Hideo Okawara's
Mixed Signal Lecture Series

DSP-Based Testing – Fundamentals 30
Jitter Injection

Verigy Japan
October 2010

Preface to the Series

ADC and DAC are the most typical mixed signal devices. In mixed signal testing, analog stimulus signal is generated by an arbitrary waveform generator (AWG) which employs a D/A converter inside, and analog signal is measured by a digitizer or a sampler which employs an A/D converter inside. The stimulus signal is created with mathematical method, and the measured signal is processed with mathematical method, extracting various parameters. It is based on digital signal processing (DSP) so that our test methodologies are often called DSP-based testing.

Test/application engineers in the mixed signal field should have thorough knowledge about DSP-based testing. FFT (Fast Fourier Transform) is the most powerful tool here. This corner will deliver a series of fundamental knowledge of DSP-based testing, especially FFT and its related topics. It will help test/application engineers comprehend what the DSP-based testing is and assorted techniques.

Editor's Note

For other articles in this series, please visit the Verigy web site at www.verigy.com/go/gosemi.

Jitter Injection

Signals in mission mode are exposed in noisy environment so that they are influenced by various noises. Phase noise or jitter is one of such noise components. Communication devices are especially designed to tolerate from severe jitter. In order to test jitter tolerance, you may want to modify ideally modulated signal or beautiful bit stream smeared by specific jitter signals. The theme of this month's article is how to inject jitter into a given ideal waveform.

Theory of Jitter Injection

An ideal test signal is described as a simple sinusoidal wave $g(t)$ as Equation (1). A jitter in the signal can be emulated as a phase noise $\phi(t)$ described as Equation (2).

$$g(t) = A\cos(\omega t) \tag{1}$$

$$\begin{aligned} u(t) &= A\cos(\omega t + \phi(t)) \\ &= A\cos(\omega t)\cdot\cos\phi(t) - A\sin(\omega t)\cdot\sin\phi(t) \\ &\left[A\sin(\omega t) = h(t)\right] \\ &= g(t)\cdot\cos\phi(t) - h(t)\cdot\sin\phi(t) \end{aligned} \tag{2}$$

Examining the expanded part of Equation (2), the jitter injected wave $u(t)$ is constructed by the summation of two components. One is multiplication of the original $g(t)$ and $\cos\phi(t)$, and the other is multiplication of the modified original $h(t)$ and $\sin\phi(t)$. Considering Equation (3), the modified signal $h(t)$ can be generated by rotating the original wave by $-\pi/2$ [rad] (-90 [deg]).

$$\cos\left(\alpha - \frac{\pi}{2}\right) = \sin(\alpha) \tag{3}$$

Consequently Equation (2) can be illustrated as follows;

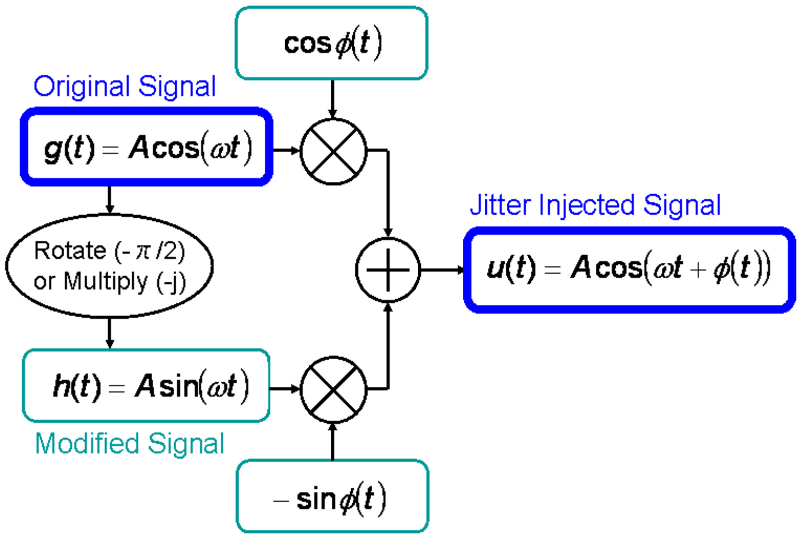

Figure 1 **Basic Theory of Jitter Injection**

The original signal $g(t)$ is converted into $h(t)$ by rotating -90 degrees, which can be performed in the frequency domain. The signal wave is converted into complex number frequency spectrum. The processing of -90 deg rotation means that (-j) is multiplied to each one of the spectral lines. According to Equation (4), it can be realized by exchanging the real and imaginary components, and negating the sign of imaginary part.

$$(a + j \cdot b)\cdot(-j) = (-j)a + (-j)j \cdot b = b - j \cdot a \qquad (4)$$

Test signals in actual device testing can be a complex waveform such as a clock or PRBS bit stream. Then the original signal is not a simple sinusoid as Figure 1 illustrates, but it can be decomposed into $\Sigma A_n \cdot \cos(n\omega t)$ by applying FFT.

