

Design of a Wideband Fractional-N Frequency Synthesizer Using CppSim

Scott E. Meninger

<http://www-mtl.mit.edu/research/perrottgroup>

May 13, 2005

Revised July 2008

Copyright © 2005 by Scott E. Meninger

Some material reprinted with permission from Michael H. Perrott

All rights reserved.

Table of Contents

Setup.....	2
Introduction.....	4
A. Quantization Noise Impact on Synthesizer Performance.....	5
B. Wideband Synthesizer Architecture.....	5
C. Qualitative Explanation of PFD/DAC Operation: The Charge Box.....	7
Design Goals for an Example PFD/DAC Synthesizer.....	8
Performing Basic Noise Analysis Using the PLL Design Assistant.....	9
Performing Basic Operations within Sue2 and CppSimView.....	12
A. Opening Sue2 Schematics.....	12
B. Running the CppSim Simulation.....	15
Plotting Time Domain Results.....	17
A. PFD/DAC Output and VCO control Voltage.....	17
Plotting Frequency Domain Results.....	19
Exploring Non-Idealities within the PFD/DAC.....	21
A. PFD/DAC Unit Element Mismatch.....	22
B. Timing Mismatch in the PFD/DAC.....	25
C. Shape Mismatch and the Sample-and-Hold function.....	27
Directly Modulating the PFD/DAC Synthesizer.....	29
Conclusion.....	33

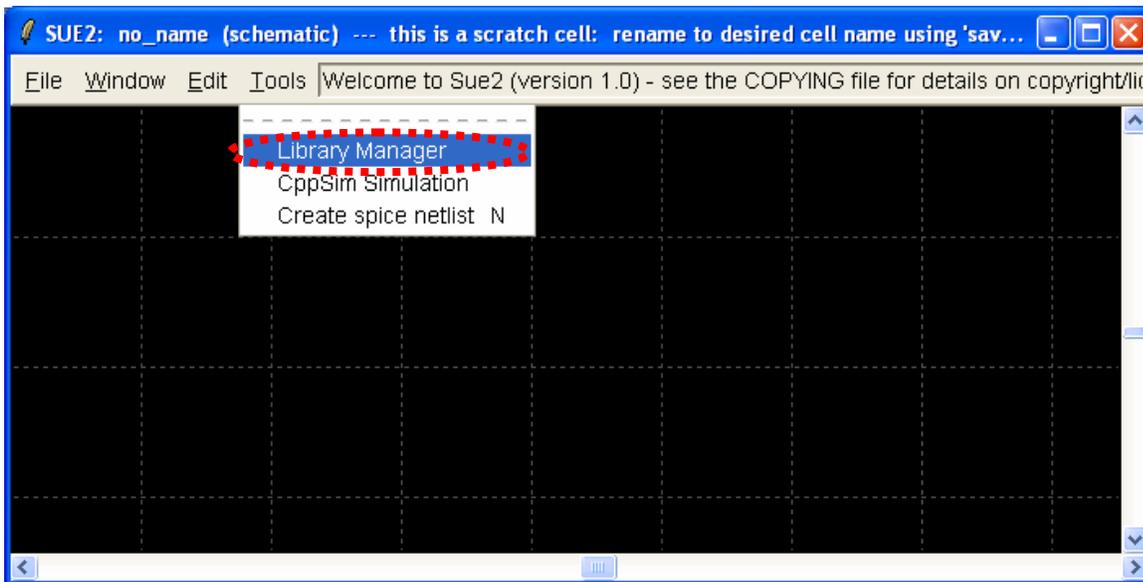
Setup

Download and install the CppSim Version 3 package (i.e., download and run the self-extracting file named **setup_cppsim3.exe**) located at:

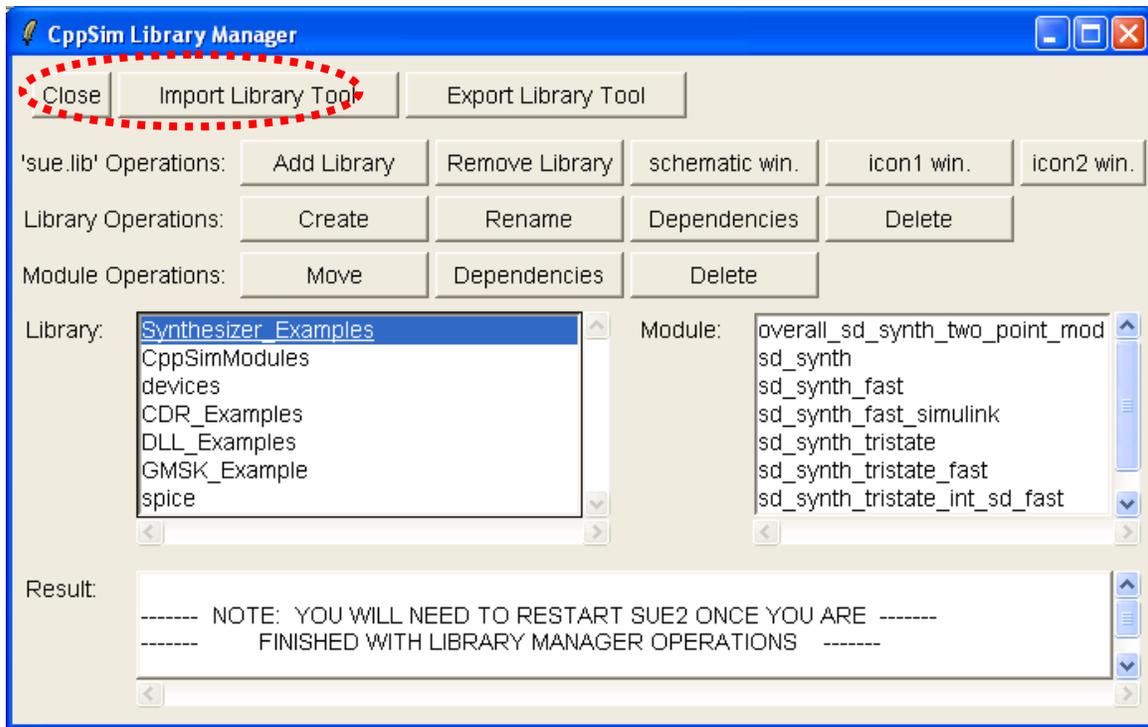
<http://www.cppsim.com>

Upon completion of the installation, you will see icons on the Windows desktop corresponding to the PLL Design Assistant, CppSimView, and Sue2. Please read the “**CppSim (Version 3) Primer**” document, which is also at the same web address, to become acquainted with CppSim and its various components. You should also read the manual “**PLL Design Using the PLL Design Assistant Program**”, which is located at <http://www.cppsim.com>, to obtain more information about the PLL Design Assistant as it is briefly used in this document.

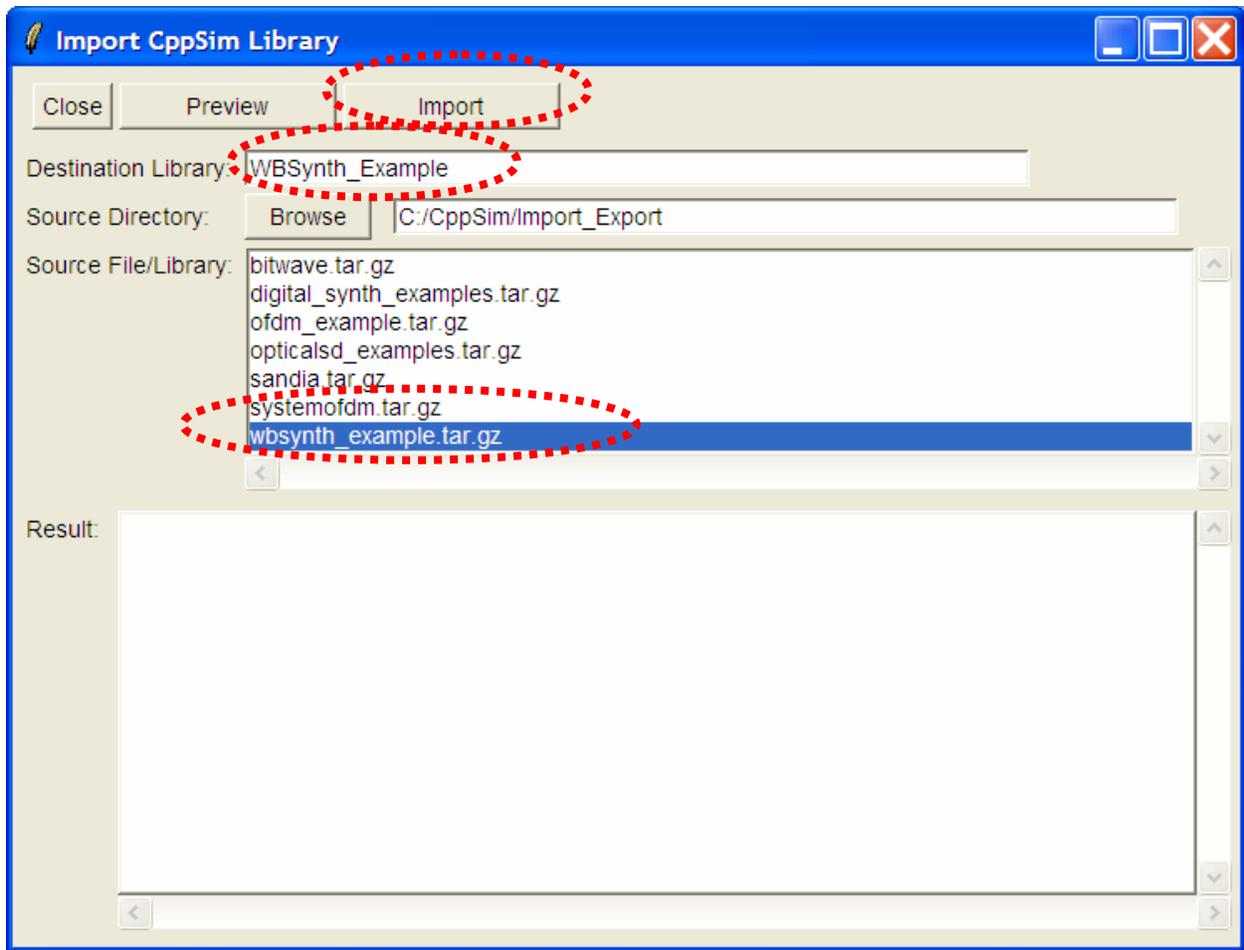
To run this tutorial, you will also need to download the file **wbsynth_example.tar.gz** available at <http://www.cppsim.com>, and place it in the **Import_Export** directory of CppSim (assumed to be **c:/CppSim/Import_Export**). Once you do so, start up **Sue2** by clicking on its icon, and then click on **Tools->Library Manager** as shown in the figure below.



In the **CppSim Library Manager** window that appears, click on the **Import Library Tool** button as shown in the figure below.



In the **Import CppSim Library** window that appears, change the **Destination Library** to **WBSynth_Example**, click on the **Source File/Library** labeled as **wbsynth_example.tar.gz**, and then press the **Import** button as shown in the figure below. Note that if **wbsynth_example.tar.gz** does not appear as an option in the **Source File/Library** selection listbox, then you need to place this file (downloaded from <http://www.cppsim.com>) in the **c:/CppSim/Import_Export** directory.



Once you have completed the above steps, restart **Sue2** as directed in the above figure.

Introduction

This tutorial explores the design of a wideband fractional-N frequency synthesizer using the CppSim design tool. Wideband synthesizers are very desirable in a variety of applications, including fast-hopping local oscillator (LO) generation and direct-modulated transmitters. Up to now, fractional-N synthesizer bandwidths have typically been limited to bandwidths below 100kHz because of the impact of fractional-N dithering noise on the output phase noise spectrum. By introducing a new circuit element, the mismatch compensated PFD/DAC [1], we are able to substantially reduce the magnitude of fractional-N quantization noise, thereby making high bandwidth (> 1MHz) fractional-N synthesis possible.

The synthesizer architecture is described in detail in [1, 2]. This tutorial will focus on simulation and experimentation with relevant design variables. However, a brief explanation of the architectural advantage of the synthesizer follows below.

For an introductory tutorial to fractional-N synthesis, we recommend first running the tutorial, described in “Fractional-N Frequency Synthesizer Design Using The PLL Design Assistant and CppSim Programs” available at

[1] S. Meninger and M. Perrott, "A Fractional-N Frequency Synthesizer Architecture Utilizing a Mismatch Compensated PFD/DAC Structure for Reduced Quantization-Induced Phase Noise", IEEE Trans. Circuits and Systems II, Nov 2003

[2] S. Meninger, "High Bandwidth, Low Noise Frequency Synthesis Techniques", PhD Thesis, Massachusetts Institute of Technology, May 2005

A. Quantization Noise Impact on Synthesizer Performance

Figure 1. depicts the architecture use by state-of-the-art $\Sigma\Delta$ fractional-N synthesizers, the architecture typically chosen in high performance applications. A digital $\Sigma\Delta$ modulator is used to dither the divide value between integer values such that, *on average*, a fractional divide value is produced. The dithering process introduces quantization noise into the system, a result that is intuitive when we consider that a fractional divide value is obtained by dithering between integer values. So while the average divide value is fractional as desired, and results in zero average phase error detected by the PFD circuitry, instantaneous errors are produced. These instantaneous errors represent the quantization noise.

