Challenges in UVM + Python random verification environment for Digital Signal Processing datapath design.

Shabbar Vejlani and Ashok Chandran
Analog Devices
Bangalore, India – 560016

***Abstract*- The paper discusses challenges in verification of reconfigurable datapath designs using novel constrained random approach for signal generation and performing dynamic scoreboard/check in frequency domain. We highlight the key challenges in generating random streams for characterizing a design in frequency domain. This approach uses UVM and leverages the signal processing power of Python packages. Communication is established between UVM and python to implement advanced signal processing capabilities inside the UVM testbench. This avoids need for post-processing and provides path to extended frequency domain checks within UVM.**

I. INTRODUCTION

In context of this paper, the reconfigurable signal processing block primarily references to design consisting of cascaded filter chains, interpolation/decimation blocks and direct digital synthesizers (frequency up/down translators). For high speed datapath designs, we typically have highly parallelized designs and involved clocking schemes. To support reconfiguration in such designs, the interfaces (muxes and retimers) between signal processing blocks too constitute an important part of the design. Thus, the verification involves making sure that the overall integration as well as system level signal processing is happening as per the specification.

There are two popular methods for verifying a signal processing datapath

1. *Time Domain* – For time domain, the traditional approach is to develop cycle/bit accurate reference models. The time domain model might be a thorough approach accounting for fine level details. The downside is that it may take lot of time for model development often requiring to model to level of detail of design itself. Also, the interface level details get obscured in this approach, unless end to end modelling is done.
2. *Frequency Domain* – In this approach, the system is characterized in the frequency domain and checked against application specific frequency domain metrics typically (SNR, SFDR, DC,THD etc.).

Due to the nature of the end application, we have chosen the frequency domain approach for datapath verification. The frequency domain approach has been primarily directed testing approach for past work. This was because of the difficulties involved in input generation and output checking in frequency domain. Also, design complexity had been simpler in past projects. There were multiple limitations noticed with the traditional datapath verification approach:

1. The checking approach has been using propriety software (like MATLAB) which are license limited or precompiled executables.
2. The past approach involves checking after simulation is complete. There was no clean way of feedback to the testbench so that a dynamic scoreboard approach can be established.
3. With highly configurable, multi-clock designs, the number of test cases just explodes at system level and it becomes quite involved to thoroughly verify using the directed approach.

To overcome these limitations in past directed verification platforms, a random testing environment using UVM + Python was developed. The key highlights of the approach are:

1. *Python:* For signal processing capabilities which enable the checks in frequency domain
2. *Constrained Random Verification (CRV):* CRV approach for exhaustive verification for signal processing designs. The UVM framework was leveraged for block level reuse and scalability at system level.
3. *Dynamic Scoreboarding in Frequency Domain:* The approach implemented a frequency domain scoreboard which enables it to verify the frequency characteristics which were randomized. This enables dynamic adjustment of verification check based on system configuration. For example, reduced SFDR performance may be fine in some modes.

II. Datapath UVM Testbench Architecture

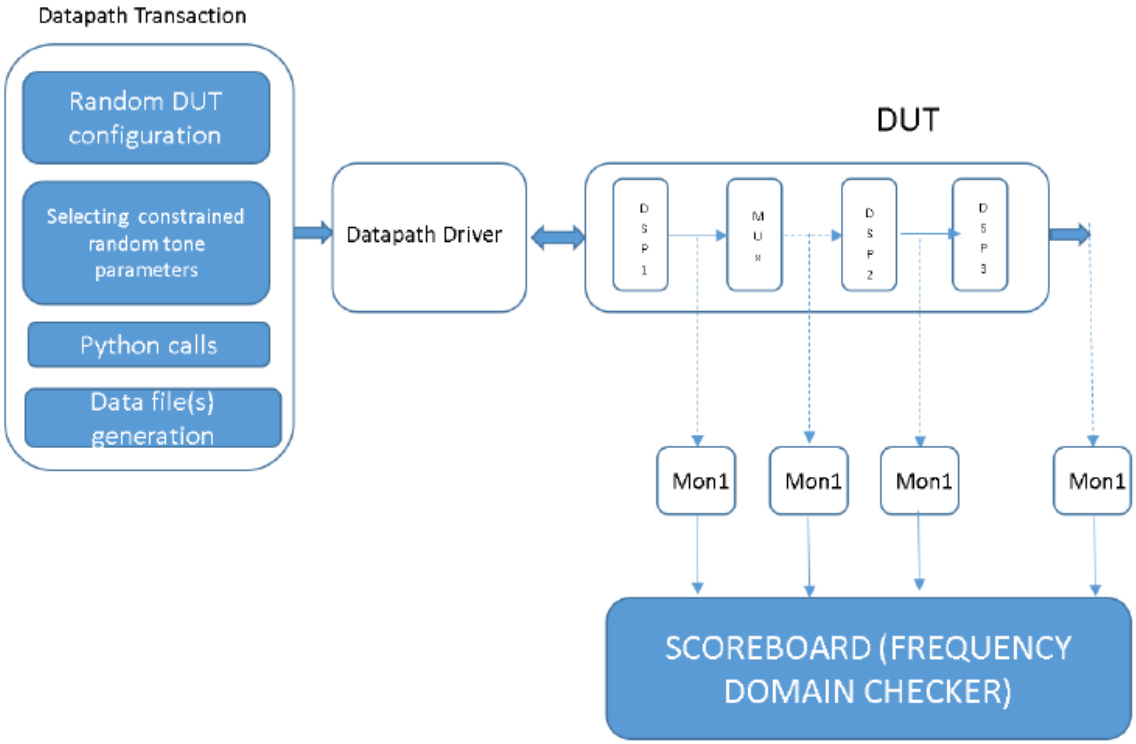

Figure 1- Datapath Testbench Architecture

The datapath testbench is developed in UVM with configurability in mind. This enables it to be reused to a different signal processing datapath architecture very easily. We will go over the architecture in depth below and understand the challenges.