Let's simulate the phase jitter component $\phi(t)$ as follows.

$$\phi(t) = A_{rad} \cdot \sin(\omega_a t) \qquad (5)$$

This is a normal sinusoidal jitter whose amplitude A_{rad} and the jitter frequency is ω_a. If you want to inject more complex jitter such as multi-tone, you can program $\phi(t)$ as you like. According to Figure 1, $\phi(t)$ is finally wrapped up in cosine and sine functions.

As a summary, for the whole picture of jitter injection processing, the original signal should be expanded to the summation of multiple components so that the basic diagram in Figure 1 should be expanded as follows;

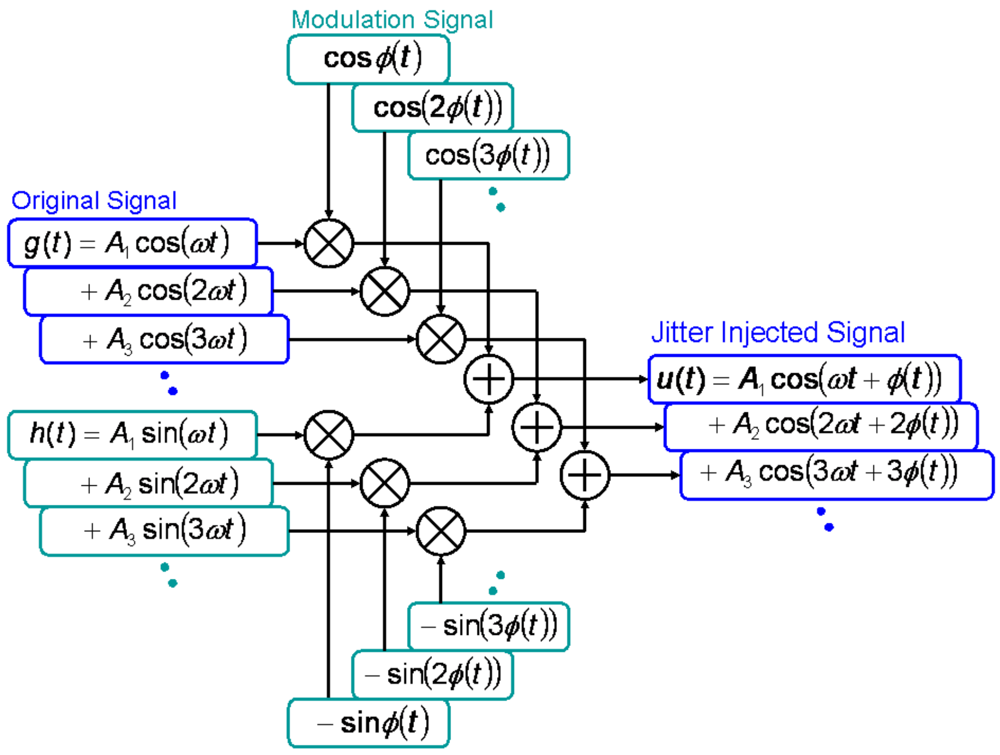

Figure 2 **Actual Jitter Injection Processing**

The original signal is converted into the multiple components by FFT so that the cosine/sine multiplications and the summation are also multiplicate. The important point in this processing is the phase noise component $\phi(t)$ is proportionally applied to the original signal frequencies. When the original frequency is $n\omega$, the phase noise also follows as $n\phi(t)$. This point is already discussed in the previous article.[1]