The $\Sigma\Delta$ modulator dithering action results in a quantization noise profile that is shaped. In other words, the low frequency component of the shaped quantization noise is suppressed relative to the high frequency component. The shaped noise is filtered by the low pass filter response associated with the synthesizer closed loop dynamics. The result is an undesirable noise-bandwidth tradeoff, where more shaped quantization noise appears at the output as closed loop synthesizer bandwidth is increased.

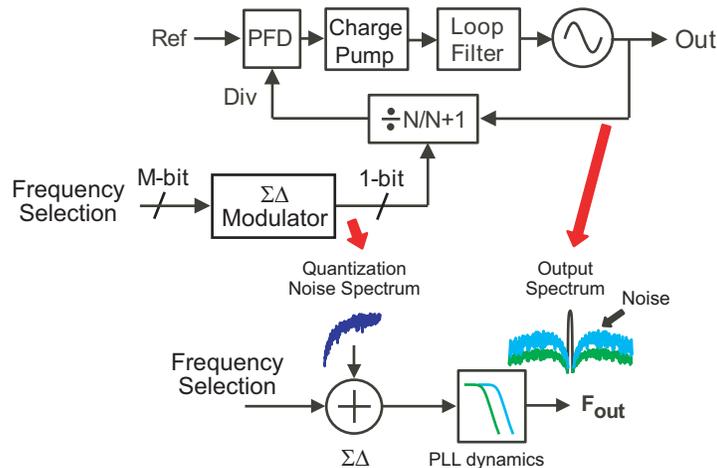


Figure 1: $\Sigma\Delta$ Fractional-N Synthesizer

B. Wideband Synthesizer Architecture

The quantization noise-bandwidth tradeoff associated with $\Sigma\Delta$ fractional-N synthesis can be removed if quantization noise can be reduced. In order to achieve noise reduction, we propose the architecture

depicted in Figure 2 [1, 3]. This topology uses a cancellation path to actively remove the quantization noise, whereas a $\Sigma\Delta$ synthesizer relies on the PLL dynamics to attenuate quantization noise.

Active noise cancellation is not a new concept. Phase interpolation based fractional-N synthesis employs a cancellation DAC which attempts to actively cancel quantization noise at the charge-pump output [4]. The main limitation of phase interpolation is due to the mismatch that exists between the cancellation signal and noise signal. For more details, please see [2].

The key advantage of the proposed architecture is that the circuitry that injects quantization noise into the loop (the PFD and charge-pump) is combined with the cancellation signal (a DAC) to create an inherent gain match between the two signals [3]. We denote the overall hybrid structure as a PFD/DAC to emphasize that the phase detection and noise cancellation are performed in the same circuitry. As we will see through simulation, mismatches internal to the PFD/DAC limit performance if left unchecked. By using dynamic element matching techniques, these mismatches can be converted into broadband phase noise that is filtered by the PLL dynamics. The resulting mismatch compensated PFD/DAC synthesizer is capable of dramatically reducing quantization induced phase noise, as detailed in [1, 2].

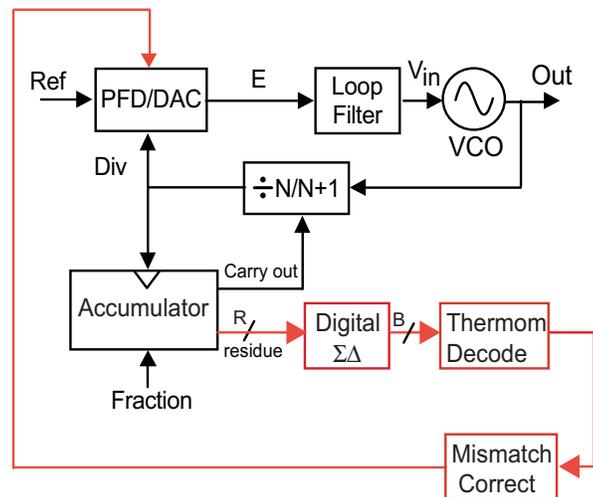


Figure 2: Wideband Fractional-N Synthesizer Architecture

[3] Y. Dufour, “Method and Apparatus for Performing Fractional Division Charge Compensation in a Frequency Synthesizer”, US Patent No. 6,130,561 2000

[4] W. Egan, “Frequency Synthesis by Phase-lock”, Wiley Press, 2000

C. Qualitative Explanation of PFD/DAC Operation: The Charge Box

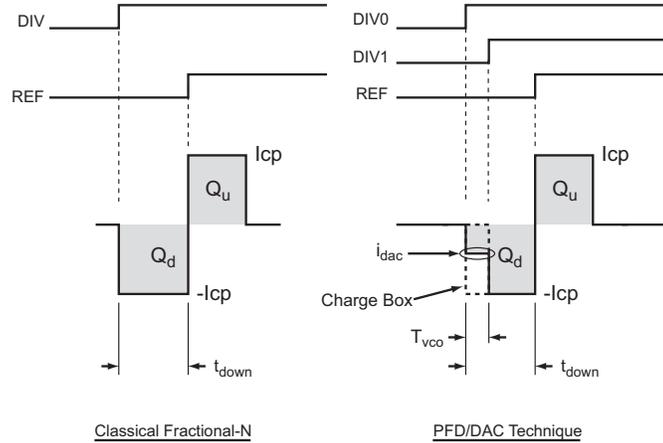


Figure 3: The Charge-Box

Figure 3 presents a simple qualitative explanation of operation of the PFD/DAC. Shown in the figure are outputs of a typical PFD and charge-pump configuration for both a classical fractional-N synthesizer as well as a PFD/DAC synthesizer. The classical fractional-N synthesizer achieves an *average* divide value that is fractional, as desired, but exhibits *instantaneous* phase errors on a period-by-period basis.

Shown in the figure is the output of the charge-pump for one particular period where an instantaneous error is produced. The error occurs because the fractional-N dithering process causes the divider phase to vary over time, which changes the width of the negative charge-pump output (t_{down}). The width of the positive charge-pump output is determined by the reference, whose phase is unchanging in time, and is therefore constant in time as well. The result is a net *instantaneous* charge imbalance, $Q_d \neq Q_u$. Again, we must emphasize that we are showing a single phase comparison. The classical fractional-N synthesizer will achieve steady-state operation where the average error charge is zero, but, on a period-by-period basis as we show in Figure 3, instantaneous errors are produced. These errors move the VCO control voltage and result in spurs. The instantaneous errors in a fractional-N synthesizer are related to the fractional divide value, and so the spurs occur at frequencies related to the fractional portion of the divide value, and are denoted as *fractional spurs*.

$\Sigma\Delta$ fractional-N synthesis causes the errors to be randomized rather than periodic, and so the instantaneous errors do not result in spurs, but rather result in shaped broadband noise. The shaped noise results in the noise-bandwidth tradeoff already discussed.

The PFD/DAC approach actively cancels the quantization noise by creating a charge-box which compensates for the variation in t_{down} by controlling the current in a one-VCO period wide window. The charge-box is created by using two divider signals in the PFD logic (Div1 is delayed by one VCO period from Div0, as indicated in the figure). The net result is that a constant amount of negative charge is delivered every period, and instantaneous errors are removed! A one-VCO period wide window is used, because the variation of t_{down} is referenced to a single VCO period [1]. The magnitude of current is controlled by the divider control $\Sigma\Delta$ residue, since it contains information about the magnitude of the quantization error [4].

There are two key observations to make about the PFD/DAC technique. The first is that the degree to which the quantization error is removed is related to the resolution of the DAC current. The second observation is that, as the DAC current varies period-to-period to compensate for variations in t_{down} , there is some information about the fractional divide value in the varying shape of the negative current pulse, so there will be some small amount of fractional spur information present. We will show how employing a sample-and-hold can eliminate this spurious content in the PFD/DAC output.

Design Goals for an Example PFD/DAC Synthesizer

As a target application for the PFD/DAC synthesizer, we set the following specifications:

- PLL specifications
 - 1MHz closed loop synthesizer bandwidth
 - Order: 2 (We want a simple implementation)
 - Filter Shape: Butterworth
 - Parasitic Pole at 2.5MHz (helps to attenuate high frequency noise. This is a standard noise reduction “trick” used in synthesizer design)
 - Type: 2 with $f_z/f_o = 1/9$
 - 3.6GHz output frequency
 - 50MHz reference frequency

- Noise Specifications
 - No worse than -100dBc/Hz in-band noise. We begin assuming a -110 dBc/Hz in-band phase noise level. (In-band noise is noise within the loop bandwidth)
 - -155dBc/Hz VCO phase noise at 20MHz offset from the carrier
 - Minimal residual spurs present in the output. For practical purposes, we will aim for spurs no larger than -80dBc/Hz

These design goals would allow the synthesizer to be used as a direct modulated GSM transmitter if the VCO output is divided by four. Having a 3.6GHz output frequency results in a large degree of quantization noise cancellation relative to 900MHz, because the VCO period is four times smaller at 900MHz than at 3.6GHz.

Below is the GSM 900MHz phase noise mask specifications, as well as the equivalent noise performance required by a 3.6GHz signal that is divided down to generate the 900MHz signal, assuming a noiseless division is performed.

	100	200	250	400	600	1.8	3.0	6.0	10	20
--	-----	-----	-----	-----	-----	-----	-----	-----	----	----

	kHz	kHz	kHz	kHz	kHz	MHz	MHz	MHz	MHz	MHz
900 MHz	-52.3 dBc/Hz	-82.8 dBc/Hz	-85.8 dBc/Hz	-112.8 dBc/Hz	-112.8 dBc/Hz	-121 dBc/Hz	-123 dBc/Hz	-129 dBc/Hz	-150 dBc/Hz	-162 dBc/Hz
3.6 GHz	-40.3 dBc/Hz	-70.8 dBc/Hz	-73.8 dBc/Hz	-100.8 dBc/Hz	-100.8 dBc/Hz	-109 dBc/Hz	-111 dBc/Hz	-117 dBc/Hz	-138 dBc/Hz	-150 dBc/Hz

Performing Basic Noise Analysis Using the PLL Design Assistant

Open the PLL Design Assistant tool and put in the above parameter values. Click on the Noise Plot radio button to see the estimated phase noise of the synthesizer. Now set the axis values of the noise plot to:

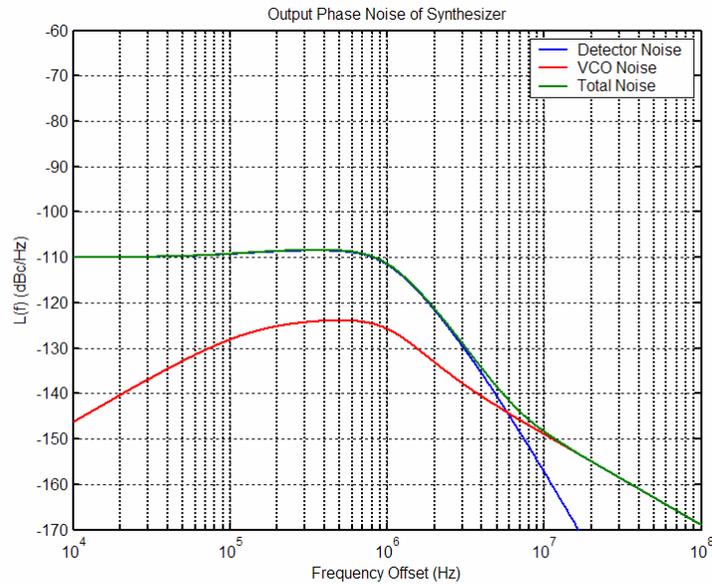
- Noise axis limits: **1e4 1e8 -170 -60**

As a check to your work, the figures below illustrate what the PLL Design Assistant and resulting phase noise plot should look like.

