

---

# 2010 CASPER Workshop

## Tutorial 4: Wideband Poco

Author: W. New

Expected completion time: 2hrs



SKA SOUTH AFRICA  
SQUARE KILOMETRE ARRAY

### Contents:

- Introduction
- Setup
- Background
- Creating Your Design
- Software

## 1 Introduction

In this tutorial, you will create a simple Simulink design which uses the iADC board on ROACH and the CASPER DSP blockset to process a wideband signal, channelize it and output the UV visibilities through ROACH's PPC.

By this stage, it is expected that you have completed tutorials 1 and 2 and are reasonably comfortable with Simulink and basic Python. We will focus here on higher-level design concepts, and will provide you with low-level detail preimplemented.

## 2 Setup

The lab at the workshop is preconfigured with the CASPER libraries, Matlab and Xilinx tools. Start Matlab.

## 3 Background

Some of this design is similar to that of the previous tutorial, The Wideband Spectrometer. So completion of Tutorial 3 is recommended.

### 3.1 The Correlator

The correlator we will be designing is a 4 input correlator. It uses 2 inputs from each ADC, although you do not need to populate ROACH with the second ADC.

---

## 3.2 Antennae and Baselines

When doing correlation on a set of antennae we introduce the term baseline. A baseline is the product of the signal from two antennas. We calculate all baselines. For example, if we have 3 antennae, A, B and C, we need to perform correlation across each baseline, AB, AC and BC. We also need to do auto-correlations, which will give us the power in each signal. ie AA, BB, CC. We will see this implemented later.

## 3.3 Polarization

Dish type receivers are typically dual polarized (horizontal and vertical feeds). Each polarization is fed into separate ADC inputs. When correlating these antennae, we differentiate between full Stokes correlation or a half Stokes method. A full Stokes correlator does cross correlation between the different polarizations (ie for a given two antennas, A and B, it multiplies the horizontal feed from A with the vertical feed from B and vice-versa). A half stokes correlator only correlates like polarizations with each other, thereby halving the compute requirements.

Our correlator here performs all cross correlations and so can be thought of as a 2-input full Stokes correlator or as a four input single polarization correlator.

# 4 Creating Your Design

## 4.1 Create a new model:

Start Matlab and open Simulink (either by typing simulink on the Matlab command line, or by clicking the Simulink icon in the taskbar). Create a new model and add the *Xilinx System Generator* and *XSG core config* blocks as before in Tutorial 1.

### 4.1.1 System Generator and XSG Blocks



System  
Generator

By now you should have used these blocks a number of times. Pull the **System Generator** block into your design from the Xilinx Blockset menu under Basic Elements. The settings can be left on default.



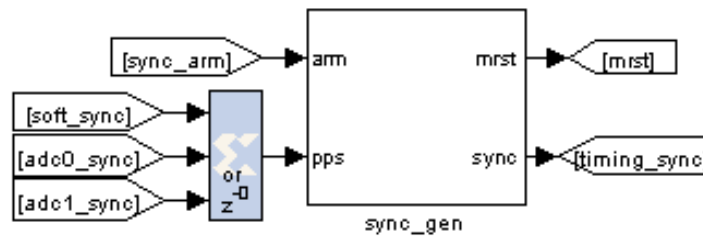
XSG core config

The **XSG** block can be found under the BEE\_XPS System Blockset. Set the Hardware platform to ROACH:sx95t, the Clock Source to adc0\_clk and the rest of the configuration as the default.

Make sure you have an ADC plugged into ZDOK0 to supply the FPGA's clock!

