2012 CASPER Workshop

Tutorial 3: Wideband Spectrometer

Author: **Jason Manley and Danny Price**

(Updated 2012: Jack Hickish)

Version: **August 2012, v1.2**

Expected completion time: 2hrs

Contents:

1. Introduction
2. Spectrometer basics

PART ONE
3. Simulink design overview
4. Detailed blockumentation

PART TWO
5. Hardware configuration
6. The spectrometer.py script
7. iPython walkthrough
8. spectrometer.py notes

Introduction

A spectrometer is something that takes a signal in the *time* domain and converts it to the *frequency* domain. In digital systems, this is generally achieved by utilising the FFT (Fast Fourier Transform) algorithm. However, with a little bit more effort, the signal to noise performance can be increased greatly by using a Polyphase Filter Bank (PFB) based approach.

When designing a spectrometer for astronomical applications, it's important to consider the science case behind it. For example, pulsar timing searches will need a spectrometer which can dump spectra on short timescales, so the rate of change of the spectra can be observed. In contrast, a deep field HI survey will accumulate multiple spectra to increase the signal to noise ratio. It's also important to note that “bigger isn't always better”; the higher your spectral and time resolution are, the more data your computer (and scientist on the other end) will have to deal with. For now, let's skip the science case and familiarize ourselves with an example spectrometer.

In this tutorial, we will build a 400MHz bandwidth, 2048 channel, PFB based spectrometer on the ROACH. You'll need to have done Tutorial 1, 2 and the iADC tutorial[1]. You should also have installed python, iPython, corr, aipy, numpy and pylab. As far as hardware goes, you'll need:

- a ROACH board;
- an iADC, which should be connected to *ZDOK0* on the ROACH; and
- a clock source, such as a signal generator, which should be connected to *clk_i* on the iADC.

You'll need to be familiar with the basic concepts of sampling, have a solid understanding of what a Fourier transform is, and have a vague idea of what a FFT is. A good reference is Smith's free online DSP guide (at *http://www.dspguide.com/*); in particular, have a read of chapters 3, 8 and 12.

Spectrometer Basics

When designing a spectrometer there are a few main parameters of note:

- *Bandwidth:* The width of your frequency spectrum, in Hz. This depends on the sampling rate; for complex sampled data this is equivalent to:

$$BW = sampling\ rate = \frac{1}{sampling\ \ period}.$$

In contrast, for real or Nyquist sampled data the rate is half this:

$$BW = \frac{sampling\ rate}{2} = \frac{1}{2 \times sampling\ \ period},$$

as two samples are required to reconstruct a given waveform[2].

- *Frequency resolution:* The frequency resolution of a spectrometer, *Δf*, is given by

$$\Delta f = \frac{BW}{no.\ channels},$$

and is the width of each frequency bin. Correspondingly, *Δf* is a measure of how precise you can measure a frequency.

- *Time resolution:* Time resolution is simply the spectral dump rate of your instrument. We generally accumulate multiple spectra to average out noise; the more accumulations we do, the lower the time resolution. For looking at short timescale events, such as pulsar bursts, higher time resolution is necessary; conversely, if we want to look at a weak HI signal, a long accumulation time is required, so time resolution is less important.

1 While the iADC tutorial is for an iBOB, you can follow the iADC tutorial on the ROACH by simply changing the XSG core config from 'iBOB' to 'ROACH'. http://casper.berkeley.edu/wiki/Tutorials#IBOB

2 For an introduction to sampling, read http://www.dspguide.com/ch3.htm

PART ONE

Simulink / CASPER Toolflow

Simulink Design Overview

If you're reading this, then you've already managed to find all the tutorial files. Jason has gone to great effort to create an easy to follow simulink model that compiles and works. By now, I presume you can open the model file and have a vague idea of what's happening.

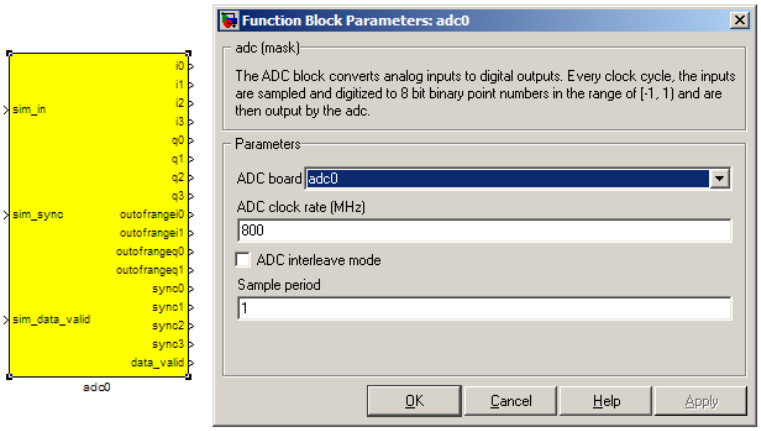
The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some "blockumentation" in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- The all important **Xilinx token** is placed to allow System Generator to be called to compile the design.
- In the **MSSGE block**, the hardware type is set to 'ROACH:sx95t" and clock rate is specified as 200MHz.
- The input signal is digitised by the **ADC**, resulting in four parallel time samples of 8 bits each clock cycle. The ADC runs at 800MHz, which gives a 400MHz nyquist sampled spectrum. The output range is a signed number in the range -1 to +1 (ie 7 bits after the decimal point). This is expressed as fix_8_7.
- The four parallel time samples pass through the **pfb_fir_real** and **fft_wideband_real** blocks, which together constitute a polyphase filter bank. We've selected 2^{12}=4096 points, so we'll have a 2^{11}=2048 channel filter bank.
- You may notice Xilinx delay blocks dotted all over the design. It's common practice to add these into the design as it makes it easier to fit the design into the logic of the FPGA. It consumes more resources, but eases signal timing-induced placement restrictions.
- The real and imaginary (sine and cosine value) components of the FFT are plugged into **power** blocks, to convert from complex values to real power values by squaring. They are also scaled by a gain factor before being quantised...
- These power values are then requantized by the **quant0** block, from 36.34 bits to 6.5 bits, in preparation for accumulation. This is done to limit bit growth.
- The requantized signals then enter the vector accumulators, **vacc0** and **vacc1**, which are **simple_bram_vacc** 32 bit vector accumulators. Accumulation length is controlled by the **acc_cntrl** block.
- The accumulated signal is then fed into software registers, **odd** and **even**.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