The screenshot shows the PLL Design Assistant interface with the following settings:

- Dynamic Parameters:**
 - fo: 1e6 Hz
 - order: 2
 - shape: Butter
 - ripple: dB
 - type: 2
 - fz/fo: 1/9
- Noise Parameters:**
 - ref. freq: 50e6 Hz
 - out freq.: 3.6e9 Hz
 - Detector: -110 dBc/Hz
 - VCO: -155 dBc/Hz
 - freq. offset: 20e6 Hz
 - S-D: 2
- Resulting Open Loop Parameters:**
 - K: 2.885e+012
 - fp: 2.807e+006 Hz
 - fz: 1.111e+005 Hz
 - Qp:
- Resulting Plots and Jitter:**
 - Selected plot: Noise Plot
 - Axis limits: 1e4, 1e8, -170, -60
 - rms jitter: 245.306 fs

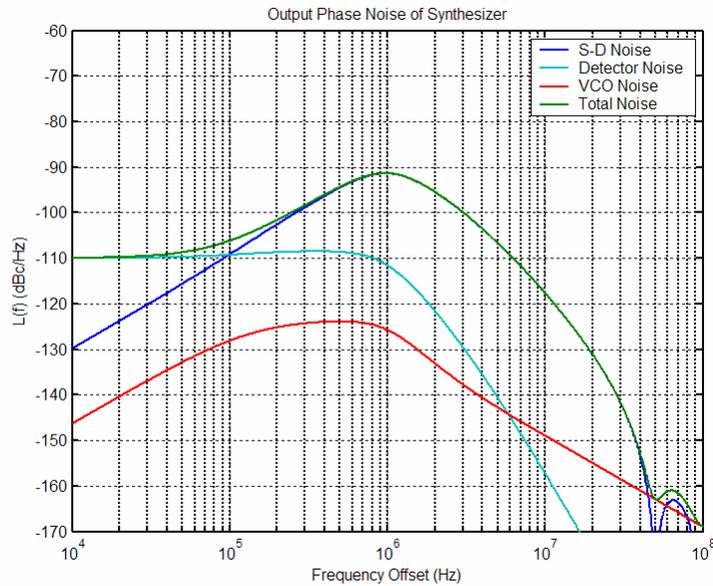
----- PLL Design Assistant -----
 Written by Michael Perrott (<http://www-mtl.mit.edu/~perrott>)



- Note that there is something missing from the noise plot. Namely, the fractional-N quantization noise! To see the impact of the quantization noise, we modify the PLL Design Assistant window. We first look at the performance of a 2nd order $\Sigma\Delta$ synthesizer with a 1MHz bandwidth that otherwise meets all of the specifications.
- In the PLL Design Assistant window, enter a 2nd order noise transfer function into the S-D noise box by first clicking the "On" button and entering [1 -2 1] into the parameter box, as depicted below.

Dynamic Parameters		Noise Parameters	
fo: 1e6 Hz	paris. pole: 2.5e6 Hz	ref. freq: 50e6 Hz	out freq: 3.6e9 Hz
order: 2	paris. Q: []	Detector: -110 dBc/Hz	VCO: -155 dBc/Hz
shape: Butter	paris. pole: [] Hz	freq. offset: 20e6 Hz	S-D: 2 On [1 -2 1] On
ripple: [] dB	paris. Q: []		
type: 2	paris. pole: [] Hz		
fz/fo: 1/9	paris. pole: [] Hz		
	paris. zero: [] Hz		
	paris. zero: [] Hz		
Resulting Open Loop Parameters		Resulting Plots and Jitter	
K: 2.885e+012	alter: [] On	<input type="radio"/> Pole/Zero Diagram	<input type="radio"/> Transfer Function
fp: 2.807e+006 Hz	alter: [] On	<input type="radio"/> Step Response	<input checked="" type="radio"/> Noise Plot
fz: 1.111e+005 Hz	alter: [] On	1e4	1e8
Qp: []	alter: [] On	-170	-60
		rms jitter: 2.168 ps	
----- PLL Design Assistant -----		Written by Michael Perrott (http://www-mtl.mit.edu/~perrott)	

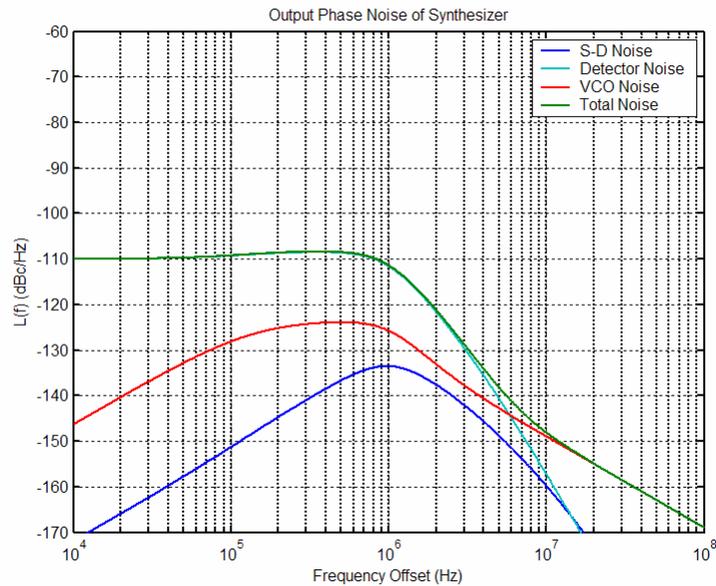
- The resulting noise plot should appear as below.



- Note that the quantization noise associated with the 2nd order $\Sigma\Delta$ synthesizer ruins noise performance! The standard way to combat this noise would be to reduce the synthesizer bandwidth. You can experiment with this using the PLL Design Assistant. We will not do so here because reducing bandwidth violates our stated goal of achieving 1MHz bandwidth.
- Now change the quantization noise function so that it corresponds to a 7-bit PFD/DAC synthesizer. This is accomplished by dividing the noise transfer function by 2^7 , as shown below:

Dynamic Parameters		Noise Parameters	
fo: <input type="text" value="1e6"/> Hz	paris. pole: <input type="text" value="2.5e6"/> Hz <input type="checkbox"/> On	ref. freq: <input type="text" value="50e6"/> Hz	
order: <input type="radio"/> 1 <input checked="" type="radio"/> 2 <input type="radio"/> 3	paris. Q: <input type="text"/> <input type="checkbox"/> On	out freq: <input type="text" value="3.6e9"/> Hz	
shape: <input checked="" type="radio"/> Butter <input type="radio"/> Bessel	paris. pole: <input type="text"/> Hz <input type="checkbox"/> On	Detector: <input type="text" value="-110"/> dBc/Hz <input type="checkbox"/> On	
<input type="radio"/> Cheby1 <input type="radio"/> Cheby2 <input type="radio"/> Elliptical	paris. Q: <input type="text"/> <input type="checkbox"/> On	VCO: <input type="text" value="-155"/> dBc/Hz <input type="checkbox"/> On	
ripple: <input type="text"/> dB	paris. pole: <input type="text"/> Hz <input type="checkbox"/> On	freq. offset: <input type="text" value="20e6"/> Hz	
type: <input type="radio"/> 1 <input checked="" type="radio"/> 2	paris. pole: <input type="text"/> Hz <input type="checkbox"/> On	S-D: <input type="radio"/> 1 <input type="radio"/> 2 <input type="checkbox"/> On	<input type="text" value="[1 -2 1]/2^7"/> <input type="checkbox"/> On
fz/fo: <input type="text" value="1/9"/>	paris. zero: <input type="text"/> Hz <input type="checkbox"/> On	<input type="radio"/> 3 <input type="radio"/> 4 <input type="radio"/> 5	
	paris. zero: <input type="text"/> Hz <input type="checkbox"/> On		
Resulting Open Loop Parameters		Resulting Plots and Jitter	
K: <input type="text" value="2.886e+012"/> alter: <input type="text"/> <input type="checkbox"/> On		<input type="radio"/> Pole/Zero Diagram	<input type="radio"/> Transfer Function
fp: <input type="text" value="2.807e+006"/> Hz alter: <input type="text"/> <input type="checkbox"/> On		<input type="radio"/> Step Response	<input checked="" type="radio"/> Noise Plot
fz: <input type="text" value="1.111e+005"/> Hz alter: <input type="text"/> <input type="checkbox"/> On		<input type="text" value="1e4"/>	<input type="text" value="1e8"/>
Qp: <input type="text"/> alter: <input type="text"/> <input type="checkbox"/> On		<input type="text" value="-170"/>	<input type="text" value="-60"/>
		rms jitter: <input type="text" value="245.882 fs"/>	
----- PLL Design Assistant -----		Written by Michael Perrott (http://www-mtl.mit.edu/~perrott)	

- The resulting phase noise plot demonstrates that, if the quantization noise can be reduced by 42dB using a 7-bit PFD/DAC, it is effectively removed from consideration when calculating the output noise. The plot below shows this improved phase noise performance.



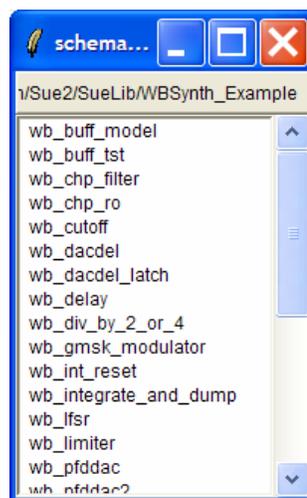
Having established the desired noise performance of the example PFD/DAC synthesizer, we move on to behavioral simulation using CppSim.

Performing Basic Operations within Sue2 and CppSimView

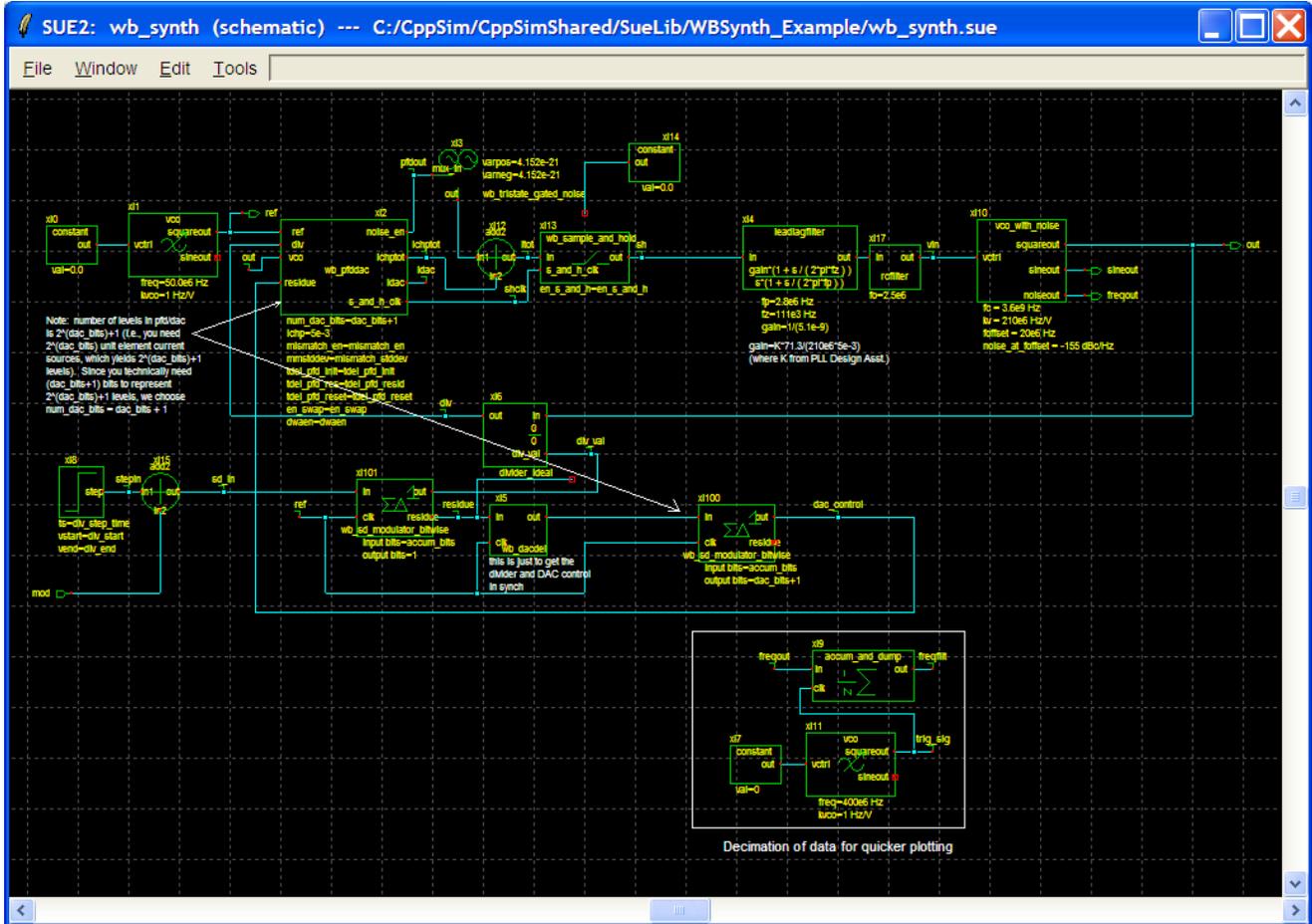
In this section, the user will be guided through basic tasks such as opening the wideband synthesizer example within the Sue2 schematic editor and running CppSim simulations.