---

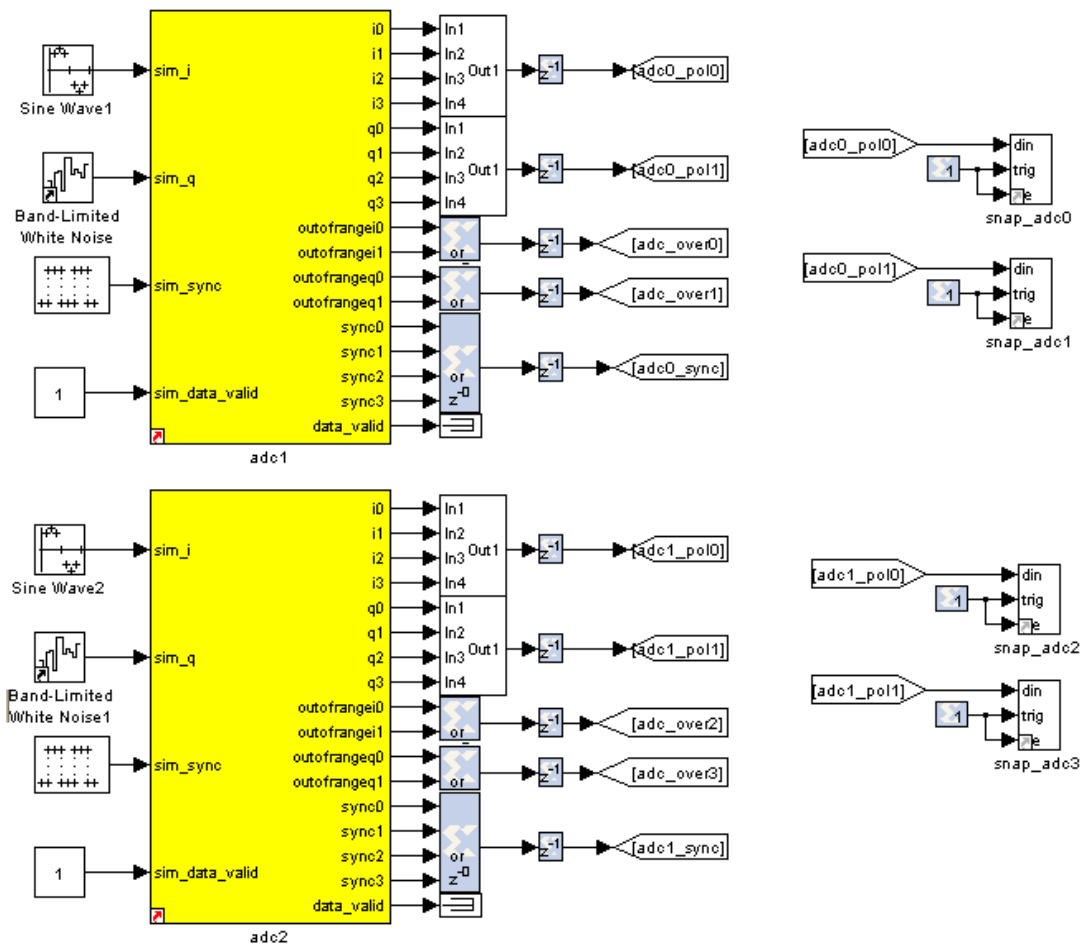
### 4.1.2 Sync Generator



The Sync Generator puts out a sync pulse which is used to synchronize the blocks in the design. See the CASPER memo on sync pulse generation for a detailed explanation and the iBOB iADC tutorial for an example on its basic use.

This sync generator is able to synchronize with an external trigger input. Typically we connect this to a GPS's 1pps output to allow the system to reset on a second boundary after a software arm and thus know precisely the time at which an accumulation was started. To do this you can input the 1pps signal into either ADCs' sync input.

### 4.1.3 ADCs

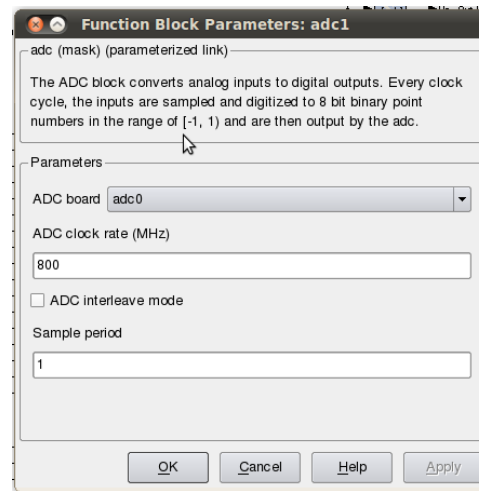


Connection of the ADCs is as in tutorial 3 except for the sync outputs. Here we OR all four outputs together to produce a sync pulse at one quarter the rate of the ADC's sample clock. This means, however, that our system can only be synchronized to within 3 ADC sample clocks.

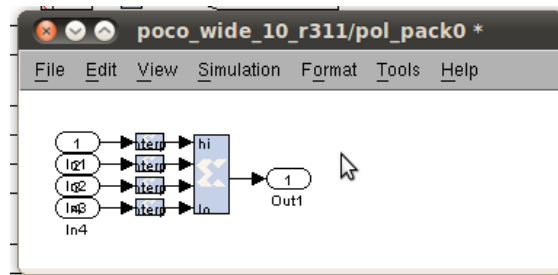
We will not use the 1pps in this tutorial although the design has the facility to do this hardware syncing.