ADC

http://casper.berkeley.edu/wiki/Adc



The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter. In Simulink, the ADC daughter board is represented by a yellow block. Work through the "iADC tutorial" if you're not familiar with the iADC card.

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 8 bit binary point numbers in the range of -1 to 1 and are then output by the ADC. This is achieved through the use of two's-compliment representation with the binary point placed after the seven least significant bits. This means we can represent numbers from -128/128 through to 127/128 including the number 0. Simulink represents such numbers with a fix_8_7 moniker.

ADCs often internally bias themselves to halfway between 0 and -1 (ie the center of the range of representable values). This means that you'd typically see the output of an ADC toggling between zero and -1 when there's no input. It also means that unless otherwise calibrated, an ADC will have a negative DC offset.

The ADC has to be clocked to four times that of the FPGA clock. In this design the ADC is clocked to 800MHz, so the ROACH will be clocked to 200MHz[3]. This gives us a bandwidth of 400MHz, as Nyquist sampling requires two samples (or more) each second.

INPUTS

Port	Description
sim_in	Input for simulated data. It's useful to connect up a simulink source, such as "band-limited white noise" or a sine wave.
sim_sync	Simulated sync pulse input. In this design, we've connected up a constant with value '1'.
sim_data_valid	Can be set to either 0 (not valid) or 1 (valid).

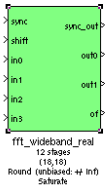
OUTPUTS

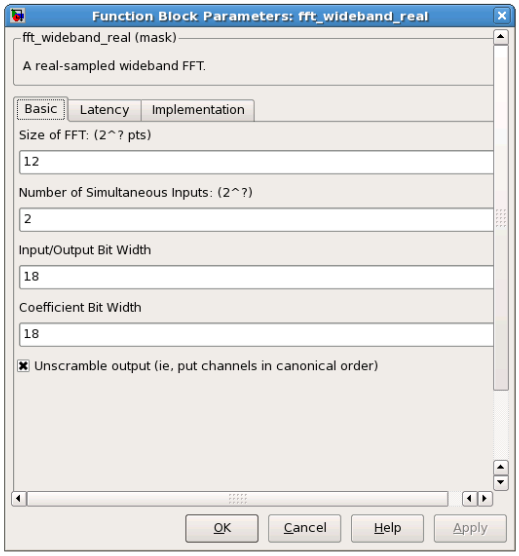
The ADC outputs two main signals: *i* and *q*, which correspond to the coaxial inputs of the ADC board. In this tutorial, we'll only be using input *i.* As the ADC runs at 4x the FPGA rate, there are four parallel time sampled outputs: *i0, i1, i2* and *i3.* As mentioned before, these outputs are 8.7 bit.

3 In the XSG core config block, we have selected *adc0_clk* as our clock, which is provided by the ADC daughter card plugged into connector ZDOK0.

PFB_FIR_REAL

http://casper.berkeley.edu/wiki/Pfb_fir_real





There are two main blocks required for a polyphase filter bank. The first is the **pfb_fir_real** block, which divides the signal into parallel 'taps' then applies finite impulse response filters (FIR). The output of this block is still a time-domain signal. When combined with the **FFT_wideband_real** block, this constitutes a polyphase filterbank.

INPUTS/OUTPUTS

Port	Data Type	Description
sync	bool	A sync pulse should be connected here (see iADC tutorial).
pol1_in1 pol1_in2 pol1_in3 pol1_in4	inherited	The (real) time-domain stream(s).

As the ADC has four parallel time sampled outputs: i0, i1, i2 and i3, we need four parallel inputs for this PFB implementation.

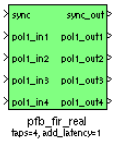
PARAMETERS

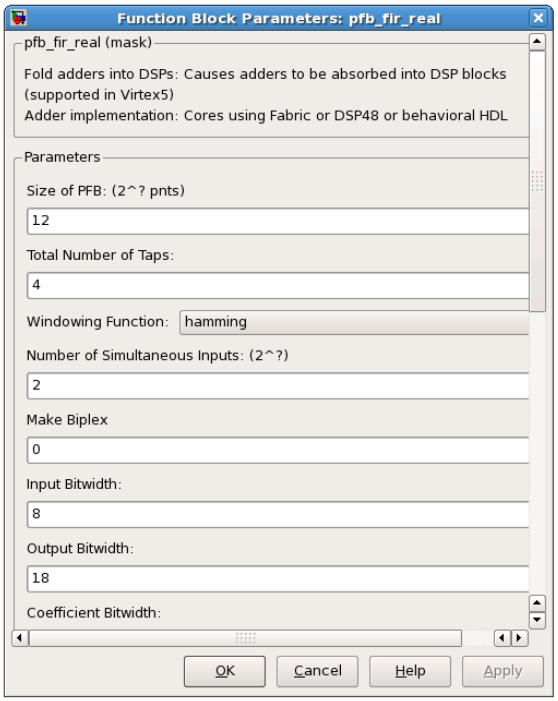
Parameter	Description
Size of PFB	How many points the FFT will have. The number of frequency channels will be half this. We've selected $2^{12} = 4096$ points, so we'll have a $2^{11} = 2048$ channel filter bank.