A. Opening Sue2 Schematics

- Click on the Sue2 icon to start Sue2, and then select the **WBSynth_Example** library from the **schematic listbox**. The **schematic listbox** should now look as follows:

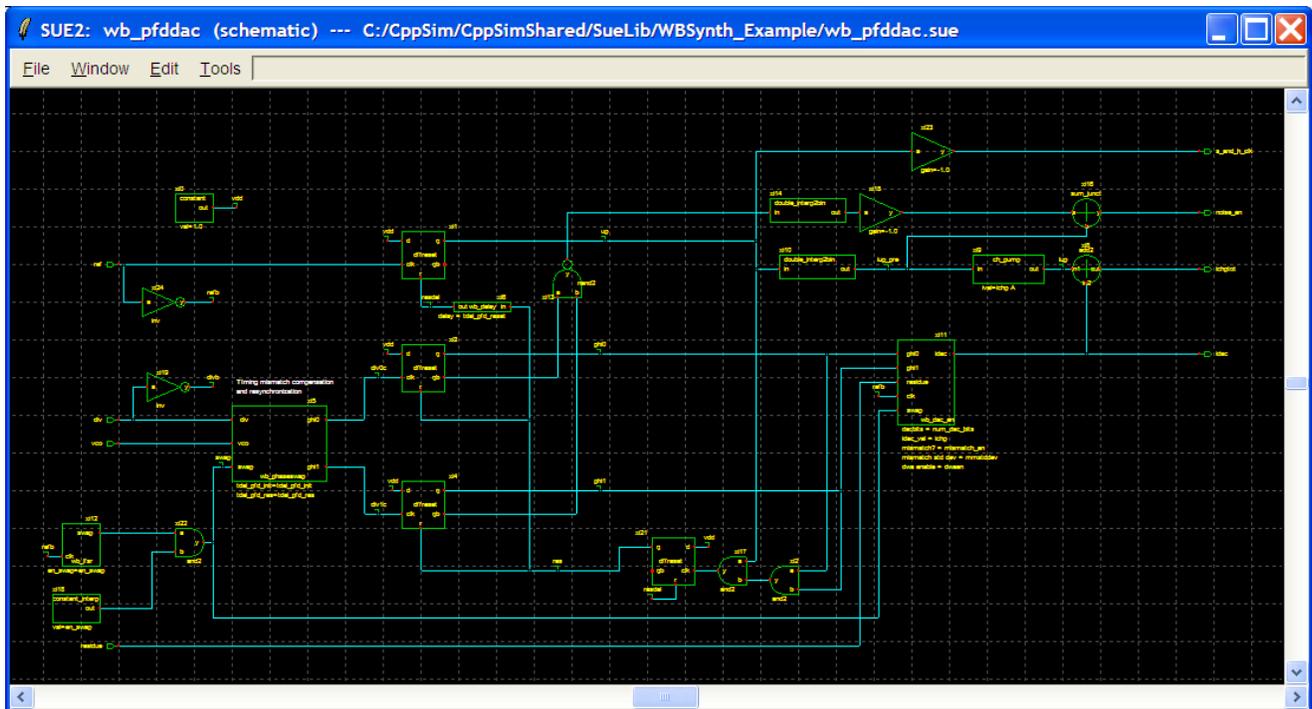


- Scroll down and click on the **wb_synth** schematic. The baseline behavioral model of the wideband synthesizer is shown below:

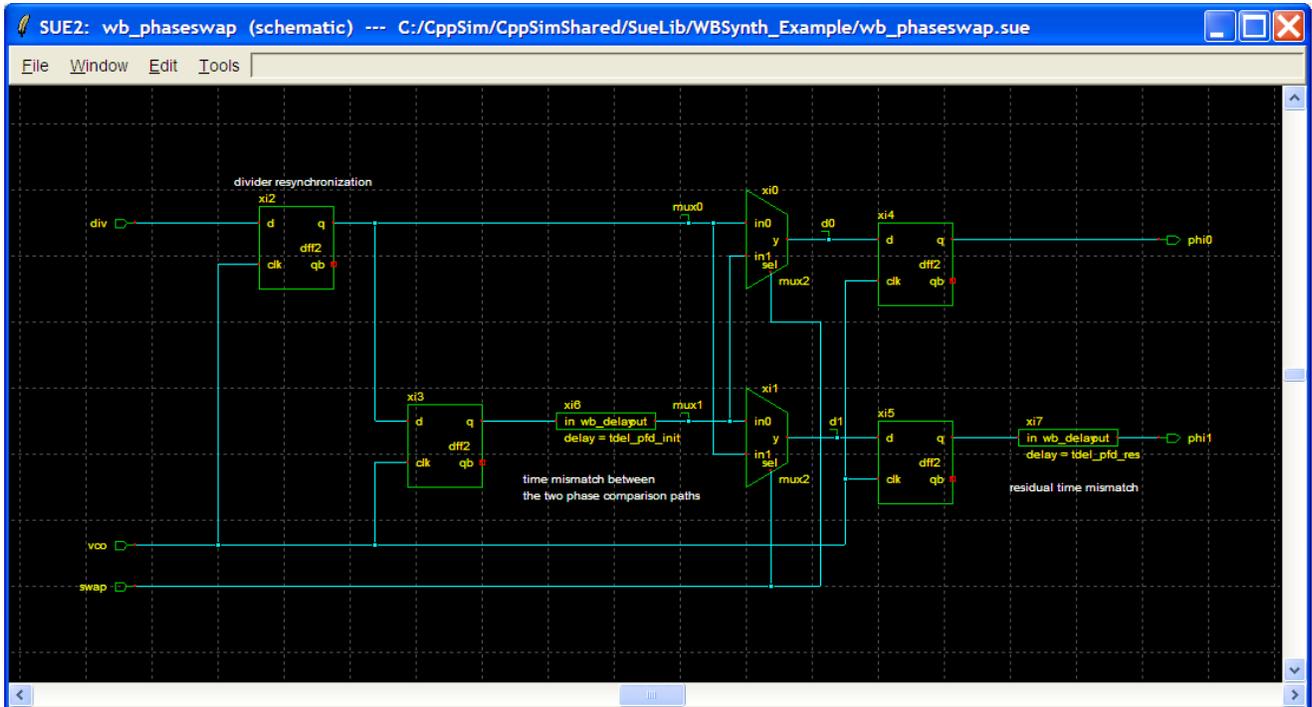


- There are several key blocks in the system:
 - **wb_pfd_dac**: This is the mismatch compensated PFD/DAC structure used to cancel the fractional-N quantization noise
 - **wb_sd_modulator_bitwise**: This is a 1st order $\Sigma\Delta$ modulator block. It is a bitwise model, meaning that it operates in the same manner as a circuit level implementation. A first bitwise modulator is used to control the divider. A second bitwise modulator operates on the least significant bits of the first modulator, which represent the quantization error associated with the first modulator. The output of the second modulator is used to control the DAC function associated with the PFD/DAC.
 - **wb_sh_and_hold**: This is a sample-and-hold block used to sample the output of the PFD/DAC before it is processed by the loop filter. As described in [2], the sample-and-hold helps remove residual spurs at the fractional and reference frequencies that would otherwise be present in the output spectrum.
 - **accum_and_dump**: This block is used to decimate the output data so that very long simulations can be quickly plotted.
- The key signals present in the system are:

- **ichptot**: The PFD/DAC output. We will see how controlling charge within a well-defined time window cancels quantization noise.
 - **freqfilt**: The filtered control voltage used to control the VCO. This signal will be used to calculate and plot the synthesizer output spectrum.
 - **residue**: The residue of the divider control SD modulator that contains information about the quantization noise.
- Select the **wb_pfdac** icon within the **wb_synth** schematic, and then press **e** to descend into it. You will see the schematic shown below. Key signals within this block include:
 - **phi0, phi1**: The mismatch compensated timing signals used to control the noise cancellation window.
 - **idac**: The PFD/DAC output current.



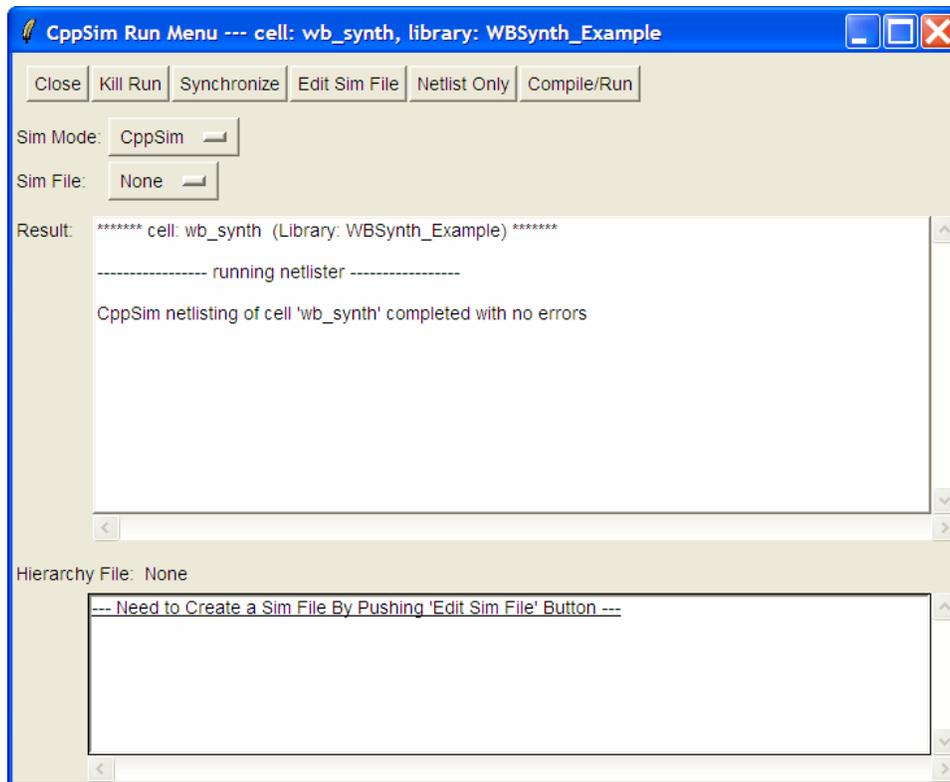
- The final block we will explore before beginning simulation is the timing mismatch compensation block, **wb_phaseswap**. Select the **wb_phaseswap** icon and press **e** to descend into it. You will see the schematic depicted below.



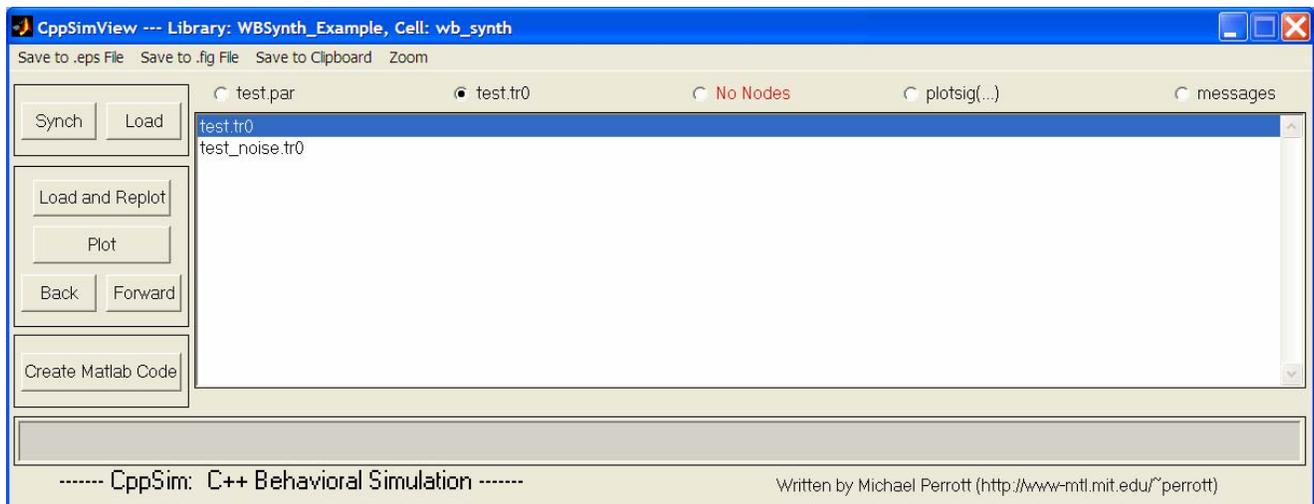
- This block is used to transform a timing mismatch between the two divider signals used to create the charge-box from a gain mismatch into a broadband noise source. We will explore its operation in simulation to follow. Note that any mismatch between phase paths mux0 and mux1 are removed by flip-flops xi4 and xi5. By swapping the paths for mux0 and mux1, we can match the average delay seen by each. In this way, we can create a delay that is one-VCO period wide for use in creating the charge-box.
- Press **Ctrl-e** twice to return to the **wb_synth** schematic.

B. Running the CppSim Simulation

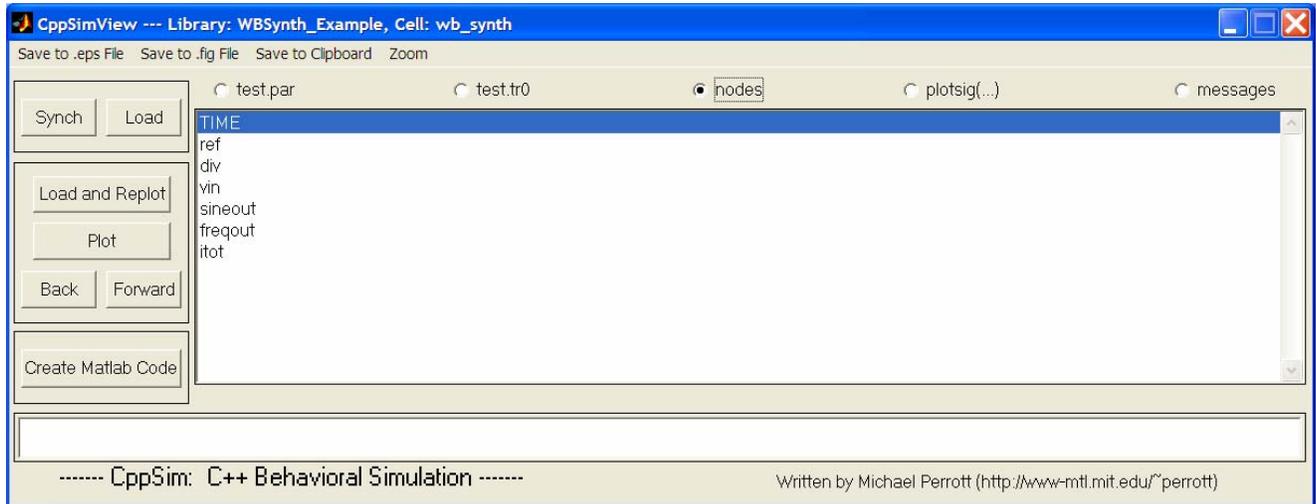
- In the Sue2 schematic window, click on the **Tools** button on the menubar and select **CppSim Simulation** from the drop down menu. The CppSim Run Menu will open in a new window, as shown below – notice that the banner indicates it is currently synchronized to the **wb_synth** cellview.