---

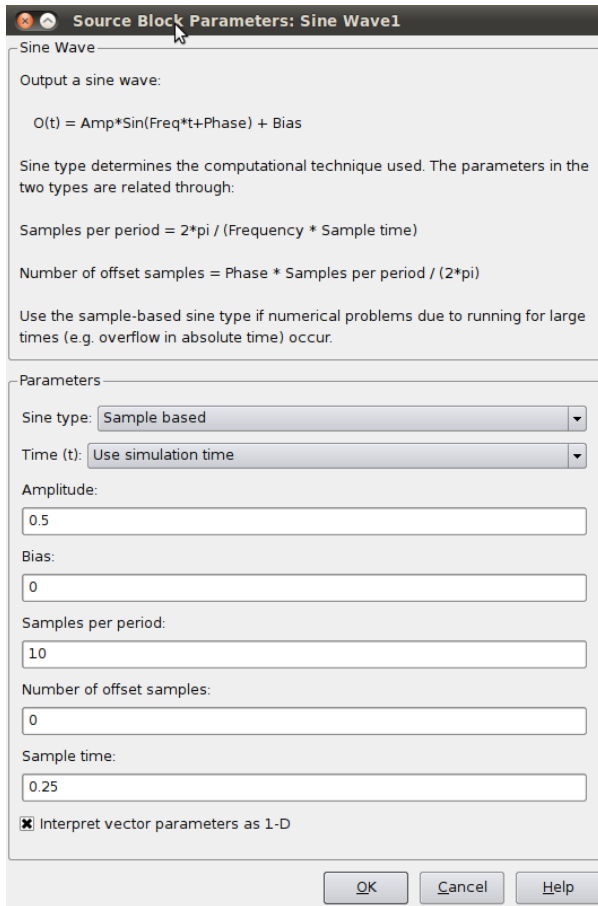
Set up the ADCs as follows and change the second ADC board's mask parameter to *adc1*...



To create the concat blocks, construct the subsystem shown below. The unnamed blocks are *Reinterpret* and *Concat* blocks.



For the purposes of simulation (and to satisfy Simulink's requirements that all inputs be connected), we need to put input signals into the ADCs. These blocks are pulse generators in the case of sync inputs and any analogue source for the RF inputs (noise, CW tones etc).



**Source Block Parameters: Sine Wave1**

Sine Wave

Output a sine wave:

$$O(t) = \text{Amp} * \sin(\text{Freq} * t + \text{Phase}) + \text{Bias}$$

Sine type determines the computational technique used. The parameters in the two types are related through:

$$\text{Samples per period} = 2 * \pi / (\text{Frequency} * \text{Sample time})$$
$$\text{Number of offset samples} = \text{Phase} * \text{Samples per period} / (2 * \pi)$$

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type: **Sample based**

Time (t): **Use simulation time**

Amplitude: 0.5

Bias: 0

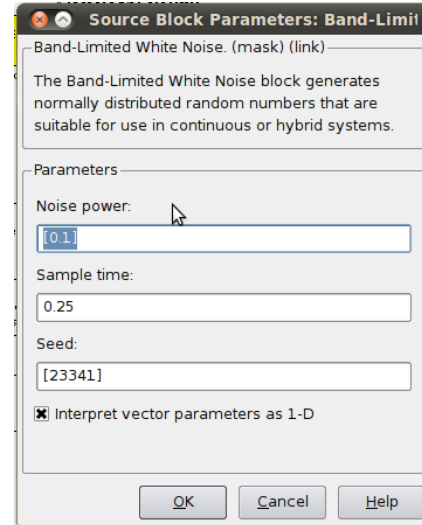
Samples per period: 10

Number of offset samples: 0

Sample time: 0.25

☒ Interpret vector parameters as 1-D

OK Cancel Help



**Source Block Parameters: Band-Limited White Noise**

Band-Limited White Noise. (mask) (link)

The Band-Limited White Noise block generates normally distributed random numbers that are suitable for use in continuous or hybrid systems.

Parameters

Noise power: 0.1

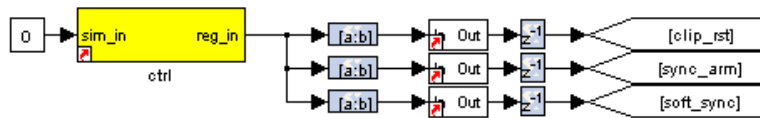
Sample time: 0.25

Seed: [23341]