Number of taps	The number of taps in the PFB FIR filter. Each tap uses 2 real multiplier cores and requires buffering the real and imaginary streams for 2*PFBSize samples. Generally, more taps means less inter-channel spectral leakage, but more logic is used. There are diminishing returns after about 8 taps or so.
Windowing function	Which windowing function to use (this allows trading passband ripple for steepness of rolloff, etc). Hamming is the default and best for most purposes.
Number of Simultaneous Inputs (2?)	The number of parallel time samples which are presented to the FFT core each clock. The number of output ports are set to this same value. We have four inputs from the ADC, so set this to 2.
Make biplex	0 (not making it biplex) is default. Double up the inputs to match with a biplex FFT.
Input bitwidth.	The number of bits in each real and imaginary sample input to the PFB. The ADC outputs 8.7 bit data, so the input bitwidth should be set to 8 in our design.
Output bitwidth	The number of bits in each real and imaginary sample output from the PFB. This should match the bit width in the FFT that follows. 18 bits is recommended for the ROACH (18x25 multipliers) and iBOB/BEE2 (18x18 multipliers).
Coefficient bitwidth	The number of bits in each coefficient. This is usually chosen to be less than or equal to the input bit width.
Use dist mem for coeffients	Store the FIR coefficients in distributed memory (if = 1). Otherwise, BRAMs are used to hold the coefficients. 0 (not using distributed memory) is default
Add/Mult/BRAM/Convert Latency	These values set the number of clock cycles taken by various processes in the filter. There's normally no reason to change this unless you're having troubles fitting the design into the fabric.
Quantization Behaviour	Specifies the rounding behaviour used at the end of each butterfly computation to return to the number of bits specified above. Rounding is strongly suggested to avoid artifacts.
Bin Width Scaling	PFBs give enhanced control over the width of frequency channels. By adjusting this parameter, you can scale bins to be wider (for values > 1) or narrower (for values < 1).
Multiplier specification	Specifies what type of resources are used by the various multiplications required by the filter.
Fold adders into DSPs	If this option is checked, adding operations will be combined into the FPGAs DSP cores, which have the both multiplying and adding capabilities.
Adder implementation	Adders not folded into DSPs can be implemented either using fabric resources (i.e. registers and LUTs in slices) or using DSP cores. Here you get to choose which is used. Choosing a behavioural implementation will allow the compiler to choose whichever implementation it thinks is best.
Share coefficients between polarisations	Where the pfb block is simultaneously processing more than one polarization, you can save RAM by using the same set of coefficients for each stream. This may, however, make the timing performance of your design worse.

FFT_WIDEBAND_REAL

http://casper.berkeley.edu/wiki/Fft_wideband_real





The **FFT_wideband_real** block is the most important part of the design to understand. The cool green of the FFT block hides the complex and confusing FFT butterfly biplex algorithms that are under the hood. You do need to have a working knowledge of it though, so I recommend reading Chapter 8 and Chapter 12 of Smith's free online DSP guide (at *http://www.dspguide.com/*).

Parts of the documentation below are taken from the documentation by Aaron Parsons and Andrew Martens on the CASPER wiki (at https://casper.berkeley.edu/wiki/Fft_wideband_real).

INPUTS/OUTPUTS

Port	Description
sync	Like many of the blocks, the FFT needs a heartbeat to keep it sync'd
shift	Sets the shifting schedule through the FFT. Bit 0 specifies the behavior of stage 0, bit 1 of stage 1, and so on. If a stage is set to shift (with bit = 1), then every sample is divided by 2 at the output of that stage. In this design, we've set Shift to $2^{13-1}-1$, which will shift the data by 1 on every stage to prevent overflows.

in0 in1 in2 in3	Four inputs for the parallel data streams coming from the ADC, through the pfb_fir_real filter block, and into here. Just connect them up.
out0 out1	The FFT produces two signals, the real part (out0, cosine wave values) and the imaginary part (out1, sine wave values). Following the lines you'll see that these two inputs end up in an “odd” and “even” software register. This is then interleaved in the spectrometer.py script to form a complete spectrum. Data is output in normal frequency order, meaning that channel 0 (corresponding to DC) is output first, followed by channel 1, on up to channel $2^{N-1} - 1$.