Program Code Example of Jitter Injection

The code in List 1.1 and 1.2 realizes the data processing illustrated in the diagram of Figure 2. The original wave is assumed as 1.2Gbps 512-bit digital stored in the 8192-point array “dWave[],” which is supposed to be downloaded to the 19.2Gsps AWG. The time domain waveform generated in the subroutine at Line 26 is converted into the frequency domain spectrum at Line 29. The subroutine “AWG_Waveform()” itself is not described in this article. The 100ps-pp sinusoidal jitter is generated as instantaneous phase data at Lines 33 to 40. The jitter frequency is 1 cycle sinusoid during the total AWG period so that it becomes 19.2GHz/8192=2.34375MHz.

```
10:   INT              i,j,Nawg,Nsp,Nbits,Mcycles,Nref;
11:   DOUBLE           dFawg,dFresln,dFbps,dJitterPP,dAppRAD,dA,dP,dScale;
12:   ARRAY_D          dWave,dWaveJ,dPhase,dWave00,dWave90,dCOS,dSIN;
13:   ARRAY_D          dWaveCos,dWaveSin,dWaveSum,dPhase1;
14:   ARRAY_COMPLEX    CWave,CSp,CSp00,CSp90,CWave00,CWave90;
15:
16:    dFawg=19.2 GHz;                            // 19.2Gsps;    AWG Sampling Rate
17:   Nawg=8192;                                  // 8192 Points; AWG Data Size
18:   Nsp=Nawg/2;                                 // # of spectrum lines
19:   dFresln=dFawg/Nawg;                         // AWG Data Frequency Resolution
20:
21:   dFbps=1.2 GHz;                              // 1.2Gbps;  Signal Bit Speed
22:   Nbits=512;                                  // 512 bits; Signal # of Bits
23:   Nref=(INT)(dFbps/dFresln+0.5);              // Bin # of Bit Speed
24:
25:   dWave.resize(Nawg);                         // Signal Data Container
26:   AWG_Waveform(dWave,dFawg,dFbps,Nbits);      // Clock Wave or PRBS Wave
27:
28:   DSP_CONV_D_C(dWave,CWave,1.0,0.0);          // {Real} --> {COMPLEX}
29:   DSP_FFT(CWave,CSp,RECT);
30:
31:   ////////////////////////////////////////////////////////////////
32:
33:   dJitterPP=100.0 ps;                         // 100.0 ps-pp Jitter Amplitude
34:   dAppRAD=2.0*M_PI*(dJitterPP/(1.0/dFbps));   // [sec]->[rad]
35:   dA=dAppRAD/2.0;                             // Amplitude (0-peak) for sinusoid
36:
37:   Mcycles=1;                                  // Jitter 1 cycles in the UWP
38:   dP=2.0*M_PI*Mcycles/Nawg;                   // Jitter Incremental phase
39:   dPhase.resize(Nawg);                        // Jitter Sinusoid container
40:   for  (i=0;i<Nawg;i++) dPhase[i]=dA*sin(dP*i);// Jitter Signal
41:
```

List 1.1 Example Coding for Jitter Injection (Part 1)

In List 1.2, the component-wise calculation is performed from Lines 56 to 83. Each time “CSp00[j]” picks up the j-th single spectrum component of the input signal spectrum so that “dWave00[]” reconstructs the j-cycle single tone wave. “CSp90[j]” is the -90 degrees rotated spectrum to “CS00[j].” So “dWave90[]” reconstructs the -90 degrees rotated wave. The instantaneous phase

[1] Mixed Signal Lecture Series: DSP-Based Testing Fundamentals 29 “Precision Waveform Shift”