- Click on the **Edit Sim File** button. An emacs window should appear, displaying the contents of the simulation parameters file (**test.par**). It indicates that the number of simulation steps, **num_sim_steps**, is set to $8e6$ and the timestep, **Ts**, is set to $1/25e9$. You can close the emacs window if you like.
- Click on the **Compile/Run** button to run the simulation.
- When the simulation finishes, start the CppSim viewer by double clicking on the **CppSimView** icon on your desktop, or by launching from the Windows Start button. Once CppSimView loads, Click on the **No Output File** radio button and select test.tr0 as the output file as shown below.



- Click on the **No Nodes** radio button to load in the simulated signals. CppSimView should now appear as shown below.

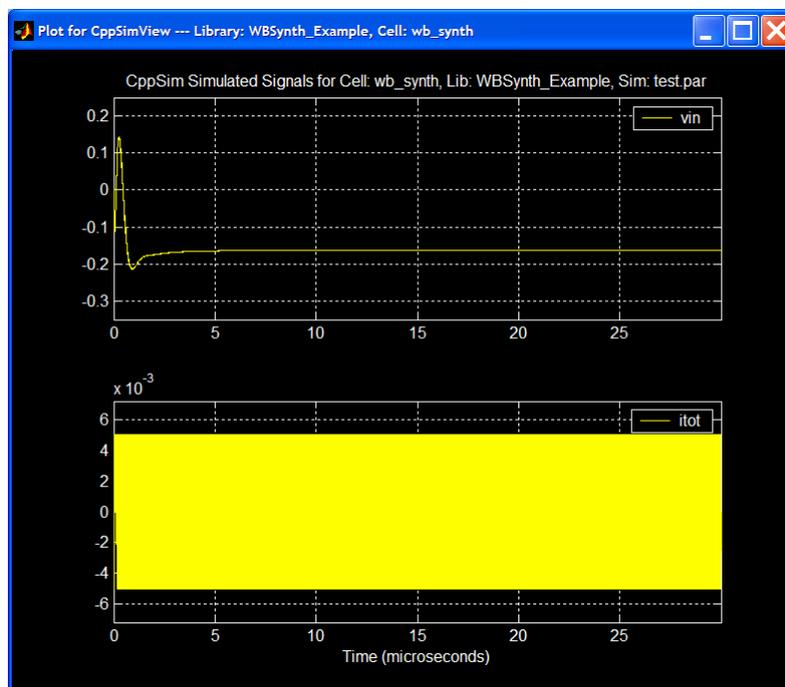


Plotting Time Domain Results

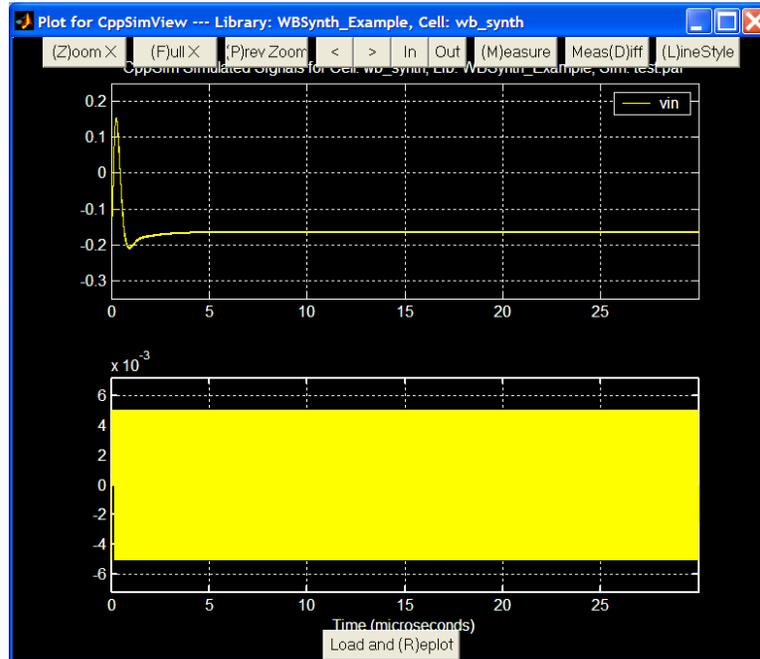
Given the above simulation, we will now examine some of the key signals in the system during normal operation.

A. PFD/DAC Output and VCO control Voltage

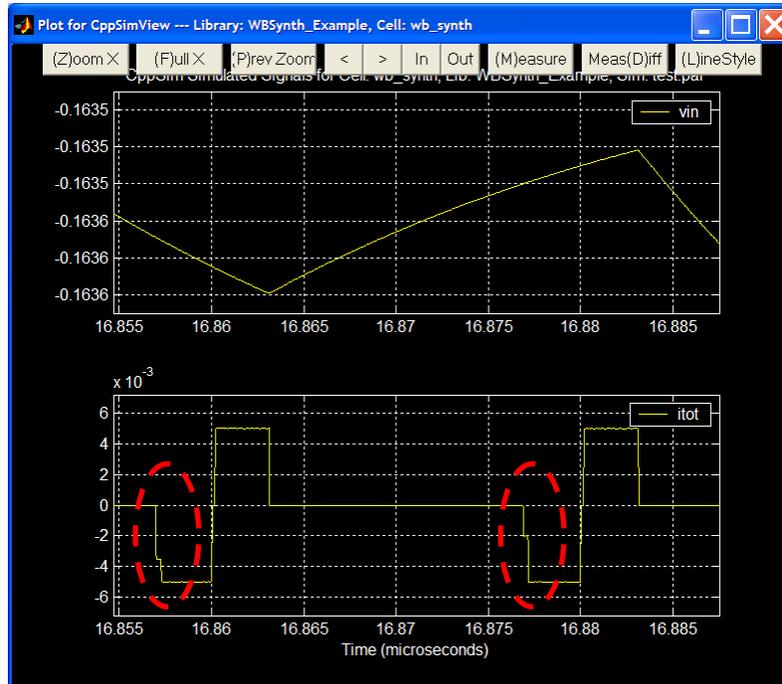
- In the CppSimView window, double-click on signals **vin** and **itot**. You should see the waveforms shown below.



- We see that, after some simulation startup conditions, the VCO control voltage settles to a steady-state value. To observe how “steady” this value is, and to get a better look at the PFD/DAC output waveforms we will zoom in on the waveforms.
- In the CppSimView window, click on **Zoom** (immediately above the test.par radio button). The waveform window will now look as below.



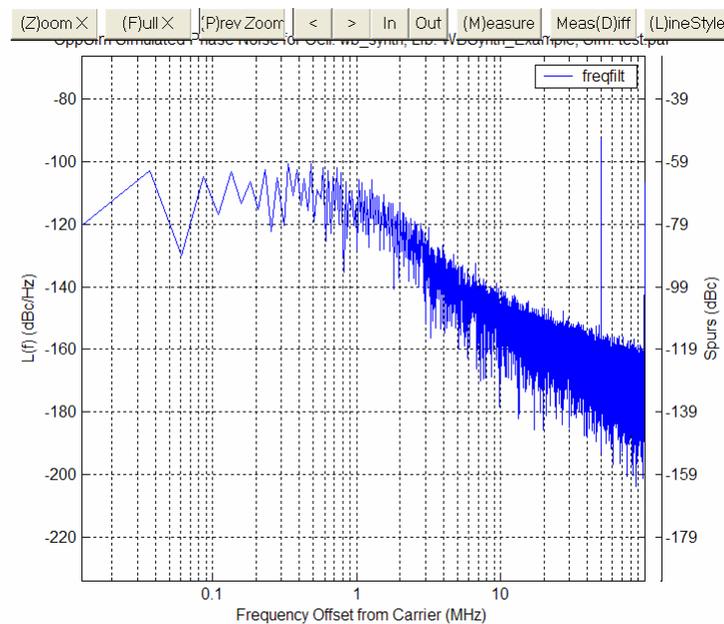
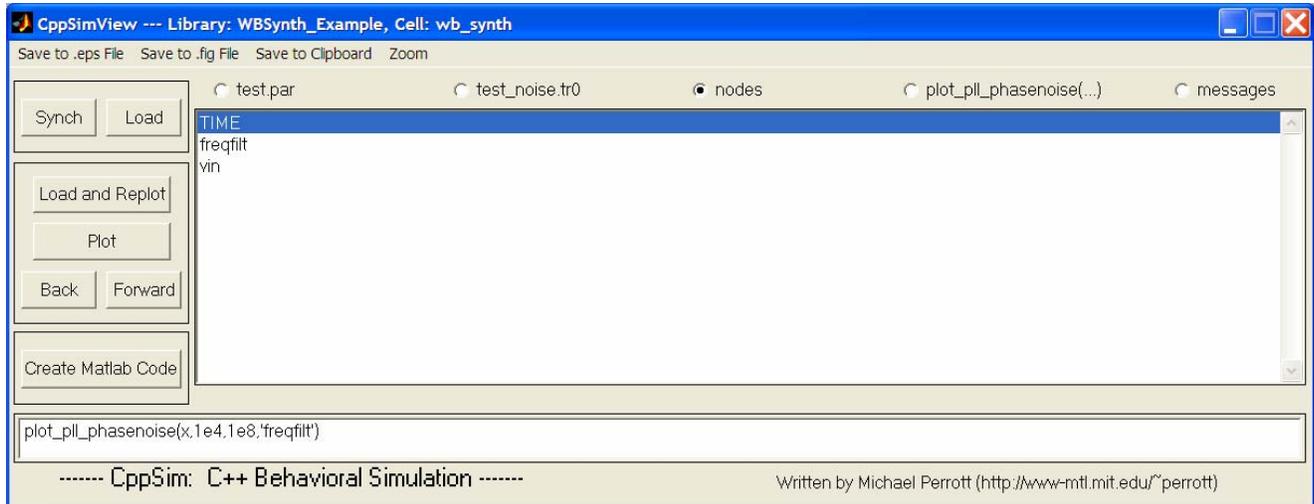
- Click on the **Zoom** button in the waveform window, and then zoom in several times in the 15us to 20us range until you observe two successive periods of operation as shown below:



- Note that the magnitude of $itot$ is changing within a one-VCO period window, as highlighted in the plot. This indicates the PFD/DAC operates as expected. The VCO control voltage is also changing over time, which is due to the random noise sources present in the simulation.

Plotting Frequency Domain Results

- We will now observe the synthesizer output spectrum.
- First, click on the **test.tr0** radio button, and select **test_noise.tr0** from the file list.
- Then click the **No Nodes** radio button.
- Next, Click on the **plotsig(...)** radio button and select **plot_pll_phasenoise(x,f_low,f_high,'nodes')** from the list.
- Next, click the **nodes** radio button.
- Edit the `plot_pll_phasenoise` command so that `f_low=1e4`, `f_high=1e8`, and `'nodes'=freqfilt`. The result should appear as below.

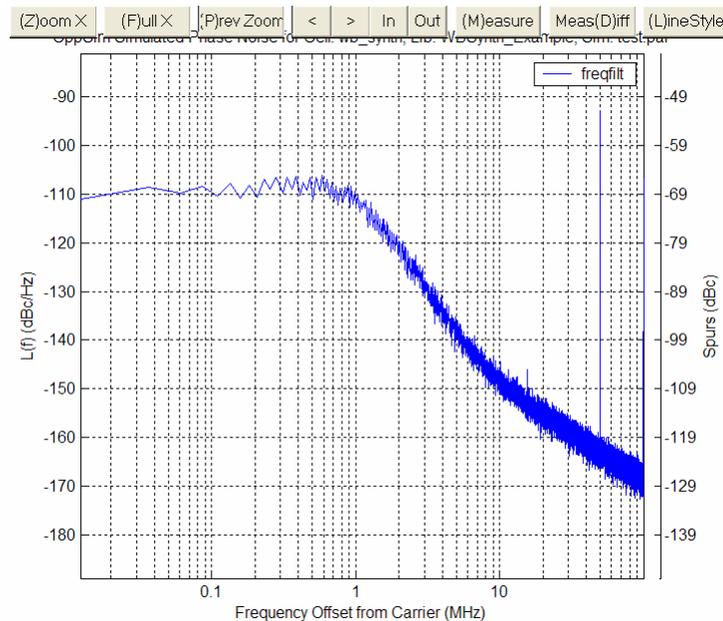


- We observe that the spectrum appears to roughly match the calculated values using the PLL Design Assistant. One difference are the spurs that appear in the behavioral model. One spur is visible at the reference frequency, at a magnitude of -49dBc. We read the spur magnitude from the right y-axis scale in the plot. This spur will have to be reduced to meet our desired specification of no spurs greater than -80dBc.