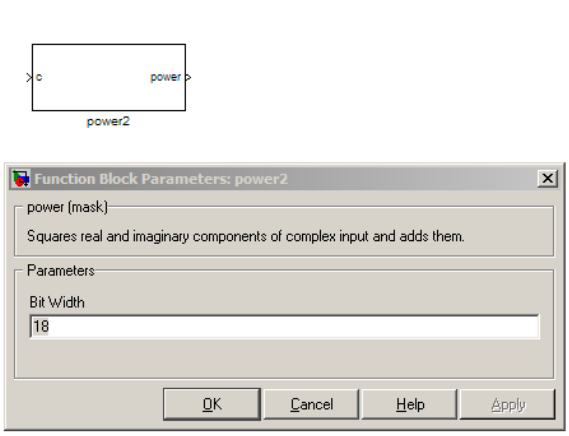
PARAMETERS

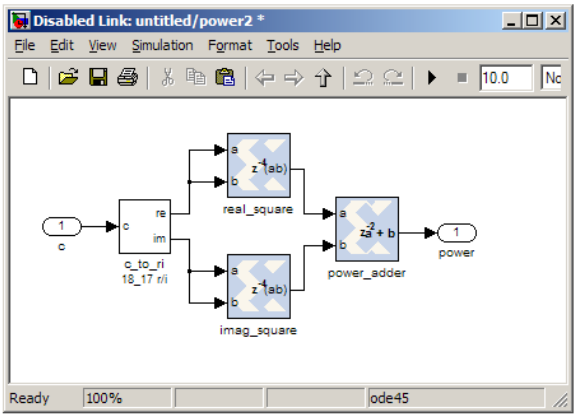
Parameter	Description
Size of FFT	How many points the FFT will have. The number of channels will be half this. We've selected $2^{12} = 4096$points, so we'll have a $2^{11} = 2048$ channel filter bank. This should match up with the pfb_fir block.
Input/output bitwidth	The number of bits in each real and imaginary sample as they are carried through the FFT. Each FFT stage will round numbers back down to this number of bits after performing a butterfly computation. This has to match what the pfb_fir is throwing out. The default is 18 so this shouldn't need to be changed.
Coefficient bitwidth	The amount of bits for each coefficient. 18 is default.
Number of simultaneous inputs	The number of parallel time samples which are presented to the FFT core each clock. We have $2^2 = 4$ parallel data streams, so this should be set to 2.
Unscramble output	Some reordering is required to make sure the frequency channels are output in canonical frequency order. If you're absolutely desperate to save as much RAM and logic as possible you can disable this processing, but you'll have to make sure you account for the scrambling of the channels in your downstream software. For now, because our design will comfortably fit on the FPGA, leave the unscramble option checked.
Overflow Behavior	Indicates the behavior of the FFT core when the value of a sample exceeds what can be expressed in the specified bit width. Here we're going to use Wrap, since Saturate will not make overflow corruption better behaved.
Add Latency	Latency through adders in the FFT. Set this to 2.
Mult Latency	Latency through multipliers in the FFT. Set this to 3.
BRAM Latency	Latency through BRAM in the FFT. Set this to 2.
Convert Latency	Latency through blocks used to reduce bit widths after twiddle and butterfly stages. Set this to 1.

Input Latency	Here you can register your input data streams in case you run into timing issues. Leave this set to 0.
Latency between biplexes and fft_direct	Here you can add optional register stages between the two major processing blocks in the FFT. These can help a failing design meet timing. For this tutorial, you should be able to compile the design with this parameter set to 0.
Architecture	Set to Virtex5, the architecture of the FPGA on the ROACH. This changes some of the internal logic to better optimise for the DSP slices. If you were using an older iBOB board, you would need to set this to Virtex2Pro.
Use less	This affects the implementation of complex multiplication in the FFT, so that they either use fewer multipliers or less logic/adders. For the complex multipliers in the FFT, you can use 4 multipliers and 2 adders, or 3 multipliers and a bunch or adders. So you can trade-off DSP slices for logic or vice-versa. Set this to Multipliers.
Number of bits above which to store stage's coefficients in BRAM	Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. By changing this, you can bias your design to use more BRAM or more logic. We're going to set this to 8.
Number of bits above which to store stage's delays in BRAM	Determines the threshold at which the twiddle coefficients in a stage are stored in BRAM. Below this threshold distributed RAM is used. Set this to 9.
Multiplier Implementation	Determines how multipliers are implemented in the twiddle function at each stage. Using behavioral HDL allows adders following the multiplier to be folded into the DSP48Es in Virtex5 architectures. Other options choose multiplier cores which allows quicker compile time. You can enter an array of values allowing exact specification of how multipliers are implemented at each stage. Set this to 1, to use embedded multipliers for all FFT stages.
Hardcode shift schedule	If you wish to save logic, at the expense of being able to dynamically specify your shifting regime using the block's "shift" input, you can check this box. Leave it unchecked for this tutorial.
Use DSP48's for adders	The butterfly operation at each stage consists of two adders and two subtracters that can be implemented using DSP48 units instead of logic. Leave this unchecked.

POWER

http://casper.berkeley.edu/wiki/Power





The **power** block computes the power of a complex number. The power block typically has a latency of 5 and will compute the power of its input by taking the sum of the squares of its real and imaginary components. The power block is written by Aaron Parsons and online documentation is by Ben Blackman.

In our design, there are two power blocks, which compute the power of the odd and even outputs of the FFT. The output of the block is 36.34 bits; the next stage of the design re-quantizes this down to a lower bitrate.

INPUTS/OUTPUTS

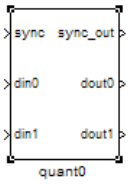
Port	Direction	Data Type	Description
c	IN	2*BitWidth Fixed point	A complex number whose higher BitWidth bits are its real part and lower BitWidth bits are its imaginary part.
power	OUT	UFix_(2*BitWidth)_(2*Bit Width-1)	The computed power of the input complex number.