III. Random Signal Generation

Sinusoidal input or the more general complex sinusoid is used to characterize datapath systems. The primary analysis is to perform an FFT on block of data sequence, analyze the spectral components and quantify the frequency domain characteristics. The point to note here is that the analysis and characterization in frequency domain is only as good as the input itself. Thus, with regards to data generation we need to ensure a clean good quality sinusoid. To appreciate this we need to understand the effects of quantization and the FFT artifacts.

A. Understanding Signal Random Variables

In this discussion, we will limit to single tone testing of the DSP datapath. A single complex data stream can be described with the below expression:

$$\boldsymbol{Y = Ae^{(2j\pi ft+\emptyset)}}$$ (Single Complex Data Stream)

To generate such a complex data stream, you would need to parameterize various attributes of the signal.

1. A – Amplitude
2. f – Frequency
3. ϕ – Phase
4. T – time instant
5. Real vs Complex Input

Unlike a typical transaction randomization scenario where the address or data are randomized, in this case it is the frequency, amplitude etc. which is randomized and constrained as per the frequency domain characteristics of the datapath. Even though it might seem that any real frequency value could be chosen, the fact that we use a finite length Fast Fourier Transform (FFT) places some constraints on the values in order to have meaningful analysis.

B. Spectral Leakage

For any frequency domain analysis, we need to take the FFT of the data stream. FFT (which is just a way to perform DFT efficiently) is an N-point to N-point mapping. The transformation decomposes the data as a weighted sum of N complex sinusoids. These N complex sinusoids can be viewed as N-frequency bins and the magnitude of the bin tells which bin our data closely matches with. If the data is sampled at a frequency of fs, the frequency represented by the n^{th} bin is,

$$freq\_bin[n] = n * fs/NFFT$$

Where,

n – Nth bin number
fs – Sampling frequency
NFFT – Length of FFT

Now let us consider a simple case to understand the relationship between data generation and analysis. Figure 2 shows frequency domain plot of 1MHz sinusoid sampled at 100MHz. If we take a 100 point FFT, each frequency bin has resolution of fs/NFFT, 100MHz/100 = 1MHz.

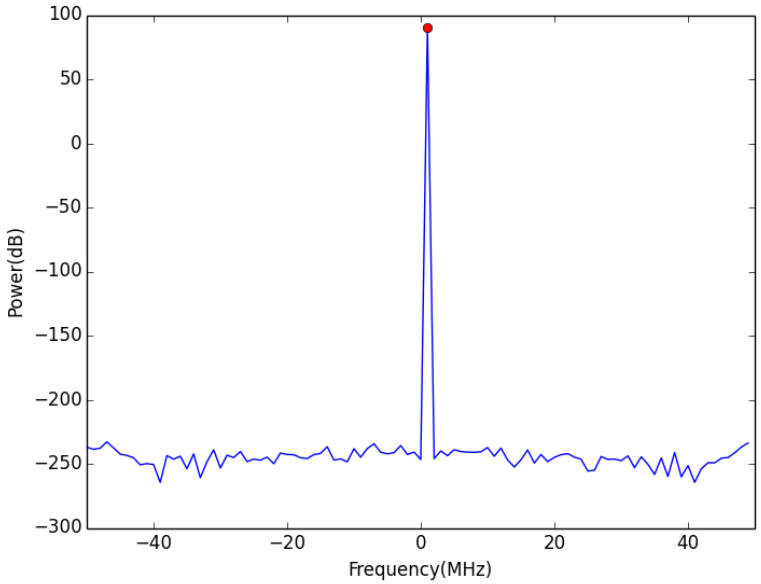

Figure 2 - 100 point FFT of 1 MHz sinusoid sampled at 100 MHz

As expected, we observe the power to be concentrated only in one bin (1MHz) and the rest is quantization noise for floating point resolution far way beyond the -200dB . Figure 3 below shows 99 point fft for the same input.

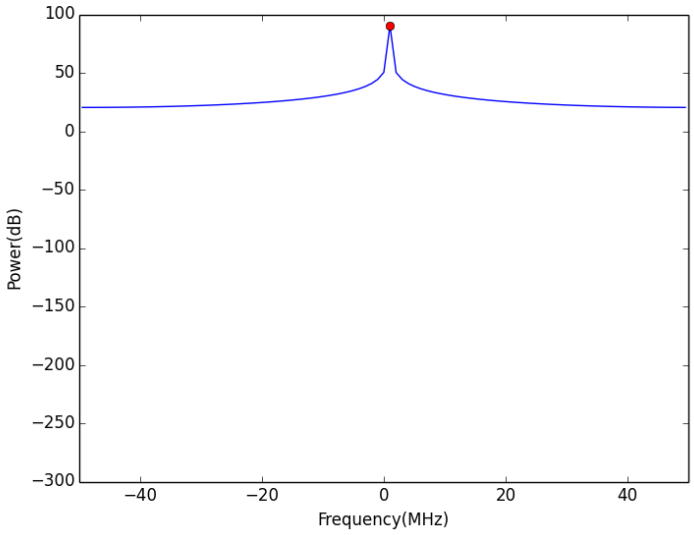

Figure 3 - 99 point FFT of 1 MHz sinusoid sampled at 100 MHz

We now notice that power is now not concentrated in one bin, but is spread now spread across all the bins, though the prominent peak is still around 1MHz. This is because for 99 point FFT, the resolution is 100/99=1.01 and the input frequency thus does not fall into a bin. More technically, it is because FFT looks at the input with periodicity same as the length of FFT. Thus, when we take the number of samples (or length of FFT) such that non-integral number of cycles are captured, FFT perceives it as a discontinuity which shows up as power spread across bins. This effect is known as "Spectral Leakage".