☒ Interpret vector parameters as 1-D

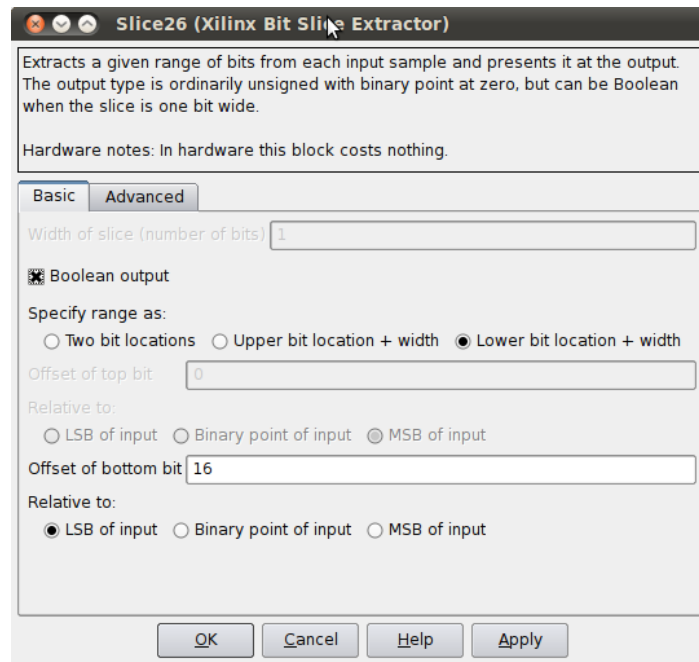
OK Cancel Help

#### 4.1.4 Control Register



This part of the Simulink design sets up a software register which can be configured in software to control the correlator. Set the yellow **software register's** IO direction as **from processor**. You can find it in the BEE\_XPS System blockset. The constant block input to this register is used only for simulation.

The output of the software register goes to three slice blocks, which will pull out the individual parameters for use with configuration. The first slice block (top) is setup as follows:



The slice block can be found under the Xilinx Blockset → Control Logic. The only change with the subsequent slice blocks is the Offset of the bottom bit. They are, from top to bottom, respectively, 16, 17 & 18.

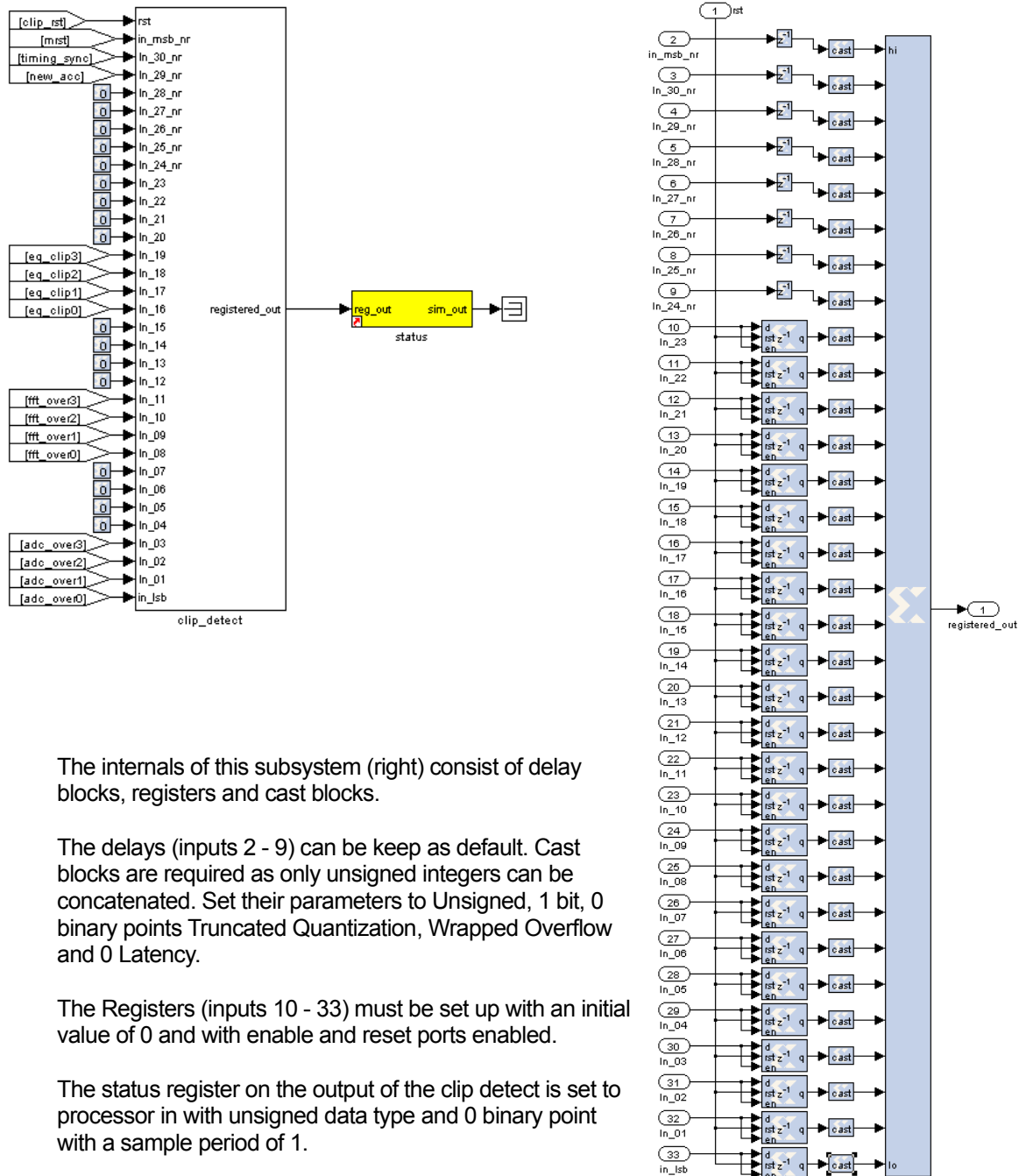
After each slice block we put a **posedge block**, this outputs true if a boolean input signal is true this clock and was false last clock. Found under CASPER DSP Blockset → Misc.