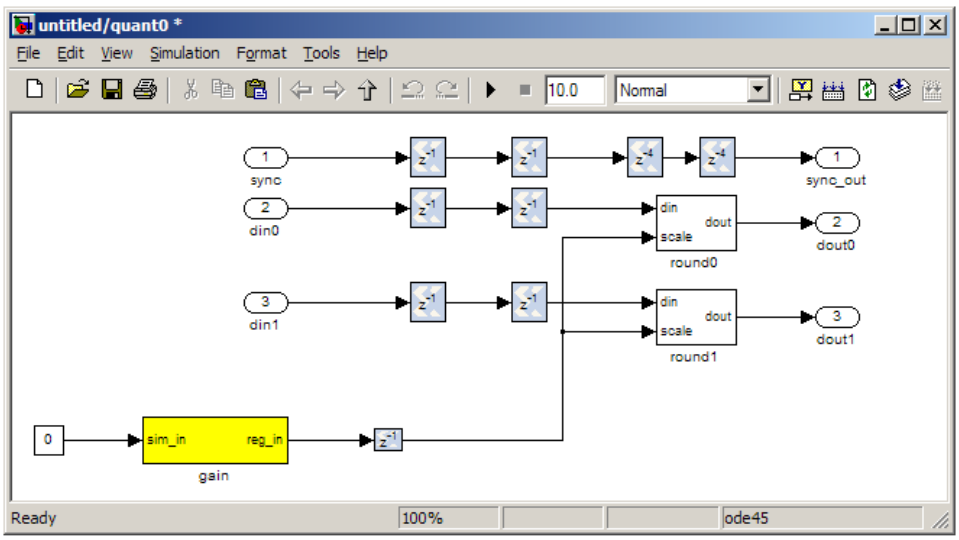
PARAMETERS

Parameter	Description
Bit Width	The number of bits in its input.

QUANT

no documentation online





The **quant0** was written by Jason Manley for this tutorial and is not part of the CASPER blockset. The block re-quantizes from 36.34 bits to 6.5 unsigned bits, in preparation for accumulation by the 32 bit **bram_vacc** block. This block also adds gain control, via a software register. The *spectrometer.py* script sets this gain control. You would not need to re-quantize if you used a larger vacc block, such as the 64bit one, but it's illustrative to see a simple example of re-quantization, so it's in the design anyway.

Note that the *sync_out* port is connected to a block, **acc_cntrl**, which provides accumulation control.

PARAMETERS

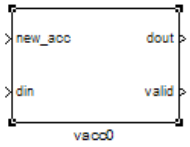
None.

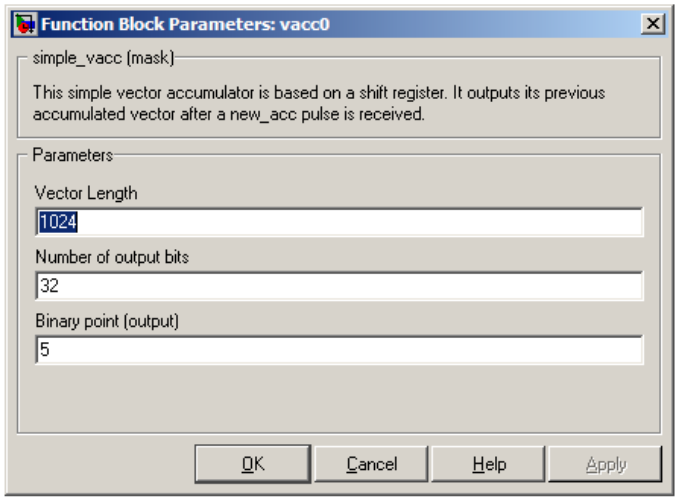
INPUTS/OUTPUTS

Port	Description
Sync	Input/output for the sync heartbeat pulse.
din0 din1	Data inputs – odd is connected to din0 and even is connected to din1. In our design, data in is 36.34 bits.
dout0 dout1	Data outputs. In this design, the quant0 block requantizes from the 36.34 input to 6.5 bits, so the output on both of these ports is 6.5 unsigned bits.

SIMPLE_BRAM_VACC

no documentation online





The **simple_bram_vacc** block is used in this design for vector accumulation. Vector growth is approximately 28 bits each second, so if you wanted a really long accumulation (say a few hours), you'd have to use a block such as the **qdr_vacc** or **dram_vacc**. As the name suggests, the **simple_bram_vacc** is simpler so it is fine for this demo spectrometer.

The FFT block outputs 1024 cosine values (odd) and 1024 sine values, making 2048 values in total. We have two of these bram vacc's in the design, one for the odd and one for the even frequency channels. The vector length is thus set to 1024 on both.