The result of this artifact is that it gives a misleading picture, giving incorrect power values and also shows power to be present in other bins. This further gives the wrong estimation of the frequency domain characteristics. There are many techniques present to avoid this issue, for example, by using different windowing techniques or applying various other power spectrum estimation and correction techniques. One simple way is to ensure to generate a frequency such that the frequency falls in a bin for a pre-determined FFT size. Even though it might limit the resolution of the frequencies available, that can be adjusted for by taking reasonable large FFT size.

This technique of choosing frequency which falls in a bin (or equivalently choosing FFT length which captures an integral number of cycles) is known as coherent frequency generation.

Thus, when we generate a tone, we make sure that after all the datapath processing the tone translates to a frequency such that it falls in a bin for the given FFT length.

C. Effect of Truncation vs rounding

Let us consider the effect of truncation vs rounding for fixed point systems. Following is the expression for the data for 16-bit quantization case:

$$x_n = 32767 * \exp(j * 2 * \pi * freq * n * fs)$$

$$x_n(real) = 32767 * \cos\left(2 * \pi * freq * \frac{n}{fs}\right) + 1j * 32767 * \sin(2\pi * freq * \frac{n}{fs})$$

$$x_n(integer) = int\left(32767 * \cos\left(2 * \pi * freq * \frac{n}{fs}\right)\right) + 1j * int\left(32767 * \sin\left(2 * \pi * freq * \frac{n}{fs}\right)\right)$$

The data constitutes two parts, real and the imaginary, each of which is quantized to 16bits and represents a full scale value of +-32767. The point to note here is that the quantization takes place from the real value to integer value, and it is important to make sure we implement rounding when doing the quantization. This is to avoid DC bias in the data, which gets introduced if data is truncated directly. Also the magnitude of spurs is smaller in the rounding case when compared with truncation case. Figure 4 shows the frequency domain plot of truncation vs rounding.

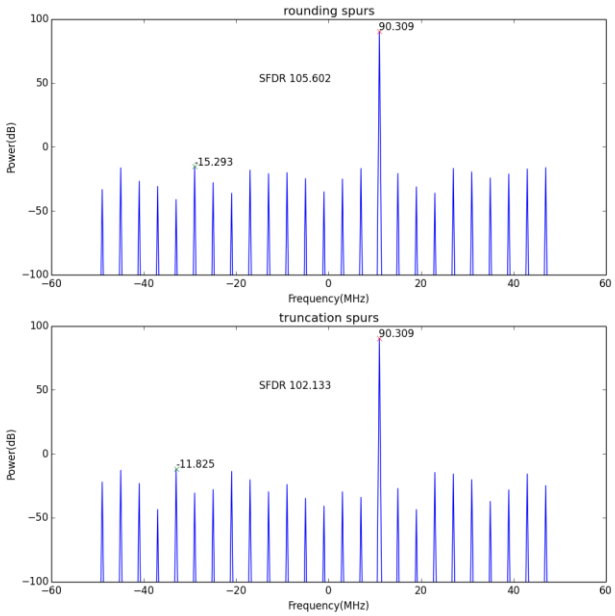

Figure 4 - Effect of truncation vs rounding

D. Effect of Quantization

To understand the effect of quantization noise, let us see the quantization noise for a complex sinusoid of 10MHz(fs=100MHz) vs frequency of 11MHz (fs=100MHz), both quantized to 16bits. Let us consider FFT block size of 100, so that both the frequencies 10MHz and 11MHz fall in bin 10 and bin 11 respectively. Figure 5 and Figure 6 shows the time and frequency domain plots of quantization noise.

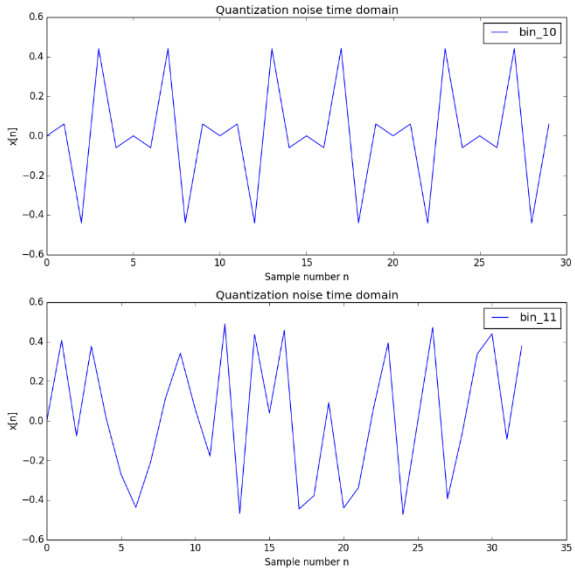

Figure 5 - Quantization noise time domain plot

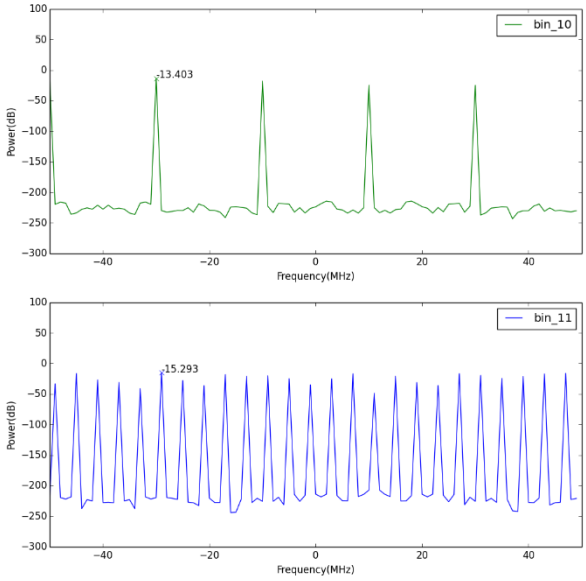

Figure 6 - Quantization noise frequency domain plot

We see when the frequency is placed in bin_10, the quantization noise is periodic. This results in power to distribute more in the harmonics and degrades the spurious performance. In contrast, for bin_11, we have quantization noise to not have such periodic behavior and appears to be more random. This happens when the sampling frequency and the tone frequency have integer relationship, which results in every period to have the exactly same set of data. Since we generate the tone to fall in a bin, *fs/freq* relationship is *nfft/bin_no*. For first case, this is 100/10 = 10, which is an integer and for the other case it is 100/11 which is a fractional number. There are multiple ways to resolve this issue. We can introduce dither such that the quantization noise distribution is random or we can play around with the FFT size and bin no. This paper uses the second approach. Basically we need that the ratio of *nfft/bin_no* is non-integer. Stated otherwise, *nfft* and *bin_no* should be relatively prime. There are two ways to achieve this *:*