One thing we note from the plot is that the phase noise spectrum appears very noisy. This is because of the length of the simulation run. In order to generate a less noisy output spectrum, we can run a longer simulation.

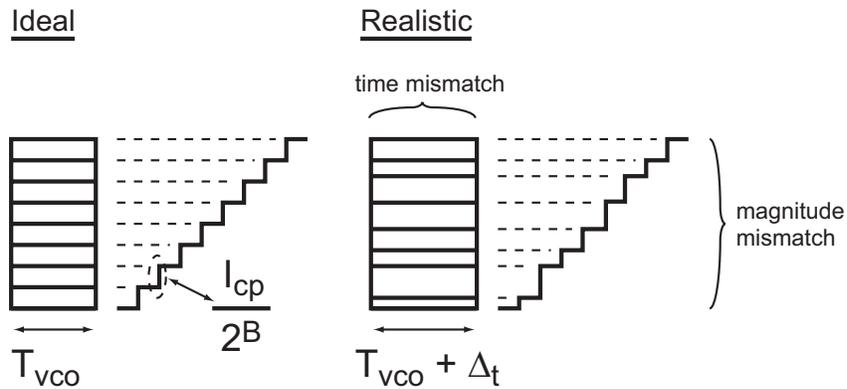
- In the CppSim Run Menu, click on the **Edit Sim File** button. (This assumes the simulation file has been closed from earlier in the tutorial. If it is already open, skip this step.)
- Change the **num_sim_steps** variable to 50e6
- Save the file by typing **Ctrl-x Ctrl-s**. You can then close the simulation file if you like.

- In the CppSim Run Menu, click the **Compile/Run** button.
- Once the simulation is complete, click on the **Load and Replot** button in the CppSimView Window. The resulting plot is shown below. We see that increasing the simulation length reduces the “noisy-ness” of the phase noise calculation. Comparing the simulation result with our calculated result reveals that the two have excellent agreement. For simulations from now on, we will use shorter simulation runs to reduce simulation time.
- We note from the longer simulation that there is also a spur occurring at 15.6535 MHz, which corresponds to the fundamental fractional spur frequency. (The nominal divide value set in the sim file is 71.31307, which results in a fundamental spur at $0.31307 \cdot 50\text{MHz} = 15.6535\text{MHz}$.) This spur occurs due to the changing PFD/DAC output wave-shape, as described previously, and is very small, below -100dBc in magnitude. A compensation scheme for this spur will be simulated later in this tutorial.



Exploring Non-Idealities within the PFD/DAC

In this section, we explore non-idealities associated with the PFD/DAC, and observe their impact on overall noise performance. The non-ideal charge-box is compared to an ideal charge-box in the figure below.



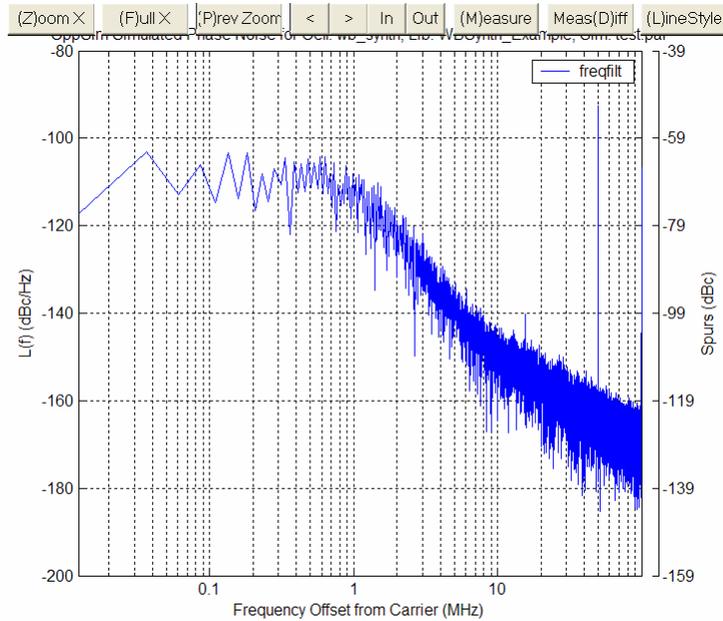
Ideal Vs. Non-Ideal Charge-box

The two primary sources of mismatch within the PFD/DAC are magnitude mismatch between the unit element current sources and timing mismatch between the two phase paths that generate the charge-box.

A. PFD/DAC Unit Element Mismatch

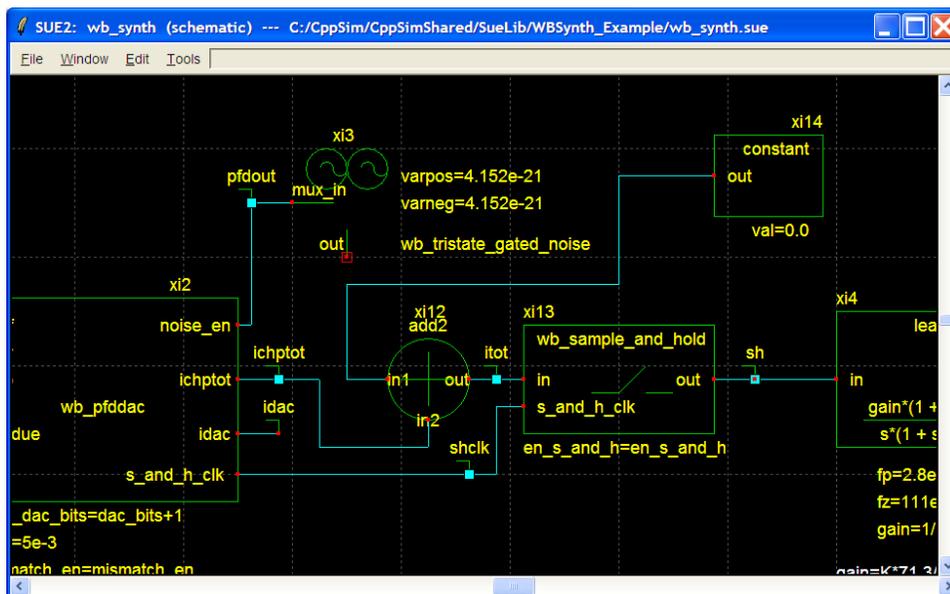
As detailed in [1, 2], the PFD/DAC is comprised of 2^B unit elements, where B is the number of bits in the PFD/DAC. Mismatch between the unit elements result in a gain mismatch between the quantization noise signal and cancellation signal. We explore the impact of this mismatch in the behavioral model.

- If the simulation control file is not open, open it by clicking the **Edit Sim File** button in the CppSim Run Menu.
- Turn on unit element mismatch by changing the **mismatch_en** control parameter from -1.0 (disabled) to 1.0 (enabled).
- The mismatch is set using a Gaussian profile that has a user specified standard deviation. The standard deviation is set by the **mismatch_stddev** control parameter. Leave it set to the default value of 0.05, which corresponds to a standard deviation of 5% mismatch between the unit elements, a conservative value.
- Change the **num_sim_steps** parameter to 10e6 to reduce the overall simulation runtime. (If you want to return later to run the sim again with more points to get a “cleaner” phase noise plot, you are encouraged to do so. However, your phase noise plot won’t match the tutorial results exactly because of the difference in simulation time.)
- Save the changes to the simulation control file.
- Click the **Compile/Run** button in the CppSim Run Menu to start the simulation.
- Load in the simulation results and plot the output phase noise plot following the same procedure previously described in this tutorial. You should see the following phase noise plot:



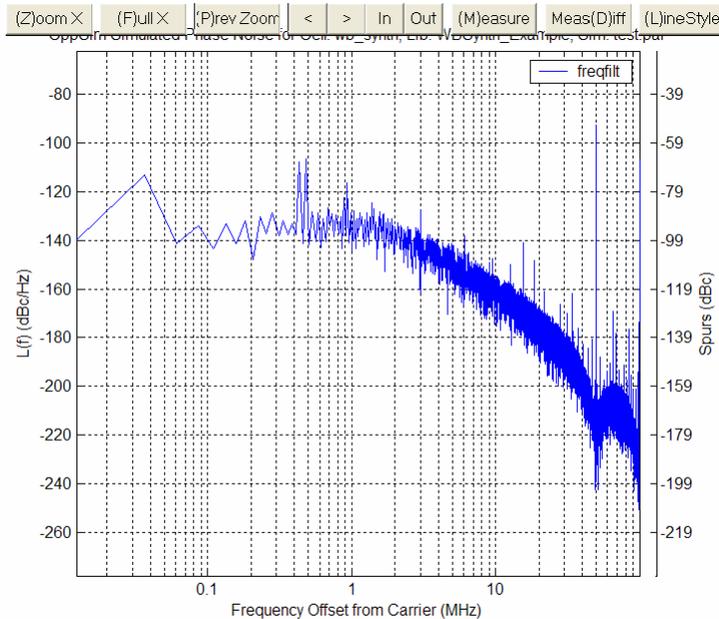
We see that unit element mismatch appears to have minimal impact on phase noise performance. Is this completely accurate? To check, we will re-run the simulation, but “turn off” other noise sources in the system to isolate the mismatch noise. To do this, we edit the schematic view of **wb_synth**.

- Disconnect the charge-pump noise source, instance **xi3**, from the summing junction, instance **xi12**, and connect the 0-valued constant source, **xi14**, to the summing junction input, as shown below. This removes the charge-pump noise from affecting the PLL output.



- Change the value of the VCO noise at 20MHz from -155dBc/Hz to -1550dBc/Hz so that it is, effectively, out of the picture.

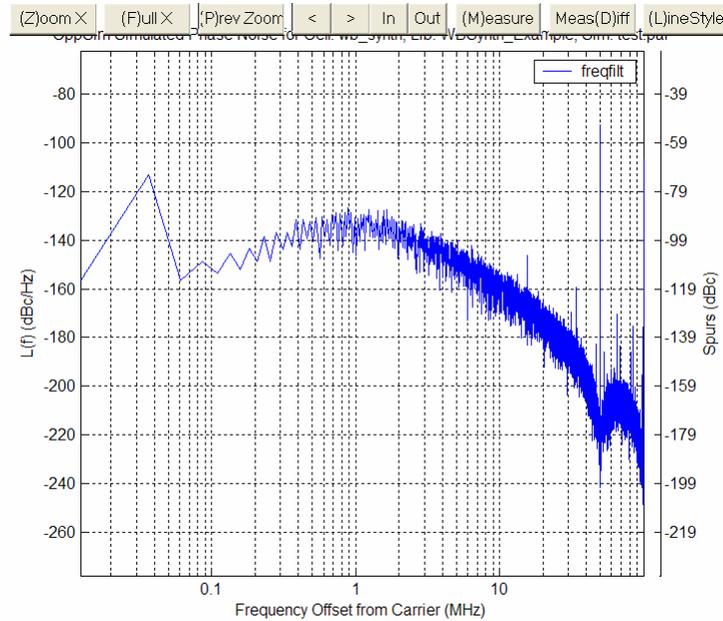
- Run the simulation by clicking the **Run CppSim** button in the CppSimView window.
- Plot the output phase noise, and observe the plot, shown below.



This simulation result shows that unit element mismatch results in a significant spurious component to the remaining quantization noise profile. Depending on the magnitude of unit element mismatch and particular divide value, it is possible that these spurs will be significant, and adversely affect synthesizer performance. The largest spur shown in the plot above is at approximately -64dBc, which is outside our desired specification. This simulation reveals that the ability to selectively turn off noise sources is a useful advantage to behavioral simulation because it allows use to examine noise sources individually.

To combat unit element mismatch, the PFD/DAC uses data weighted averaging (DWA), a technique widely used by $\Sigma\Delta$ DACs, to randomize selection of the unit elements. In order to see how this dynamic element matching technique improves performance, we run another simulation with it enabled.

- In the simulation control file, enable dynamic element matching of the PFD/DAC unit elements by changing the **dwaen** parameter from -1.0 (disabled) to 1.0 (enabled).
- Save your changes to the control file.
- Run the simulation by clicking the **Run CppSim** button in the CppSimView window.
- Plot the phase noise profile. You should obtain the plot shown below.

