefficients and shift of filters.
- Set a new value to K and shift for the trigger.
- Allow to disable some MR in presum if there are some parasites or something else.

4.1.2 How to generate those files

To generate those files there is python script which take .yaml files to set

config_boot.yaml

```
# Input file to generate boot configurations files.
board:
    - 0:
        filter:
            - 1:
                - path: 22M_rej.conf
                - shift: 15
            - 2:
                - path: 36M_rej.conf
                - shift: 15
            - 3:
                - path: 22M_73M_band.conf
                - shift: 15

        trigger:
            - k: 200
            - shift : 18
etc..
```

config_recursive.yaml

```
# Input file to generate recursive configurations files.
board:
    - 0:
        filter:
            - 1:
                - path: 22M_rej.conf
                - shift: 15
            - 2:
                - path: 36M_rej.conf
                - shift: 15
            - 3:
                - path: 22M_73M_band.conf
                - shift: 15

        trigger:
            - k: 400
            - shift : 18
etc..
```

Then just need to run the appropriate python script

boot

```
python gen_boot_conf_file.py
```

recursive

```
python gen_recursive_conf_file.py
```

After running scripts. Configurations files are generated, they have the same format than `HPAPB_ADC.conf`

boot_configuration_board_0.conf

```
TODO
```

4.2 Boot sequence

This chapter will present step by step the configuration needed at the boot of the HPAPB board.
The firmware RadioGaGa is based on LANewBa, so there is a lot of similitude between both.

1. Ping

First of all when tartup is detected we need to ping the board on the 1G interface.
1G configuration :

- GEDEK_IP_ADDRESS : `192.168.6.128 + number_of_the_card`
- GEDEK_MACADDRESS : `00:07:ed:a8:06:80`
- GEDEK_UDP_AVM_PORT : `2271`
- GEDEK_UDP_TXPORT : `1271`

2. Read status

Read constant of the board (version of the firmware, temperature etc.) Sames addresses than LANewBa.

```
TODO
```

3. ADC and PLL configuration

There is exactly same on LANewBa, we need to run the file :
svn/NenuFar/trunk/recepteur_LANewBa/trunk/firmware/HPAPB/soft/HPAPB_LMK.conf

4. Read debug info

Here we need to read debug info, with monitor such as LANewBa.

```
TODO
```

5. Config RadioGaGa modules

To configure RadioGaGa modules, we need to run the `boot_configuration_board_X.conf` for each board. For more information read this page : Software/Configuration files

END of the boot configuration

4.3 Running mode

4.3.1 Monitor

When the board is booted correctly, it needed to read in some registers for safety of the board. For this we read in registers of monitor modules.
To this it's exactly the same than LANewBa, we need to :

```
TODO
```

4.3.2 Recursive reconfiguration

Like is explained here : Software/Configuration files

5. Divers

5.1 PyFDA

TODO