“dPhase[]” is proportionally scaled to the bin number at Lines 71 and 72 every time. The scaling is based on the point discussed in the previous article. Then cosine and sine of the phase are generated at Lines 74 to 77. The cosine multiplication and the sine multiplication are performed at Lines 78 and 79, and then they are summed up at Line 80. Finally the waveform component is accumulated at Line 82. The subroutines “Conjugate()”, “C_j()” and “CZero()” are not described in this article but they are very simple operations. “Conjugate()” multiplies -1 to the imaginary part of the input complex number. “C_j()” just performs Equation (4). “CZero()” is the complex number of zero.

```
42:   ///////////////////////////////////////////////////////////////////////////
43:
44:   CSp00.resize(Nawg); CSp00.init(CZero());    // 1 Component Spectrum of g(t)
45:   CSp90.resize(Nawg); CSp90.init(CZero());    // 1 Component Spectrum of h(t)
46:
47:   dWave00.resize(Nawg);                       // 1 Component Wave of g(t)
48:   dWave90.resize(Nawg);                       // 1 Component Wave of h(t)
49:
50:   dCOS.resize(Nawg);                          // cos(phi(t))
51:   dSIN.resize(Nawg);                          // sin(phi(t))
52:
53:   dWaveJ.resize(Nawg);                        // Jitter Injected Wave (Output)
54:   dWaveJ.init(CSp[0].real());                 // Initialize by DC level
55:
56:   for (j=1;j<Nsp;j++) {                       // Component-wise Processing
57:      CSp00[j]     =CSp[j];                    // 1 spectrum component extracted
58:      CSp00[Nawg-j]=Conjugate(CSp00[j]);       // Complex Conjugate Function
59:      DSP_IFFT(CSp00,CWave00);
60:      dWave00=CWave00.getReal();               // 1 waveform component reconst.
61:      CSp00[j]     =CZero();                   // Reset for Next Calculation
62:      CSp00[Nawg-j]=CZero();
63:
64:      CSp90[j]     =C_j(CSp[i]);               // -90 deg Rotation Function
65:      CSp90[Nawg-j]=Conjugate(CSp90[j]);       // Complex Conjugate Function
66:      DSP_IFFT(CSp90,CWave90);
67:      dWave90=CWave90.getReal();               // -90 deg Rotated Waveform Reconst
68:      CSp90[j]     =CZero();                   // Reset for Next Calculation
69:      CSp90[Nawg-j]=CZero();
70:
71:      dScale=(DOUBLE)j/(DOUBLE)Nref;           // n*phi(t)
72:      DSP_MUL_SCL(dScale,dPhase,dPhase1);      // Scaled Phase (Jitter) Amplitude
73:
74:      for (i=0;i<Nawg;i++) {
75:         dCOS[i]= cos(dPhase1[i]);
76:         dSIN[i]=-sin(dPhase1[i]);
77:      }
78:      DSP_MUL_VEC(dWave00,dCOS,dWaveCos);      // Multiplication
79:      DSP_MUL_VEC(dWave90,dSIN,dWaveSin);      // Multiplication
80:      DSP_ADD_VEC(dWaveCos,dWaveSin,dWaveSum); // Summation
81:
82:      DSP_ADD_VEC(dWaveSum,dWaveJ,dWaveJ);     // Component Accumulation
83:   }                                           // Jitter Injection Complete
84:
```

List 1.2 Example Coding for Jitter Injection (Part 2)

Simulation Results

Following figures show the jitter injected results by the codes listed in 1.1 and 1.2. Figure 3(a) illustrates the input clock waveform corresponding to $g(t)$ in Figure 1 and "dWave[]" in List 1. This is supposed to be 1.2GHz/2=600MHz clock waveform. Figure 3(b) illustrates the 100ps.pp jitter injected output waveform corresponding to "dWaveJ[]" in List 1.2. The jitter frequency is 2.34375MHz.

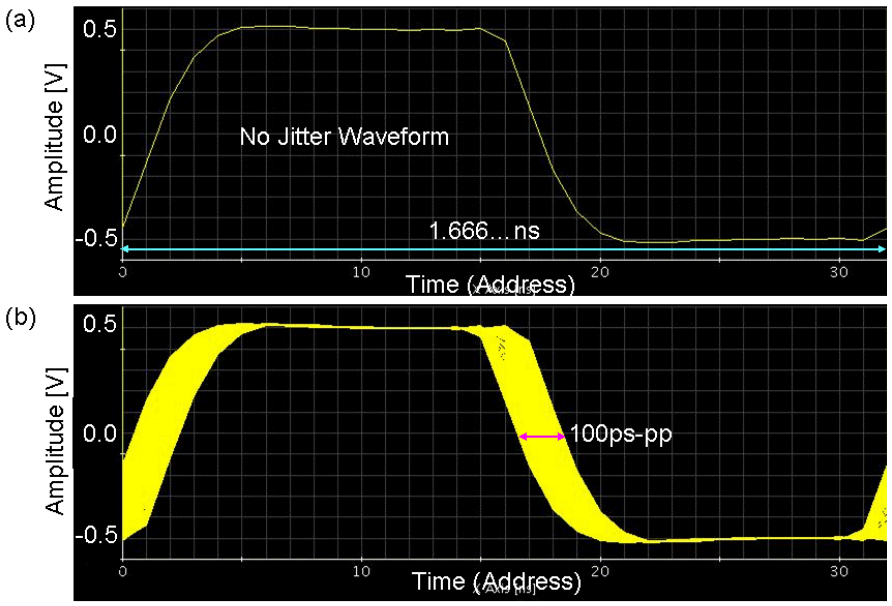

Figure 3 Jitter Injected Clock

Figure 4 illustrates a waveform of 512-bit PRBS bit stream. This corresponds $g(t)$ in Figure 1 and "dWave[]" in List 1. This is supposed to be 1.2Gbps NRZ waveform. The UTP is approximately 426.7ns.