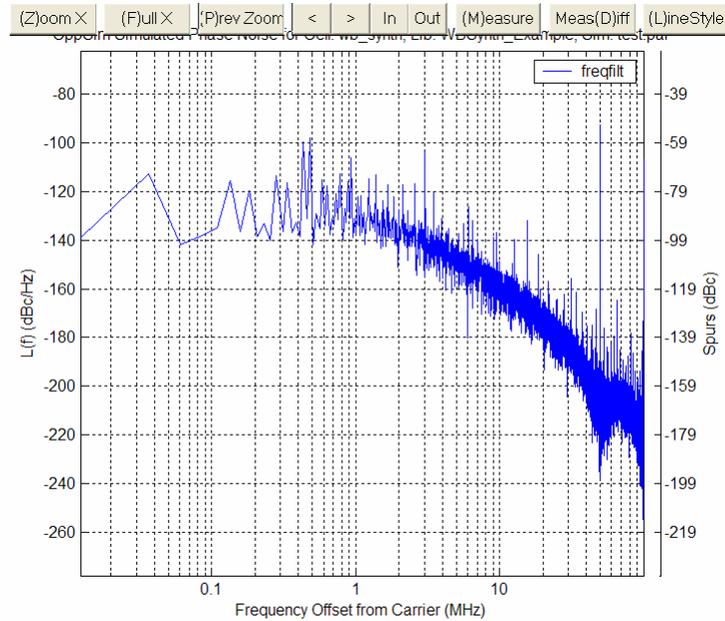


Note that the dynamic element matching technique not only eliminated the spurs created by unit element mismatch, but also results in a shaped mismatch noise profile. This is beneficial in cases where the mismatch noise is significant relative to the other low frequency noise sources in the system, because the shaped nature provides extra low frequency margins. We also observe that the residual fractional spur at 15.6535MHz is still present, as well as some spurs at harmonics of 15.6535MHz. However, these spurs occur at very small magnitudes, as all are <-100dBc. One final thing to note is that the low frequency deviation in the noise plot occurring below 100kHz is an artifact due to the fact that the quantization noise in that range is very small.

B. Timing Mismatch in the PFD/DAC

As already mentioned, any timing mismatch between the two phase paths that generate the charge-box will adversely affect PFD/DAC performance. In this section, we will add timing mismatch to the simulation and observe its impact on overall synthesizer noise performance. We will leave the unit element mismatch as is, and turn on timing mismatch to see how this added non-ideality affects overall performance.

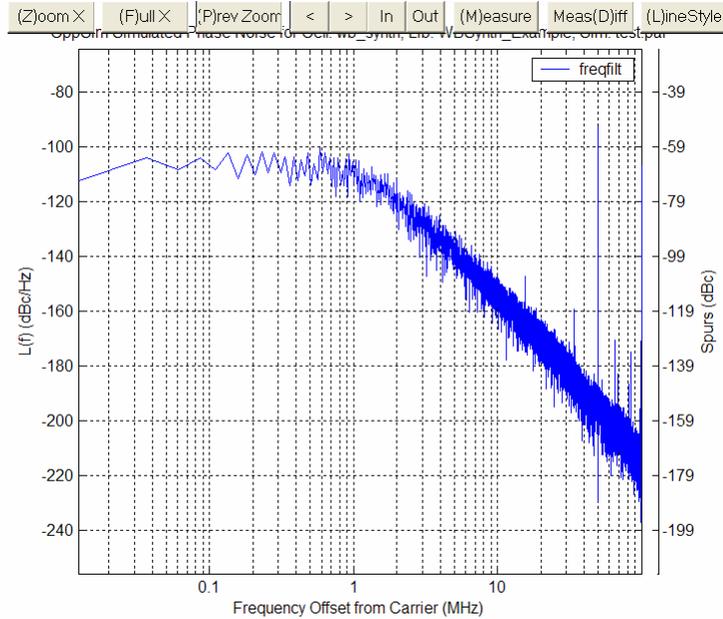
- In the simulation control file, change the **tdel_pfd_resid** parameter from 0.0e12 to 5e-12. This changes the value of the timing mismatch between the phase paths from zero to 5ps.
- Save the changes to the simulation file.
- Run the simulation by clicking the **Compile/Run** button in the CppSim Run Menu.
- Plot the phase noise profile. You should obtain the plot shown below.



We see that timing mismatch also has a very adverse impact on overall synthesizer noise performance. Fractional spurs up to -60dBc are observed.

In order to combat timing mismatch, we enable the timing mismatch compensation block, which dynamically matches the two phase paths in the PFD/DAC by swapping them back-and-forth over time. The swap function is controlled by a linear feedback shift register (LFSR) random number generator that causes the swap signal to have a white noise profile with an average duty cycle = 0.5.

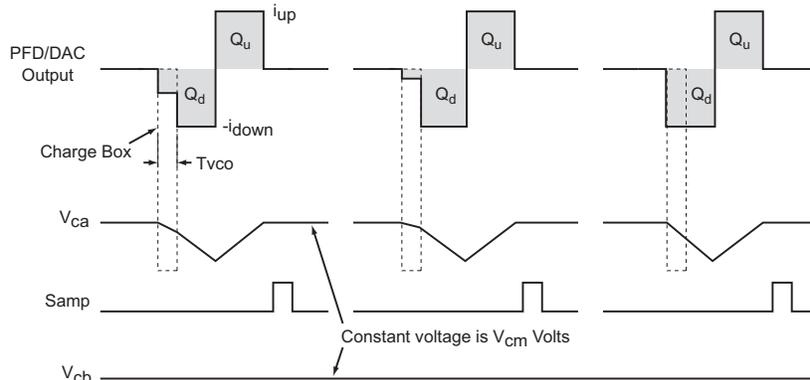
- In the simulation control file, change the **en_swap** parameter from -1.0 (disabled) to 1.0 (enabled).
- Save the changes to the simulation file.
- Run the simulation by clicking the **Compile/Run** button in the CppSim Run Menu window.
- Plot the phase noise profile. You should obtain the plot shown below.



We see that the timing mismatch compensation converts the spurious energy caused by timing mismatch into a broadband noise source. Since the compensated timing mismatch noise has a white low frequency profile, its magnitude must be reduced as much as possible so that low frequency noise performance does not suffer. In the case of the example synthesizer being simulated, the -107dBc/Hz low frequency noise resulting from 5ps timing mismatch is worse than the -110dBc/Hz low frequency noise performance originally assumed in the simulation! The -107dBc/Hz noise level is below the desired specification of -100dBc/Hz , however, and so is acceptable.

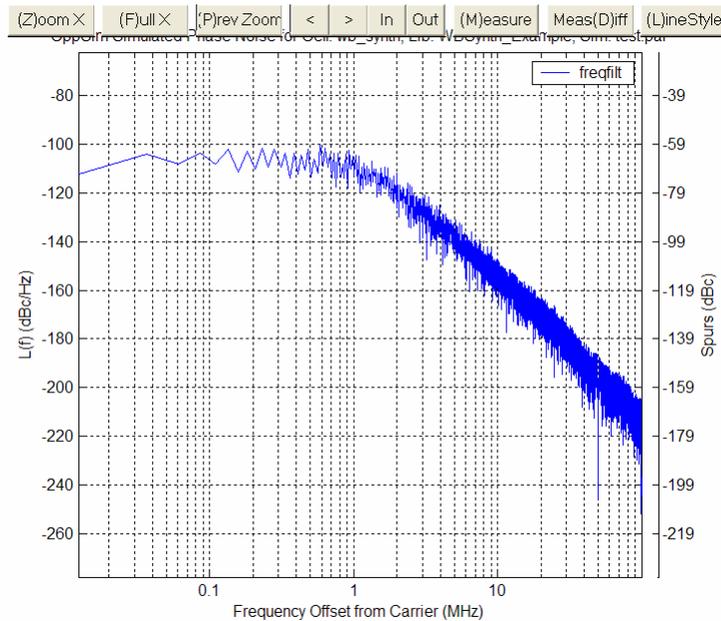
C. Shape Mismatch and the Sample-and-Hold function

The final non-ideality we examine is the shape of the PFD/DAC output and its impact on output noise. As has been discussed, the PFD/DAC output waveform contains information in it about the fractional divide value. If the output is presented directly to the loop filter, then some residual fractional spurs will appear in the output spectrum. To isolate the loop filter from the PFD/DAC output a sample-and-hold is used. By sampling the PFD/DAC output after it completes operation as depicted in the figure below, its shape is ignored. An additional benefit is that spurious energy occurring at the reference frequency associated with the phase detection process is also eliminated.



In the figure, the charge produced by the PFD/DAC is represented by a capacitor voltage, V_{ca} . Note that the slope of the V_{ca} voltage changes from period to period within the time window defined by the charge-box. This is because the PFD/DAC varies the current in that window. However, the *net* voltage excursion once the PFD/DAC completes operation is zero, because no net charge is delivered. If the integrated charge is sampled after the positive current pulse completes, the result is V_{cb} , a voltage that contains no spurious energy, but rather only changes due to any random variation of the PFD/DAC output currents due to thermal noise.

- In the simulation control file, change the **en_s_and_h** parameter from -1.0 (disabled) to 1.0 (enabled).
- Save the changes to the simulation file.
- Run the simulation by clicking the **Compile/Run** button in the CppSim Run Menu.
- Plot the phase noise profile. You should obtain the plot shown below.

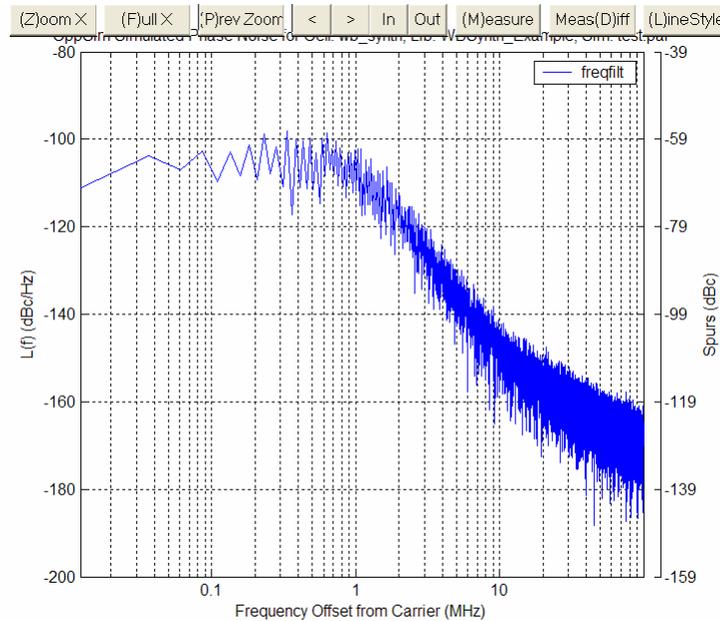


We see that the sample-and-hold has removed the residual fractional spurs as well as the spur at the reference frequency! Of course, a practical sample-and-hold implementation will have some error associated with it, but the behavioral simulation indicates that this is a worthwhile technique to pursue.

To examine the final synthesizer performance with all noise sources accounted for, we edit the schematic once more and re-run the simulation.

- Reconnect the charge-pump noise source by disconnecting **xi14** from the adder, **xi12**, and reconnecting **xi3** to the adder. This adds the charge-pump noise back into the simulation.

- Change the VCO noise at 20MHz from -1550dBc/Hz back to -155dBc/Hz. This adds the VCO noise back into the simulation.
- Save the changes to the simulation file.
- Run the simulation by clicking the **Compile/Run** button in the CppSim Run Menu.
- Plot the phase noise profile. You should obtain the plot shown below.



The overall phase noise performance of the synthesizer as shown in the phase noise plot is very good. It meets the equivalent of the GSM spectral mask at 3.6GHz across all frequencies, and meets the stated spurious goal.

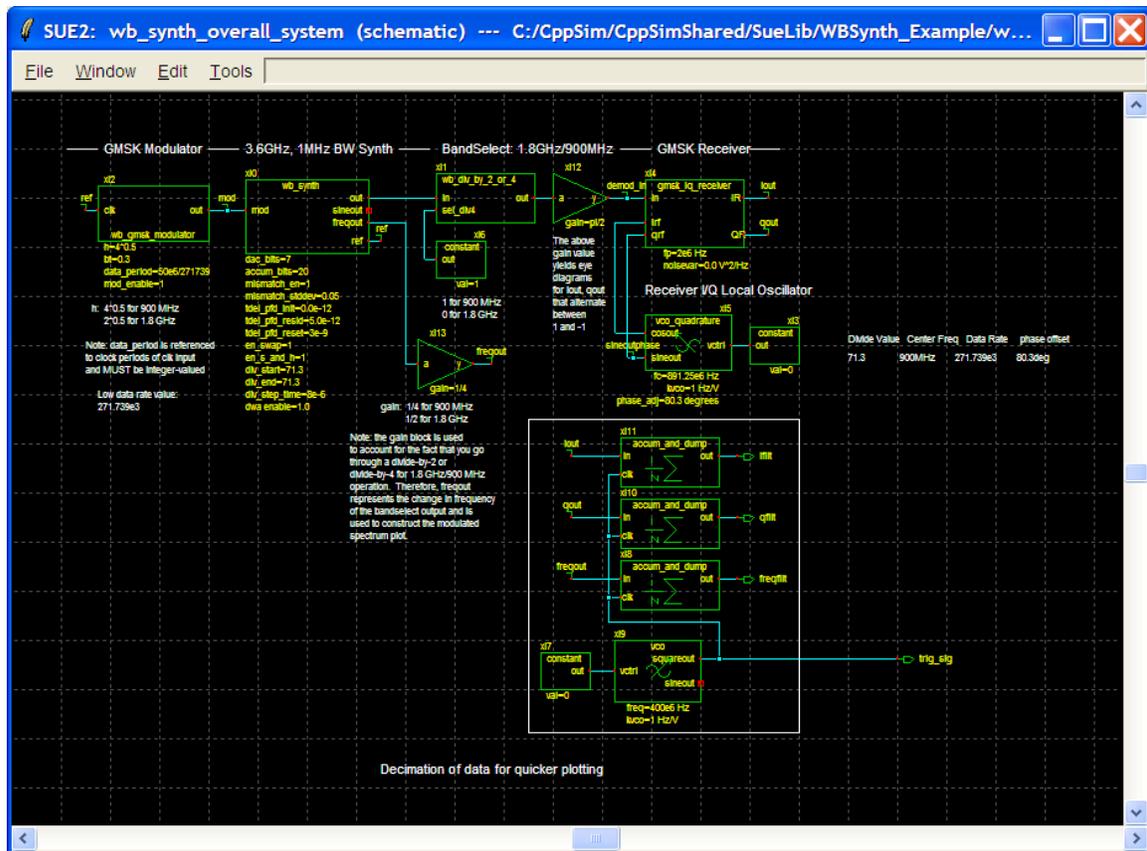
Directly Modulating the PFD/DAC Synthesizer

In this section, we look at utilizing the PFD/DAC synthesizer as a direct modulated GMSK transmitter. For an explanation of GMSK modulation, see the tutorial “Behavioral Simulation of a Basic GMSK Transceiver using the CppSim Program” available at

<http://www-mtl.mit.edu/researchgroups/perrottgroup/tools.html>

The schematic for the PFD/DAC synthesizer based transmitter is named **wb_synth_overall_system**.