PARAMETERS

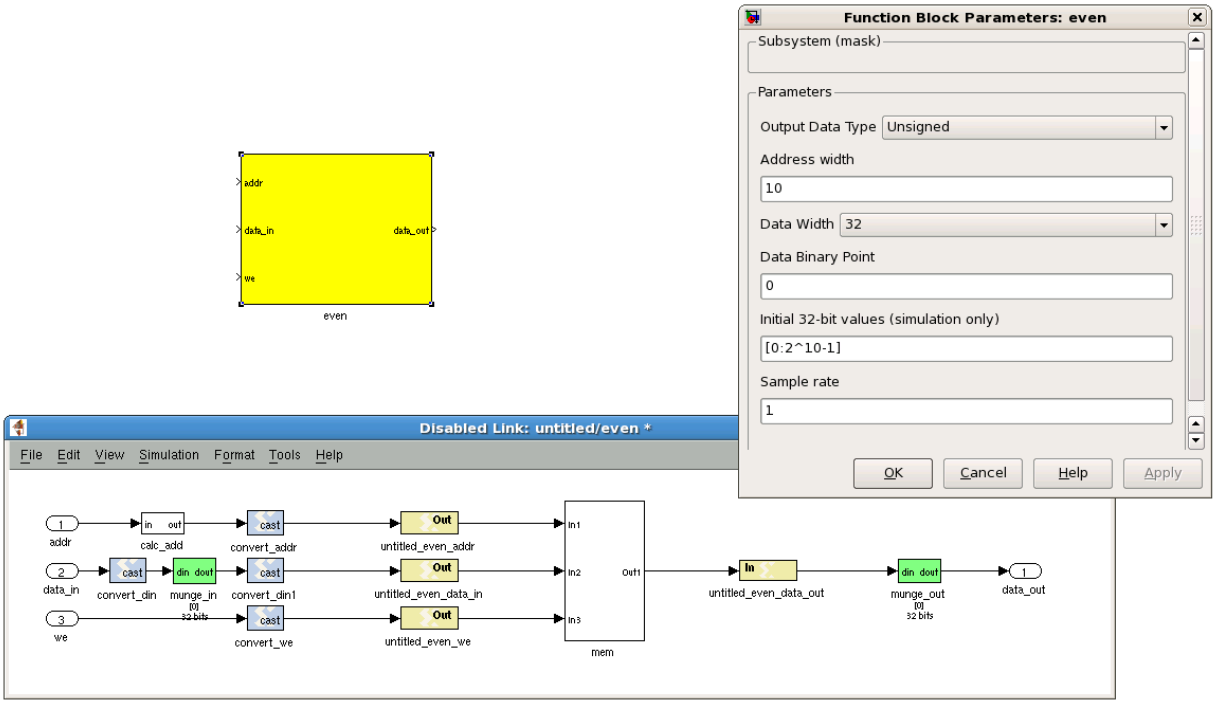
Parameter	Description
Vector length	The length of the input/output vector. The FFT block produces two streams of 1024 length (odd and even values), so we set this to 1024.
no. output bits	As there is bit growth due to accumulation, we need to set this higher than the input bits. The input is 6.5 from the quant0 block, we have set this to 32 bits. Note: We could set this to 64 bits and skip the quant block.
Binary point (output)	Since we are accumulating 6.5 values there should be 5 bits below the binary point of the output, so set this to 5.

INPUTS/OUTPUTS

Port	Description
new_acc	A boolean pulse should be sent to this port to signal a new accumulation. We can't directly use the sync pulse, otherwise this would reset after each spectrum. So, Jason has connected this to acc_cntrl, a block which allows us to set the accumulation period.
din/dout	Data input and output. The output depends on the no. output bits parameter.
Valid	The output of this block will only be valid when it has finished accumulating (signalled by a boolean pulse sent to new_acc). This will output a boolean 1 while the vector is being output, and 0 otherwise.

EVEN AND ODD BRAMS

no documentation online



The final blocks, **odd** and **even** are shared BRAMs, from which we will read out the values of using the *spectrometer.py* script.

PARAMETERS

Parameter	Description
Output data type	Unsigned
Address width	2^(Address width) is the number of 32 bit words of the implemented BRAM. There is no theoretical maximum for the Virtex 5, but there will be significant timing issues at bitwidths of 13. QDR or DRAM can be used for larger address spaces. Set this value to 11 for our design.
Data Width	The Shared BRAM may have a data input/output width of either 8,16,32,64 or 128 bits. Since the vector accumulator feeds the shared bram data port with 32 bit wide values, this should be set to 32 for this tutorial.
Data binary point	The binary point should be set to zero. The data going to the processor will be converted to a value with this binary point and the output data type.
Initial values	This is a test vector for simulation only. We can leave it as is.
Sample rate	Set this to 1.

INPUTS/OUTPUTS

Port	Description
Addr	Address to be written to with the value of data_in, on that clock, if write enable is high.

data_in	The data input
we	Write enable port
data_out	Writing the data to a register. This is simply terminated in the design, as the data has finally reached its final form and destination.

CONTROL REGISTERS

no documentation online

There are a few control registers, led blinkers and snap block dotted around the design too:

1. **cnt_rst:** Counter reset control. Pulse this high to reset all counters back to zero.
2. **acc_len:** Sets the accumulation length. Have a look in *spectrometer.py* for usage.
3. **sync_cnt:** Sync pulse counter. Counts the number of sync pulses issued. Can be used to figure out board uptime and confirm that your design is being clocked correctly.
4. **acc_cnt:** Accumulation counter. Keeps track of how many accumulations have been done.
5. **led0_sync:** In the spirit of Kanye (feat. Dwele), we have some flashing lights. I would argue that ours are *the best flashing lights of all time*, but what do I know[4]? Back on topic: the **led0_sync** light flashes each time a sync pulse is generated. It lets you know your ROACH is alive.
6. **led1_new_acc:** This lights up led1 each time a new accumulation is triggered.
7. **led2_acc_clip:** This lights up led2 whenever clipping is detected.

There are also some *snap* blocks, which capture data from the FPGA fabric and makes it accessible to the Power PC. This tutorial doesn't go into these blocks (in its current revision, at least), but if you have the inclination, have a look at their wiki entry at:

http://casper.berkeley.edu/wiki/Snap

In this design, the snap blocks are placed such that they can give useful debugging information. You can probe these through KATCP, as done in tutorial one, if interested.