Next are the delay blocks. They can be left with their default settings and can be found under Xilinx Blockset → Common.

The **Goto** and **From** blocks can be found under Simulink → Signal Routing. Label them as in the block diagram above.

### 4.1.5 Clip Detect and status reporting

To detect and report signal saturation (clipping) to software, we will create a subsystem with latching inputs.



The internals of this subsystem (right) consist of delay blocks, registers and cast blocks.

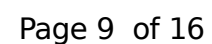
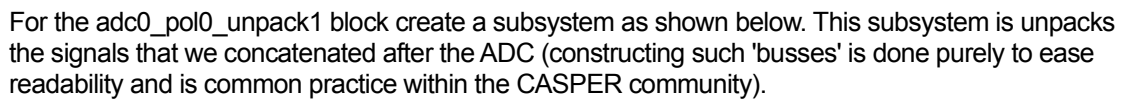
The delays (inputs 2 - 9) can be kept as default. Cast blocks are required as only unsigned integers can be concatenated. Set their parameters to Unsigned, 1 bit, 0 binary points Truncated Quantization, Wrapped Overflow and 0 Latency.

The Registers (inputs 10 - 33) must be set up with an initial value of 0 and with enable and reset ports enabled.

The status register on the output of the clip detect is set to processor in with unsigned data type and 0 binary point with a sample period of 1.



The PFB FIR, FFT and the Quantizer are the heart of this design, there is one set of each for the 4 outputs of the ADCs. This system consists of four copies of tut3.



---

Setup the slice blocks to return the original signals that we got from the ADCs.

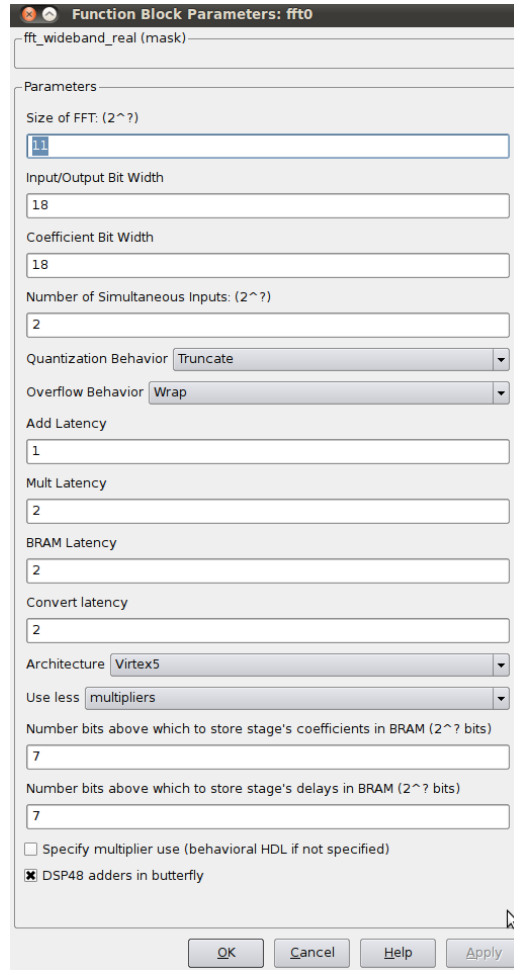
Configure the PFB\_FIR\_real blocks as shown below:

The screenshot shows the 'Function Block Parameters: pfb\_fir\_real0' dialog box. The 'Parameters' section is expanded, showing various configuration options. The 'Fold adds into DSPs' checkbox is checked. The 'Multiplier specification' is set to [2, 2].