1. *Prime Bins:* If the fft length is even, choosing an odd or prime bin ensures this, but this results in many of the bins not used.
2. *Prime FFT length:* Another approach is to choose a prime length FFT, in which case this ratio will be non-integer for all the bin numbers. For example, for the commonly chosen size of 4096, choosing a FFT size of 4099 resolves this issue.

E. Summarization Data Generation Requirements

Thus, clean sinusoid data generation involves generating tone(s) which fall on a bin for FFT block size chosen for analysis making sure that the bin(s) chosen and the FFT size are relatively prime and rounding is taken care of while quantizing. The test approach must ensure that the bin selection is constrained as per the datapath specifications such that post all datapath operations the data/tone still falls on the bin at the output.

Below figure summarizes the overall idea of constraining:

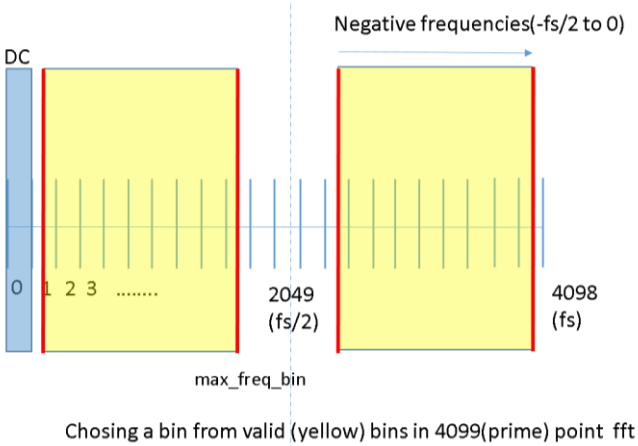

Figure 7 - Concept of Bin Based Constraining

V. Data Driving And Monitoring

A typical high-speed datapath design generally involves a lot of parallelism as well as multiple clocks with multiphasing. Due to the data represented in the frequency domain and flowing across multiple streams with multiple phase clocks, debugging from waveform tends to be unyielding and requires rather an analysis of the data in frequency domain. In some debug environments, using waveform debug features, even though it is possible to view time domain analog plot, it is misleading at times. Simple tones might look modulating in time domain even though in the frequency domain it appears perfectly fine. Following Figure 8 shows the time domain plot for a 41MHz tone sampled at 100MHz, which exemplifies this effect:

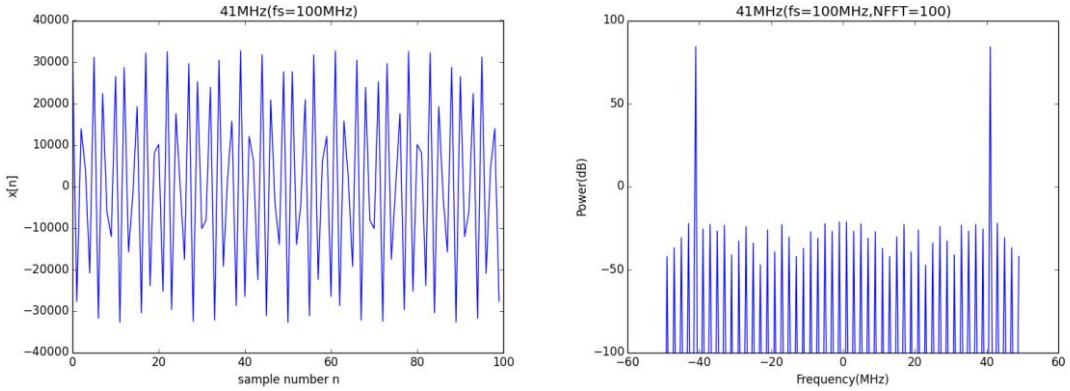

Figure 8 - Time vs Frequency Domain Plot for a 41Mhz tone sampled at 100Mhz

Thus, we require frequency domain debug utilities as well as monitors working on multi phase clocks interleaving data in a proper way, to interpret the data streams correctly. Using SV constructs it was possible to create template for reconfigurable monitors, deployable at multiple hierarchies. The monitors as such dump the interleaved data into files which are utilized by scoreboard for checks. The monitor control for dumping was achieved using UVM global events for ease of control from any part of the testbench.

VI. Frequency Domain Scoreboarding

Unlike the normal data scoreboarding with reference model data, in this case, we check for different frequency domain characteristics. Two basic checks are bin number and SFDR (Spurious free dynamic range). During data generation, we generate data considering the entire datapath configuration and we know that post datapath processing and frequency translations which bin(s) do we expect the power to be present in. Thus, when compared to normal designs, here the focus is on scoreboarding of frequency “bins” vs scoreboarding of “data”. The expected bins depend on the total datapath configuration and the frequency translation blocks. From the interface or retimers or muxes perspective though it is helpful to discuss SFDR as a verification metric. SFDR is a measure of linearity of

the system. Any non-linear effects will results in extraneous spurs which should be within the tolerance specifications of the datapath design. To state simply, for single tone case, SFDR is the difference in the power in dB between the tone's fundamental peak and the next peak. Basically, it dictates that for the single tone input, all the spurs other than the tone should be below a threshold level. There are several other metrics like THD, SNDR(SNR) etc. which too throw more light into the system information but SFDR is simplest to evaluate especially when we ensure the data generation scheme mentioned before.