- Open the schematic by clicking on the **wb_synth_overall_system** schematic name in the schematic listbox. The schematic is as shown below.



The schematic contains the **wb_synth** block as well as several other blocks used by the transmitter.

- **wb_synth_modulator**: generates a random data sequence that is properly GMSK filtered. It takes as input several values:
 - **h** = modulation depth. For GMSK $h = 0.5$. The VCO output is set to be 3.6GHz for reasons of maximizing quantization noise suppression. However, a GMSK transmission system such as a GSM transmitter requires outputs at 900MHz. There is also a GSM standard that operates at 1.8GHz, both of which can be derived from the 3.6GHz VCO output. Therefore, we ultimately take the transmitter output from the band select divider, **wb_div_by_2_or_4**. To account for the division, we multiply h by 2 or 4, depending on whether we desire a 900MHz band or 1.8GHz band output.
 - **bt** = modulation bandwidth. This is set to 0.3 for GMSK
 - **data_period** = number of reference clock cycles per data bit. We approximate the GSM data rate, which uses a 270kb/s data rate, with 271.739b/s data, since it results in a whole number for the data period, simplifying filter design.
 - **mod_enable**: Enables modulation if = 1.
- **wb_div_by_2_or_4**: divides the synthesizer output by 2 or 4 depending on the desired output band. Set **sel_div4** to 1 for divide-by-4, or 0 for divide-by-2.

- **gmsk_iq_receiver**: coherent gmsk demodulator.
- **vco_quadrature**: the quadrature VCO used by the coherent demodulator. It's key parameter is:
 - **phase_adj**: This is the initial offset in the quadrature VCO that accounts for the delay through the PFD/DAC synthesizer. It will have to be changed if you experiment with different data rates or output frequencies.

The key signals present in the system are

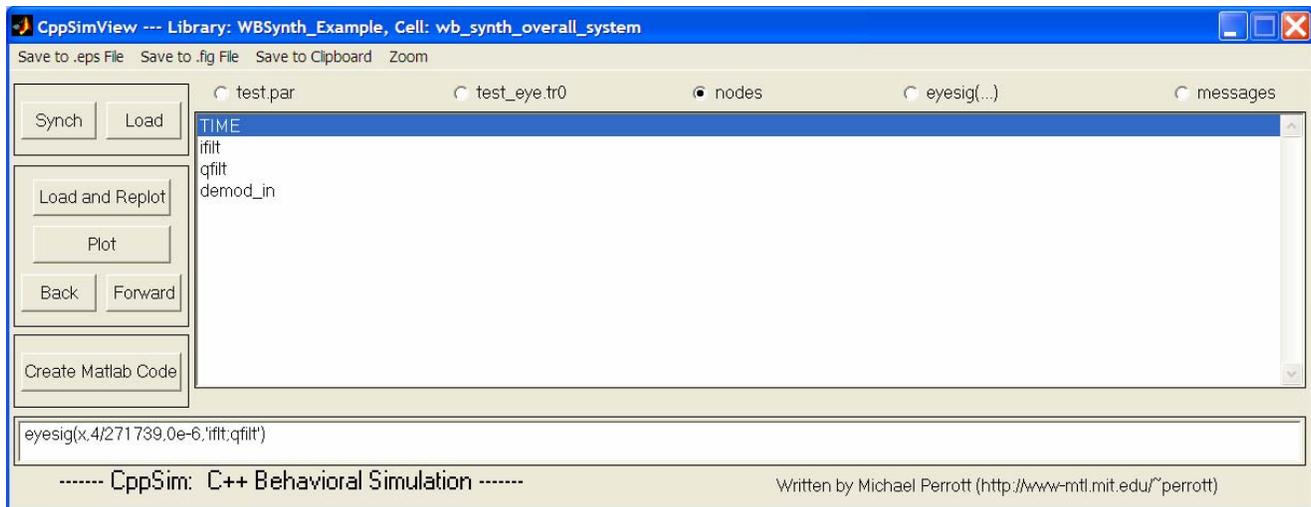
- **ifilt, qfilt**: the filtered, demodulated I and Q data channels.

Having described the system architecture, we run a modulated simulation as follows.

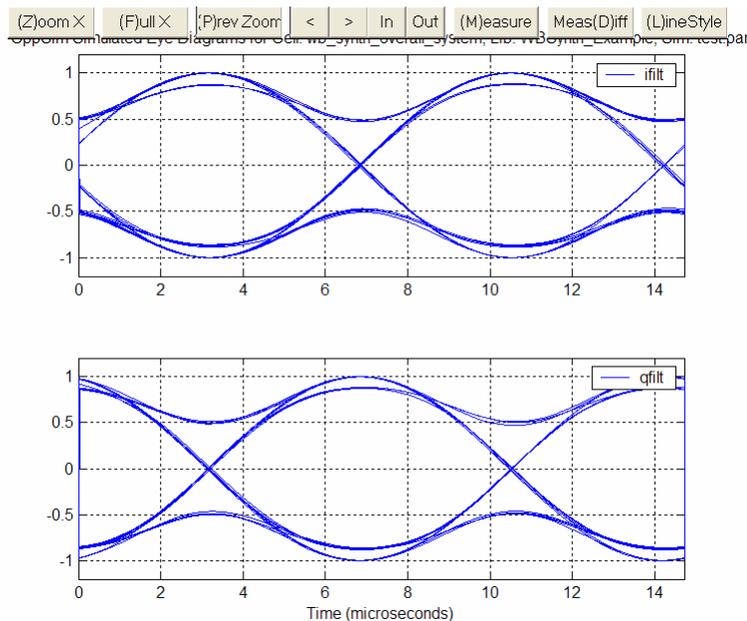
- Synchronize the CppSim Run Menu to the schematic by clicking the **Synchronize** button
- Open the simulation file by clicking on the **Edit Sim File** button. Change the **num_sim_steps** parameter to 20e6
- Change the **end_time** parameter associated with the **test_eye** output from 200e-6 to 400e-6
- Save your changes to the simulation control file.
- Run the simulation by clicking on the **Compile/Run** button

We will now examine the demodulated eyes detected by the coherent receiver.

- Synchronize CppSimView to the schematic by clicking on the **Synch** button.
- Click on the **No Output File** button to see a listing of the output files
- Select the **test_eye.tr0** file and click on the Load button
- Modify the data period in the eyesig plot command so that it reads,
 - **eyesig(x,4/271739,0e-6, 'ifilt;qfilt')**
- Click the Plot button to plot the demodulated eyes. The CppSimView window should look as follows:

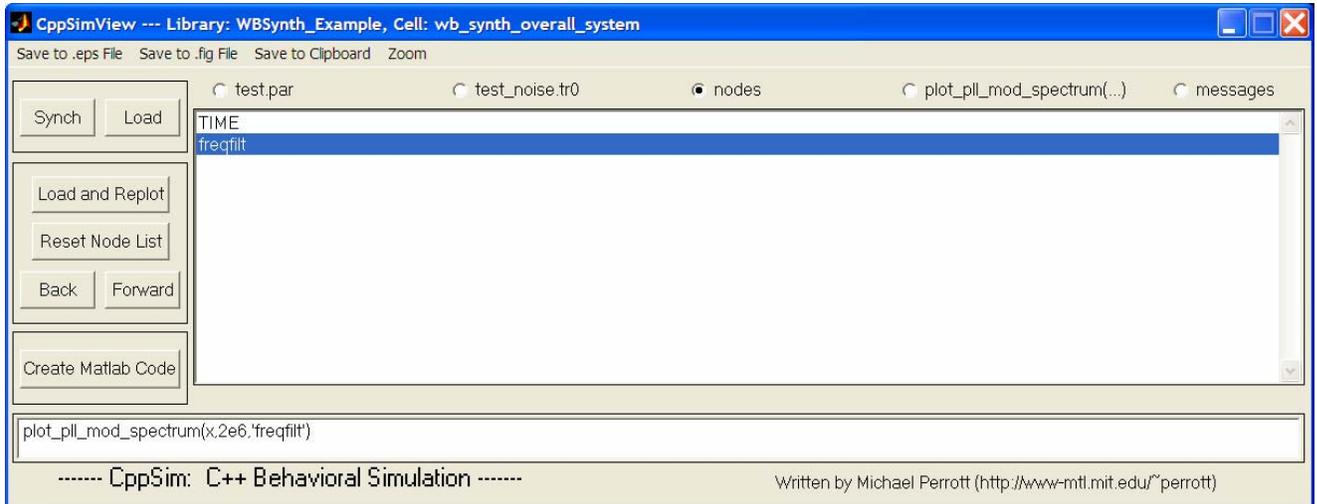


The demodulated eyes should look as follows:

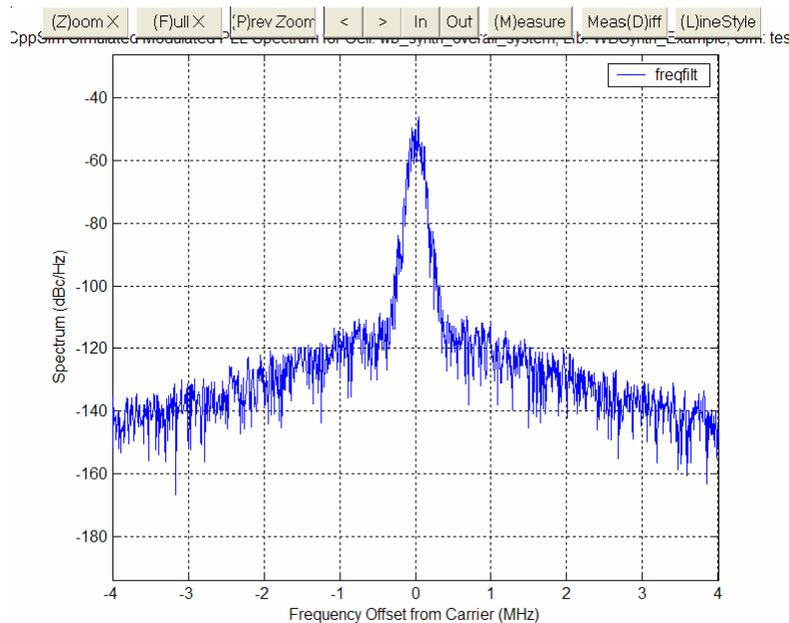


We see that the eyes are wide open. We can also examine the modulated output spectrum.

- Click the **test_eye.tr0** radio button and select **test_noise.tr0** from the file list.
- Click the **No Nodes** button to load in the signals.
- Click on the **plotsig(...)** button, and select the modulated spectrum plotting function **plot_pll_mod_spectrum(...)**. Change the noise plot bandwidth to $2e6$, and specify the node **freqfilt**. The CppSimView window should look as follows:



- Click the **Plot** button. The resulting phase noise profile should appear as follows:



We observe the modulated spectrum follows the expected GMSK profile. To obtain a cleaner spectrum, you can run a longer simulation and repeat the plotting process. 100e6 points is a good length to try.

Conclusion

In this document, we explored the mismatch compensated PFD/DAC synthesizer, an architecture capable of simultaneously achieving high bandwidth and low phase noise. By examining the impact of mismatch sources, we are able to validate the compensation schemes employed to reduce the impact of mismatch noise on overall synthesizer performance.

We have also demonstrated the use of the PFD/DAC synthesizer as a direct modulated GMSK transmitter. The user is encouraged to explore changing the data rate and output band select divider value to explore transmitter performance for various data rates. Additionally, by changing the relevant

control parameters in the **wb_synth** simulation, the user can explore different values of mismatch to observe the impact on synthesizer performance.