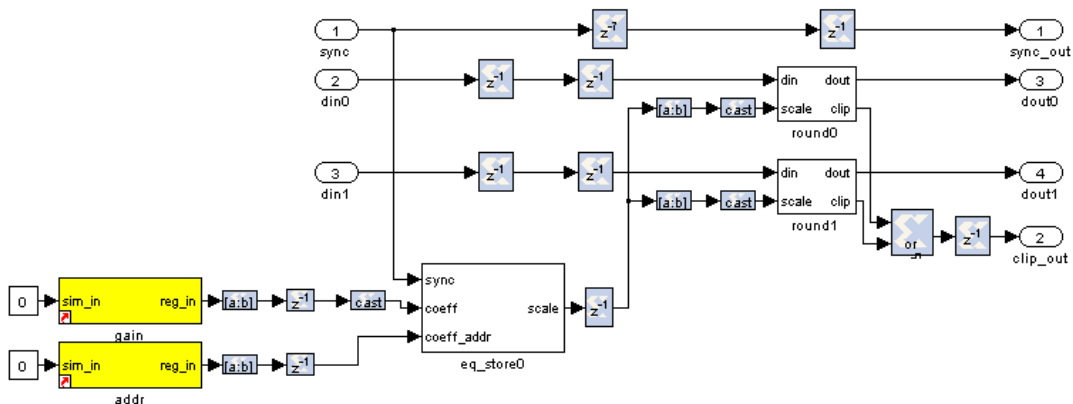
Parameter	Value
Size of PFB: (2 <sup>7</sup> pnts)	128
Total Number of Taps:	4
Windowing Function:	hamming
Number of Simultaneous Inputs: (2 <sup>?</sup> )	2
Make Biplex	0
Input Bitwidth:	8
Output Bitwidth:	18
Coefficient Bitwidth:	18
Use Distributed Memory for Coeffs	0
Add Latency	1
Mult Latency	2
BRAM Latency	2
Convert latency	1
Quantization Behavior	Truncate
Bin Width Scaling (normal=1)	1
Specify multiplier use (behavioral if not specified)	<input type="checkbox"/>
Multiplier specification (0=core, 1=embedded, 2=behavioural) (left=1st tap)	[2, 2]
Fold adds into DSPs	<input checked="" type="checkbox"/>

There is potential to overflow the first FFT stage if the input is periodic or signal levels are high as shifting inside the FFT is only performed after each butterfly stage calculation. For this reason, we recommend casting any inputs up to 18 bits with the binary point at position 17 (thus keeping the range of values -1 to 1), and then downshifting by 1 bit to place the signal in one less than the most significant bits.

The fft\_wide\_band\_real block should be configured as follows:



The Quantizer Subsystem is designed as seen below. The quantizer removes the bit growth that was introduced in the PFB and FFT. We can do this because we do not need the full dynamic range.



### 4.1.7 LEDs

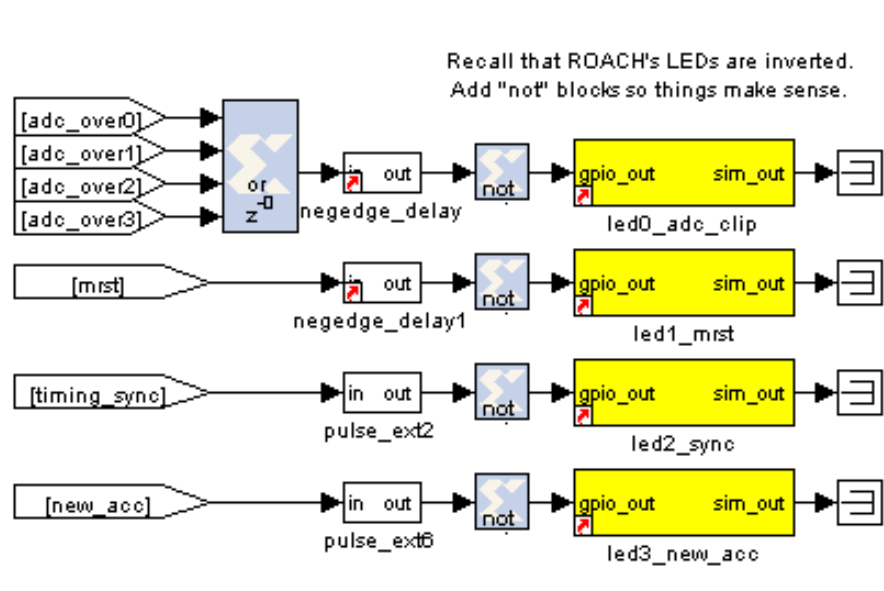
The following sections are more periphery to the design and will only be touched on. By now you should be comfortable putting the blocks together and be able to figure out many of the values and parameters. Also feel free to consult the reference design which sits in the tutorial 4 project directory or ask any questions of the tutorial helpers.

As a kind of debug output we can wire up the LEDs to certain signals. We light an LED with every sync pulse. This is a sort of heartbeat showing that the design is clocking and the FPGA is running.