If you've made it to here, congratulations, go and get yourself a cup of tea and a biscuit, then come back for part two, which explains the second part of the tutorial – actually getting the spectrometer running, and having a look at some spectra.

4 Hope you appreciate the joke more than Taylor Swift

PART TWO

Configuration and Control

Hardware configuration

The tutorial comes with a pre-compiled bof file, which is generated from the model you just went through (`r_spec_2048_r105_2010_Jul_26_1205.bof`)

Copy this over to you ROACH *boffiles* directory, chmod it to *a+x* as in the other tutorials, then load up your ROACH. You don't need to telnet in to the ROACH; all communication and configuration will be done by the python control script.

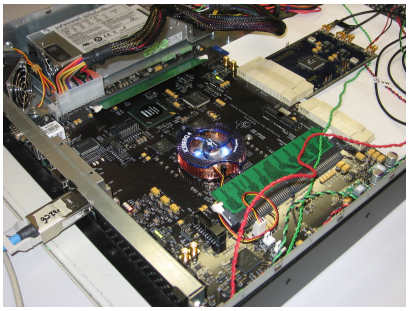
The tutorial comes with a python file called *spectrometer.py*. To use this, you need to have installed a few python libraries. If you haven't already, go through the instructions on

http://casper.berkeley.edu/wiki/Corr

Iit is recommend installing **all** the packages, and documenting any trouble you have on the discussion page of the wiki. Also, iPython is used later on in this tutorial, so install that too.

Next, you need to set up your ROACH. Switch it on, making sure that:

- You have your ADC in *ZDOK0*, which is the one nearest to the power supply.
- You have your clock source connected to *clk_i* on the ADC, which is the second on the right. It should be generating an 800MHz sine wave with 0dBm power.



If set up correctly, it should look like the photo on the right (sans the 10GbE cable).

The spectrometer.py script

Once you've got that done, it's time to run the script. First, check that you've connected the ADC to *ZDOK0,* and that the clock source is connected to *clk_i* of the ADC.

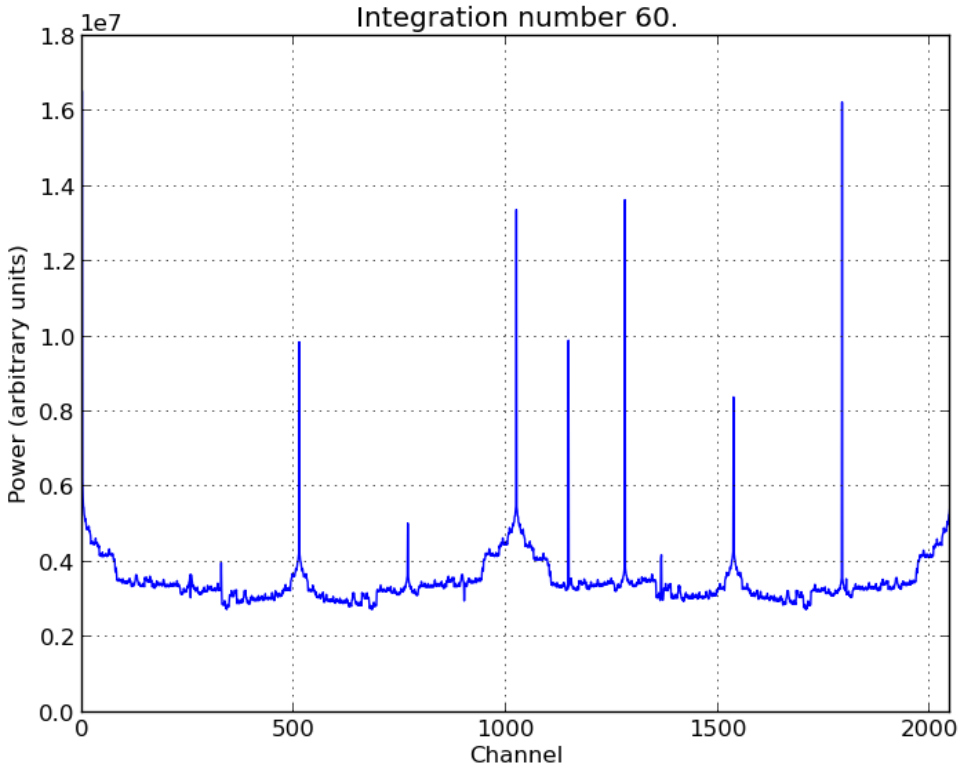
Now, if you're in linux, browse to where the spectrometer.py file is in a terminal and at the prompt type

```
.\spectrometer.py <roach IP or hostname> -b <boffile name>
```

replacing *'<roach IP or hostname>'* with the IP address of your ROACH and boffile name with your boffile. You should see a spectrum like this:

In the plot, there should be a fixed DC offset spike; and if you're putting in a tone, you should also see a spike at the correct input frequency. If you'd like to take a closer look, click the icon that is below your plot and third from the right, then select a section you'd like to zoom in to. The digital gain (-g option) is set to maximum (0xffff_ffff) by default to observe the ADC noise floor. Reduce the gain

(decrease the value (for a -10dBm input 0x100)) when you are feeding the ADC with a tone, as not to saturate the spectrum.