A. Typical Datapath Bugs

As discussed in the introduction, retimers and muxes constitute an important aspect of the datapath designs and interface related bugs are plenty in a complicated design. To appreciate the frequency domain metrics, let us see some examples of simple bugs in interface blocks.

Repeating Data

Consider that instead of the normal data sequence, each data is repeated twice at the interface. Thus we have sequence of the *a,a,b,b,c,c*. instead of *a,b,c,d,e,f*. Figure 9 below shows the SFDR impact in such a case for 11MHz complex sinusoid(fs=100MHz and NFFT=100).

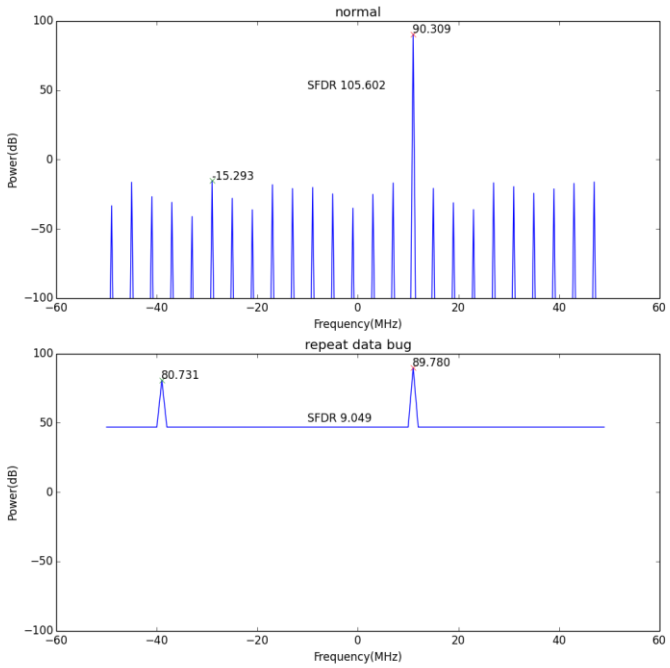

Figure 9 – Repeating Data Bug

Alternate Zero Data Bug

Consider that instead of the normal data sequence, zero is inserted every second sample due to an interface bug. Thus, we have the sequence *a,0,c,0,d,0* instead of *a,b,c,d,e,f*. Below shows the SFDR impact in such a case for 11MHz complex sinusoid(fs=100MHz and NFFT=100).

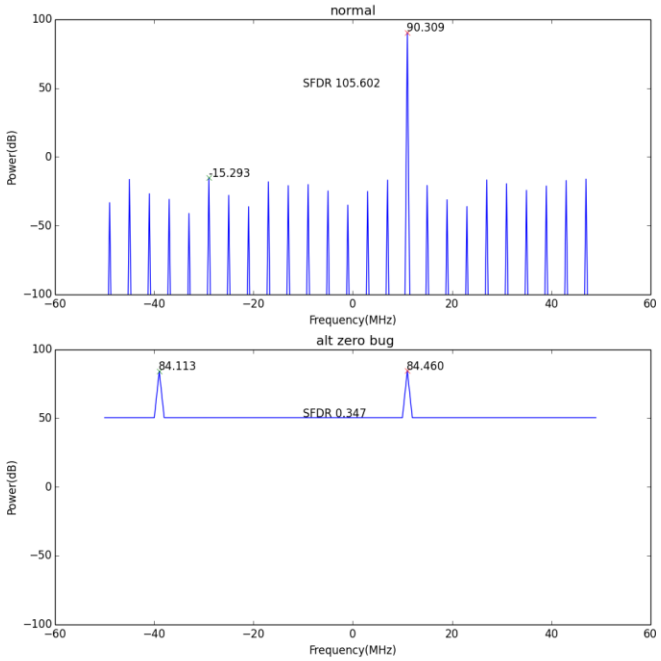

Figure 10 – Alternate Zero Bug

Saturation Issues

During data-width conversion generally, there is saturation and rounding logic implemented. Generally, it is difficult to feed directed input to exercise such logic sitting in intermediate stages. But provided that this logic is exercised, it is possible to detect such issues.

Let us consider a simple case when the saturation logic is missed and data overflow happens. In 2's complement implementation, an overflow will result in data to go to the negative minimum instead of saturating to the maximum value. The following shows the time domain and frequency domain plots for the two cases:

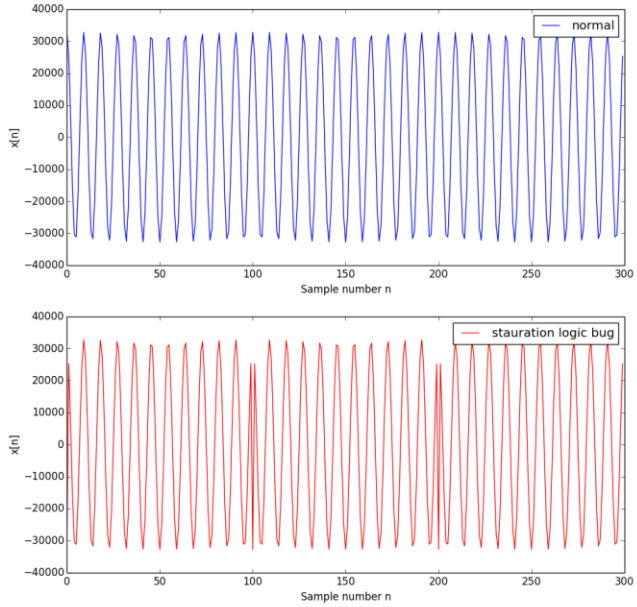

Figure 11- Saturation Logic Bug Time Domain Plot

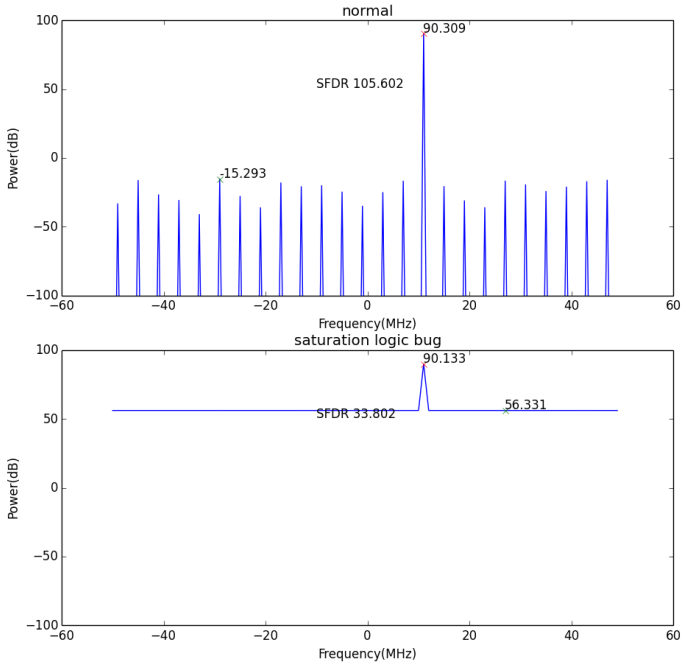

Figure 12 - Saturation Logic Bug Frequency Domain Plot

Latency Mismatch Issue

In parallel processing systems, the latency between the paths is essential to ensure the integrity of the data. Even though latency checks are generally checked in the time domain, it is also possible to check for relative latency between two paths in the frequency domain using quadrature inputs. As an example, consider the following complex sinusoid in Euler's form : $x_n = \cos(\omega t) + 1j * \sin(\omega t)$

We feed the real part on one path and the imaginary on the other. We also ensure identical configuration on both the paths. At the output, we take combined FFT of the two paths, we should ideally get just a single peak. Any latency mismatch between the two paths will show up at the mirror spur. Let us take an example where we are feeding 11MHz complex tone(fs=100MHz), and compare the spectrum(nfft=100) with the case when we have one cycle latency mismatch between the real and the imaginary paths.

Consider the normal I and Q sequences as:

$I = \ldots i_n, i_{n+1}, i_{n+2}, i_{n+3}, \ldots$

$Q = \ldots q_n, q_{n+1}, q_{n+2}, q_{n+3}, \ldots$

Then in the system due to one cycle latency mismatch, the sequence we get is :

$I = \ldots i_{n-1}, i_n, i_{n+1}, i_{n+2}, \ldots$

$Q = \ldots q_n, q_{n+1}, q_{n+2}, q_{n+3}, \ldots$

While evaluating SFDR, we do frequency analysis on the complex data stream I + 1j*Q.

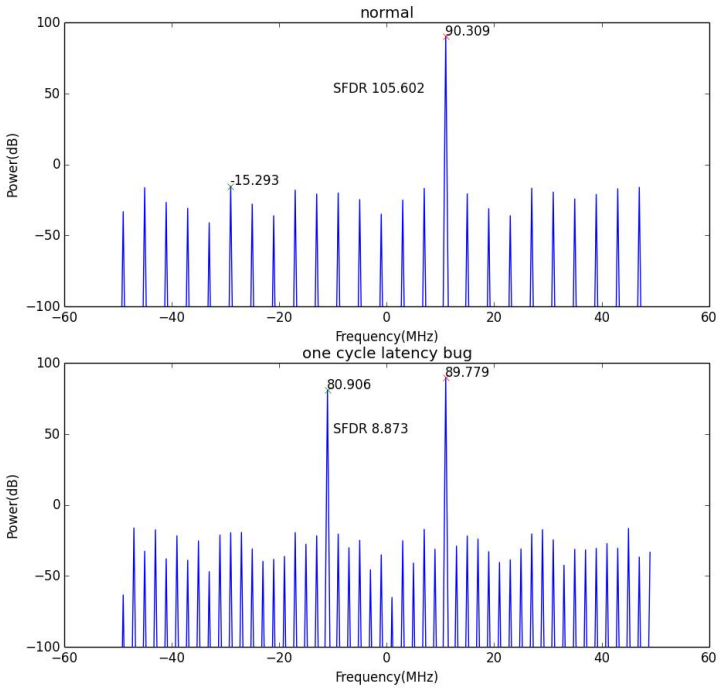

Figure 13 - Latency Mismatch Bug

Thus from the above examples, we see that there is considerable SFDR degradation for many of the common interface issues and it is very easy to detect them using SFDR as a frequency domain check metric.

VII. UVM + PYTHON INTEGRATION

Python is an open source scripting language with very powerful library support. For signal processing and visualization, the useful python libraries are *numpy, scipy and matplotlib*.

1. *Numpy* provides the basic array type and related operations. Particularly broadcasting of values and operations across array indices makes it very easy to do array processing.
2. *Scipy* is scientific computation package and includes a whole lot of typical signal processing operations.The *scipy* package goes hand-in-hand with the *numpy* package.
3. *Matplotlib* package provides for data visualization.

All these libraries along with host of other signal processing, mathematics and data analysis package (particularly *pandas* and *sqlite3*) are bundled and available in the open source distribution *anaconda*.

The approach in datapath verification is to use UVM to decide on datapath configuration. Once the configuration is decided, $system( ) calls to python are made to generate the corresponding data files. These calls as such are abstracted into SV classes and appear in the testbench as plain function calls. The OOPS approach enables to modify the function

calls by overriding the virtual methods and using the UVM factory. This provides quite some flexibility in constraining data easily.