We light an error LED in case any ADC overflows and another if the system is reset. The fourth LED gives a visual indication of when an accumulation is complete.

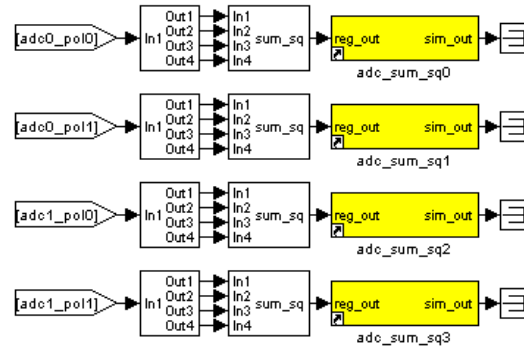
ROACH's LEDs are negative logic, so when the input to the yellow block is high, the LED is off. Since this is the opposite of what you'd normally expect, we invert the logic signals with a **NOT** gate.

Since the signals might be too short to light up an LED and for us to actually see it (consider the case where a single ADC sample overflows;  $1/800\text{MHz}$  is  $1.25\text{nS}$  – much too short for the human eye to see) we add a negedge delay block which delays the negative edge of a block, thereby extending the positive pulse. A length of  $2^{23}$  gives about a  $10\text{ms}$  pulse.



### 4.1.8 ADC RMS

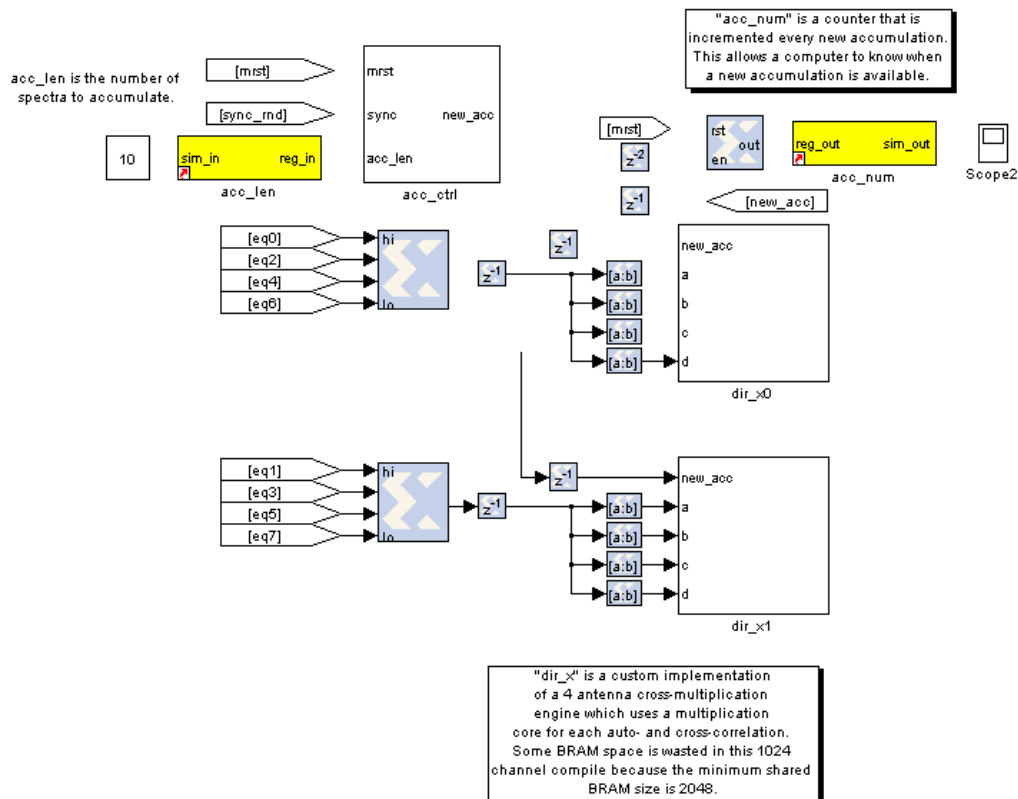
These blocks calculate the RMS values of the ADCs' input signals. We subsample the input stream by a factor of four and do a pseudo random selection of the parallel inputs to prevent false reporting of repetitive signals. This subsampled stream is squared and accumulated for  $2^{16}$  samples.



#### 4.1.9 The MAC operation

The multiply and accumulate is performed in the dir\_x (direct-x) blocks, so named because baselines are calculated directly, in parallel (as opposed to the packetised correlators' X engines which process serially).

Two sets are used, one for the even channels and another for the odd channels. Accumulation for each baseline takes place in BRAM using the same simple vector accumulator used in tut3.



---

## CONTROL:

design starts by itself. Only control register is for resetting counters and optionally syncing to external signal.