Now you've seen the python script running, let's go under the hood and have a look at how the FPGA is programmed and how data is interrogated. To stop the python script running, go back to the terminal and press *ctrl* + *c* a few times.

iPython walkthrough

The *spectrometer.py* script has quite a few lines of code, which you might find daunting at first. Fear not though, it's all pretty easy. To whet your whistle, let's start off by operating the spectrometer through iPython. Open up a terminal and type:

```
ipython --pylab
```

and press enter. You'll be transported into the magical world of iPython, where we can do our scripting line by line, similar to MATLAB. Our first command will be to import the python packages we're going to use:

```
import corr,time,numpy,struct,sys,logging,pylab
```

Next, we set a few variables:

```
katcp_port = 7147
roach = 'enter IP address or hostname here'
timeout = 10
```

Which we can then use in *FpgaClient()* such that we can connect to the ROACH and issue commands to the FPGA:

```
fpga = corr.katcp_wrapper.FpgaClient(roach,katcp_port, timeout)
```

We now have an fpga object to play around with. To check if you managed to connect to your ROACH, type:

```
fpga.is_connected()
```

Let's set the bitstream running using the *progdev()* command:

```
fpga.progdev('r_spec_2048_r103_2010_Jul_26_1205.bof')
```

Now we need to configure the accumulation length and gain by writing values to their registers. For two seconds and maximum gain: accumulation length, 2*(2^28)/2048, or just under 2 seconds:

```
fpga.write_int('acc_len',2*(2**28)/2048)
fpga.write_int('quant0_gain',0xffffffff)
```

Finally, we reset the counters:

```
fpga.write_int('cnt_rst',1)
fpga.write_int('cnt_rst',0)
```

To read out the integration number, we use *fpga.read_uint():*

```
fpga.read_uint('acc_cnt')
```

Do this a few times, waiting a few seconds in between. You should be able to see this slowly rising. Now we're ready to plot a spectrum. We want to grab the *even* and *odd* registers of our PFB:

```
a_0=struct.unpack('>1024l',fpga.read('even',1024*4,0))
a_1=struct.unpack('>1024l',fpga.read('odd',1024*4,0))
```

These need to be interleaved, so we can plot the spectrum. We can use a for loop to do this:

```
interleave_a=[]

for i in range(1024):
      interleave_a.append(a_0[i])
      interleave_a.append(a_1[i])
```

This gives us a 2048 channel spectrum. Finally, we can plot the spectrum using pyLab:

```
pylab.figure(num=1,figsize=(10,10))
pylab.plot(interleave_a)
pylab.title('Integration number %i.'%prev_integration)
pylab.ylabel('Power (arbitrary units)')
pylab.grid()
pylab.xlabel('Channel')
pylab.xlim(0,2048)
pylab.show()
```

Voila! You have successfully controlled the ROACH spectrometer using python, and plotted a spectrum. Bravo! You should now have enough of an idea of what's going on to tackle the python script. Type *exit()* to quit ipython.

spectrometer.py notes

Now you're ready to have a closer look at the *spectrometer.py* script. Open it with your favorite editor. Again, line by line is the only way to fully understand it, but to give you a head start, here's a few notes:

Connecting to the ROACH

To make a connection to the ROACH, we need to know what port to connect to, and the IP address or hostname of our ROACH. The connection is made on line 98:

```
88. fpga = corr.katcp_wrapper.FpgaClient(...)
```

The *katcp_port* variable is set on line 16, and the *roach* variable is passed to the script at the terminal (remember that you typed *python spectrometer.py roachname*). We can check if the connection worked by using *fpga.is_connected()*, which returns true or false:

```
101. if fpga.is_connected():
```

The next step is to get the right bitstream programmed onto the FPGA fabric. The *bitstream* is set on line 15:

```
18. bitstream = 'r_spec_2048_r103_2009_Nov_23_1234.bof'
```

Then the *progdev* command is issued on line 101:

```
110. fpga.progdev(bitstream)
```

Passing variables to the script

Starting from line 66, you'll see the following code:

```
from optparse import OptionParser

p = OptionParser()
p.set_usage('spectrometer.py <ROACH_HOSTNAME_or_IP> [options]')
p.set_description(__doc__)

p.add_option('-l','—acc_len',dest='acc_len',
type='int', default=2*(2**28)/2048,
help='Set the number of vectors to accumulate between dumps. default is
2*(2^28)/2048, or just under 2 seconds.')

p.add_option('-g', '--gain', dest='gain',
type='int',default=0xffffffff,
help='Set the digital gain (6bit quantisation scalar). Default is
0xffffffff (max), good for wideband noise. Set lower for CW tones.')

p.add_option('-s', '--skip', dest='skip', action='store_true',
help='Skip reprogramming the FPGA and configuring EQ.')

opts, args = p.parse_args(sys.argv[1:])

 if args==[]:
```

```
        print 'Please specify a ROACH board. Run with the -h flag to see
        all options.\nExiting.'
        exit()
    else:
        roach = args[0]
```

What this code does is set up some defaults parameters which we can pass to the script from the command line. If the flags aren't present, it will default to the values set here.

Conclusion

If you have followed this tutorial faithfully, you should now know:

- What a spectrometer is and what the important parameters for astronomy are.
- Which CASPER blocks you might want to use to make a spectrometer, and how to connect them up in Simulink.
- How to connect to and control a ROACH spectrometer using python scripting.

In the next tutorial, you'll be looking at a wideband 'FX' correlator design on the ROACH. You'll see quite a few of the blocks you were introduced to today reused, as the 'F' of a 'FX' correlator is essentially a spectrometer. Alternatively, tutorial 5 will take you through how to build a “million channel spectrometer”, along with a few simulink tricks.