For the datapath scoreboarding, the data (blocks) from the monitor are dumped into files. The transaction information is passed to the scoreboard which again makes calls to python in a similar way. Python reads appropriate files and evaluates the bins and the frequency domain metrics. The evaluated metrics are read into the testbench and based on the datapath configuration are checked against the expected values for bins and other frequency domain metrics.

By using SV string hashes, we managed to achieve a very clean way to communicate the data between python and SV by means of files. The approach is very scalable and any new metric can be added very easily. The important point to note here is that the checks happen within the SV environment. This provides a lot of flexibility in terms of thresholds and tolerances for the evaluated metrics based on different datapath configurations.

The code snippet in the Appendix show a small example illustrating the UVM-Python data exchange by means of SV hashes. Following diagram shows how this wrapper is implemented in UVM framework.

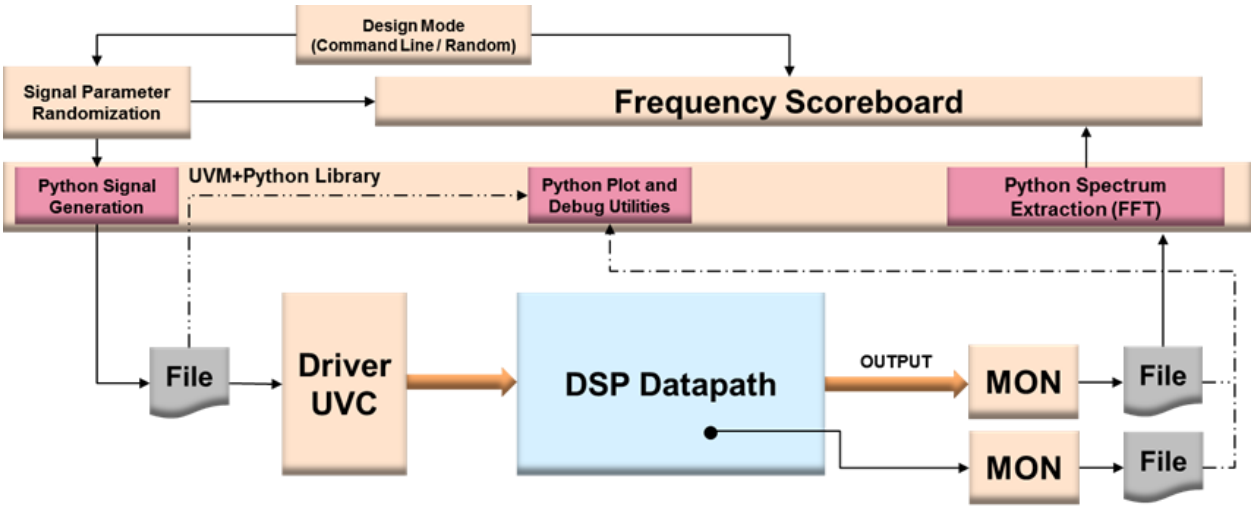

Figure 14 - UVM+Python Approach Overview

VIII. Limitations And Future Work

Fine level nuances might not show up easily in the frequency domain approach. These are the rounding/truncation blocks as well as the saturation blocks, particularly sitting in the intermediate stages, which may not get exercised easily. From a directed end-to-end approach they require custom directed inputs, which might not get exercised by feeding tone(s). Also, for the case of rounding vs truncation effects, their effect on the spectrum may not be prominent and if the limits for the metrics are too relaxed, they might be missed altogether. Hence, it is as per our experience, best to check these blocks locally too at a block level.

Even though it is possible to catch latency differences in the frequency domain, it is not possible to evaluate the absolute system latency. To calculate system latency, impulse input based, time domain checks are required.

VIII. CONCLUSION

The notion of constraining frequency bins, relationship with FFT size, rounding and quantization noise were discussed, explaining the importance of clean input sinusoid in order to have meaningful analysis in frequency domain.

Python as an open source alternative helped us to run more scenarios without dependence on any licenses. Its powerful library support for signal processing enables reuse of existing code and quick bring-up of checks in frequency domain. The UVM-Python interface wrapper enabled the data exchange with python by use of files and SV hashes.

We also saw some typical interface bugs and how interface related bugs can be caught in the frequency domain. Thus, we understood the challenges in constraining and checking of data in frequency domain.

Using the constrained random approach we achieved more thorough verification and enabled coverage of a greater number of system-level scenarios without requiring to have directed tests for a highly reconfigurable datapath design. We believe that, once the underlying DSP concepts are understood, it is fast to setup frequency domain checks when compared to time domain approach and enables quick verification of integration at system level.

Using UVM enabled reuse of sequences, monitors, and checkers at the system-level and eased the system level verification.

ACKNOWLEDGMENT

Thanks to High Speed Convertor Team at Analog Devices, especially to Dr. Ganesh Ananthaswamy.

REFERENCES

1."Understanding Digital Signal Processing",R.Lyons.

APPENDIX

The code below shows how the details of the uvm python interface. The generation and checking by means of $system calls to the python are shown abstracted in classes. It is part of the package dp_pkg.sv. There is a example_gen.py script for data generation and example_check.py to evaluate some frequency domain parameters. For ease of understanding a simple tb(tb.sv) exercising this interface is shown.