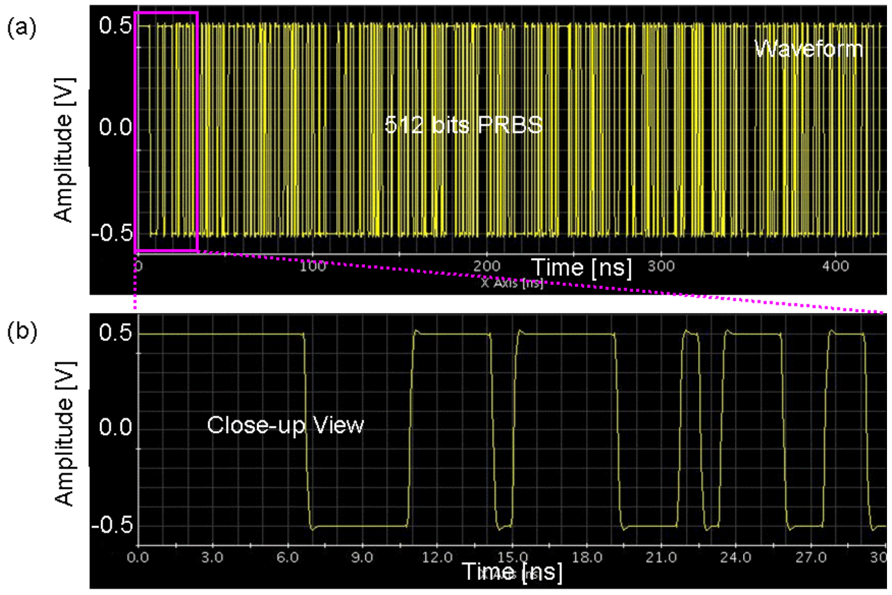

Figure 4 Original Waveform: 512 bits PRBS Data Stream

Processing the PRBS waveform by applying the program in List 1.1 and 1.2, the jitter is injected as Figure 5. It shows the eye patterns. (a) is the original eye pattern with no jitter, and (b) is the jitter injected eye pattern.

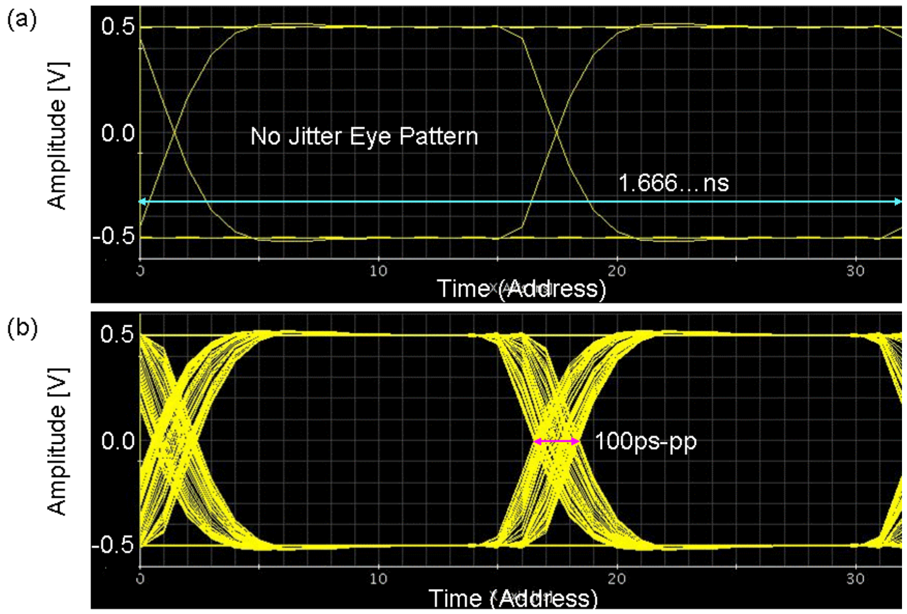

Figure 5 Eye Diagram: No Jitter vs. Jitter Injected

Point of Jitter Injection

Actually the signal processing diagram illustrated in Figure 2 is already discussed in the previous article of PM/FM generation.[2] In short, the jitter injection operation can be realized by the PM operation. The point is the jitter component $\phi(t)$ is proportionally applied to the original signal frequencies. The practical tips are -90 degree rotation operation in the frequency domain and the complex conjugate operation prerequisite for successful IFFT.

[2] Mixed Signal Lecture Series: DSP-Based Testing Fundamentals 28 “PM & FM Waveform Generation (Part 2)”