Sync LED provides a “heartbeat” signal to instantly see if your design is clocked sensibly.

New accumulation LED gives a visual indication of data rates and dump times.

Accumulation counter provides simple mechanism for checking if a new spectrum output is available. (poll and compare to last value)

## 5 Software

The python scripts are located in the project directory. We first need to run `poco_init.py` to program the FPGA and configure the design. Then we can run either the auto or the cross correlations scripts (`plot_poco_auto.py` and `plot_poco_cross.py`).

### `poco_init.py`

```
print('Connecting to server %s on port %i...'%(roach,katcp_port)),
      fpga = corr.katcp_wrapper.FpgaClient(roach, katcp_port,
timeout=10,logger=logger)
      time.sleep(1)

      if fpga.is_connected():
          print 'ok\n'
      else:
          print 'ERROR connecting to server %s on port %i.\n'%(
(roach,katcp_port)
          exit_fail()

      print '-----'
      print 'Programming FPGA...',
      if not opts.skip:
          fpga.progdev(boffile)
          print 'done'
      else:
          print 'Skipped.'

      print 'Configuring fft_shift...',
      fpga.write_int('fft_shift',(2**32)-1)
      print 'done'

      print 'Configuring accumulation period...',
      fpga.write_int('acc_len',opts.acc_len)
      print 'done'

      print 'Resetting board, software triggering and resetting error
counters...',
      fpga.write_int('ctrl',1<<17) #arm
      fpga.write_int('ctrl',1<<18) #software trigger
      fpga.write_int('ctrl',0)
      fpga.write_int('ctrl',1<<18) #issue a second trigger
      print 'done'
```

---

---

In previous tutorials you will probably have seen very similar code to the code above. This initiates the katcp wrapper named `fpga` which manages the interface between the software and the hardware. `fpga.progdev` programs the boffile onto the FPGA and `fpga.write_int` writes to a register.

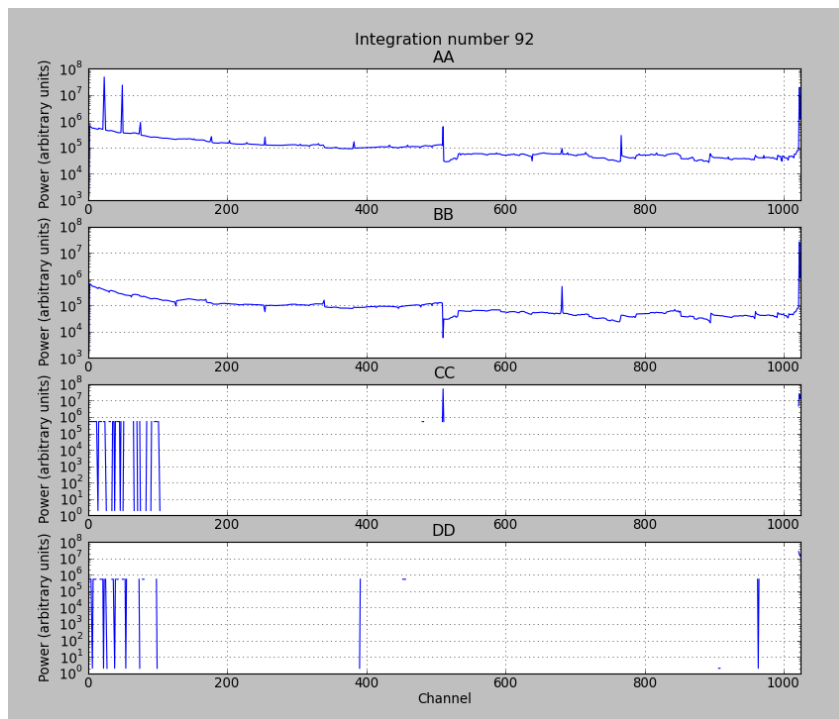
### **poco\_adc\_amplitudes.py**

This script outputs in the amplitudes (or power) of each signal as well as the bits used. It updates itself ever second or so.

```
ADC amplitudes
-----
ADC0 input I: 0.060 (3.93 bits used)
ADC0 input Q: 0.006 (0.70 bits used)
ADC1 input I: 0.622 (7.32 bits used)
ADC1 input Q: 0.709 (7.50 bits used)
-----
```

### **poco\_plot\_auto.py**

This script grabs auto-correlations from the brams and plots them. Since there are 4 inputs, 2 for each ADC there are 4 plots. Some plots will be random if there is no noise source or tone being inputted into ADC. I.e plots 3 and 4.



---

### poco\_plot\_cross.py

This script grabs cross-correlations from the brams and plots them. This plot shows the cross-correlation of AB.