dp_pkg.sv(1)

```
package dp_pkg;
`include "uvm_macros.svh"
import uvm_pkg::*;

typedef string str_hash[string];

//Input info from file to hash
function str_hash update_hash(string file_name);
int fh,not_eof,ret_val;
string val,key;
str_hash opts;
begin
    opts.delete();
    fh=$fopen(file_name,"r");
    if(fh==0)
    begin
        `uvm_fatal("FOPEN",$sformatf("Could not open file %s",file_name));
    end
    not_eof=$feof(fh);
    while(!not_eof)
    begin
        ret_val=$fscanf(fh,"%s : %s",key,val);
        not_eof=$feof(fh);
        opts[key]=val;
    end
    $fclose(fh);
    return opts;
end
endfunction
```

dp_pkg.sv(2)

```
class pycmd extends uvm_object;
`uvm_object_utils(pycmd)
string script_name,script_path;
string cmd;
string cmd_type;

function new(string name="");
        super.new(name);
        script_path=`CHK_SCRIPTS;
        cmd_type="python ";
endfunction

function void add(string opt);
        cmd={cmd,opt};
endfunction

function void rst();
        cmd="";
endfunction

function void exec();
        cmd={cmd_type,script_path,"/",script_name," ",cmd};
        $display("Executing Command:\n %s \n",cmd);
        $system(cmd);
endfunction

endclass
```

dp_pkg.sv(3)

```
//helper class used in actual datapath transaction
class tone_gen extends uvm_sequence_item;
`uvm_object_utils(tone_gen)
real freq;
string fname;
real fs;
int nfft;
int nbits;
int ns;

function new();
  super.new();
endfunction

virtual function void gen();
  pycmd c;
  c=new();
  c.script_name="example_gen.py";
  c.add($sformatf(" -fname %s",this.fname));
  c.add($sformatf(" -fs %f",this.fs));
  c.add($sformatf(" -nfft %0d",this.nfft));
  c.add($sformatf(" -nbits %0d",this.nbits));
  c.add($sformatf(" -freq %0f",this.freq));
  c.add($sformatf(" -ns %0d",this.ns));
  c.exec();
endfunction

endclass
```

dp_pkg.sv(4)

```
class  eval_param extends uvm_object;
`uvm_object_utils(eval_param)

pycmd c;
str_hash res;

real fs;
string fname;
int nfft;

function new(string name="");
  super.new(name);
endfunction

virtual function void eval();
begin
        c=new();
        c.script_name="example_check.py";
        c.add($sformatf(" -fs %0f",this.fs));
        c.add($sformatf(" -nfft %0d",this.nfft));
        c.add($sformatf(" -fname %s",this.fname));
        c.exec();
        res=update_hash({"results_",fname});
end
endfunction
endclass

endpackage
```

example_gen.py

```
from pylab import *
import numpy as np
import argparse

parser = argparse.ArgumentParser(description='Example signal generation')
parser.add_argument('-fs', help='Sampling Frequency',type=float,required=True)
parser.add_argument('-freq', help='Tone Frequency',
type=float,default=0,required=False)
parser.add_argument('-nfft', help='FFT
Length',default=4096,type=int,required=False)
parser.add_argument('-ns', help='Total number of
samples',type=int,default=1000,required=False)
parser.add_argument('-nbits', help='Number of bits to quantize to
',type=int,default=16,required=False)
parser.add_argument('-fname', help='Output file
name',type=str,default="output",required=False)
args = parser.parse_args()


fs=args.fs
ns=args.ns
freq=args.freq
nbits=args.nbits
scale_factor=2**(nbits-1) -1

nts=np.arange(0,ns,dtype=float)/float(fs)
out=np.cos(2*np.pi*freq*nts)
out=out*scale_factor
out=np.round(out)
np.savetxt(args.fname,out,fmt='%d',delimiter='\n')
```

example_check.py

```
import numpy as np
from scipy import fftpack as f
import argparse

parser = argparse.ArgumentParser(description='Example Script')
parser.add_argument('-fs', help='Sampling
Frequency',type=float,required=True)
parser.add_argument('-fname', help='Input File
name',type=str,default="None",required=False)
parser.add_argument('-nfft', help='FFT
size',type=int,default=4096,required=False)
args = parser.parse_args()

data=np.loadtxt(args.fname,dtype=int)
fft_size=args.nfft
fs=args.fs

data=data[-fft_size:]
fft_data=f.fft(data)
fft_data=abs(fft_data)
fft_data=fft_data/float(fft_size)
fft_data=fft_data[0:fft_size/2]

peak_bin=np.argmax(fft_data)
peak_freq=peak_bin*fs/float(fft_size)
peak_value=fft_data[peak_bin]
peak_value_dB=20*np.log10(peak_value)

fh=open("results_"+args.fname,"w")
fh.write("PEAK_BIN : {:d}\n".format(peak_bin))
fh.write("PEAK_FREQ : {:.5f}\n".format(peak_freq))
fh.write("PEAK_dB : {:.5f}\n".format(peak_value_dB))
fh.close()
```

tb.sv(1)

```
`define CHK_SCRIPTS "."

`include "uvm_macros.svh"
import uvm_pkg::*;
module tb();

dp_pkg::eval_param e;
dp_pkg::tone_gen t;

int peak_bin;
real peak_freq;
real peak_dB;
initial
begin

  //This sits inside transaction class
  t=dp_pkg::tone_gen::type_id::create("example_generation");
  t.fs=100e6;//100MHz
  t.nfft=100;//FFT size
  t.freq=10e6;//10MHz ==> bin=10
  t.ns=1000;//Number of samples
  t.fname="example";//Output filename
  t.nbits=16;//Number of bits
  t.gen();


  //This sits inside scoreboard class
  e=dp_pkg::eval_param::type_id::create("example_checker");
  e.fs=100e6;//100MHz sampling freq
  e.nfft=100;//FFT size
  e.fname="example";//Input filename
  e.eval();
```

tb.sv(2)

```
peak_bin=e.res["PEAK_BIN"].atoi();
 if(peak_bin!=10)
   begin
     `uvm_fatal("",$sformatf("Expected BIN=10, Found BIN=%0d",peak_bin))
   end
   else
   begin
      `uvm_info("",$sformatf("Found BIN=%0d",peak_bin),UVM_NONE)
   end
   peak_dB=e.res["PEAK_dB"].atoreal();
   if(peak_dB < 80)
   begin
     `uvm_fatal("",$sformatf("Peak power =%0f dB, expecting greater than 80
dB",peak_dB))
   end
   else
   begin
     `uvm_info("",$sformatf("Peak power =%0f dB",peak_dB),UVM_NONE)
   end

end
endmodule
```