HABIST™ Case Study

Tom Almy
Opmaxx, Inc.
August 4, 1999

Abstract

Histogram Analog Built-In Self Test (or HABIST™) is an efficient technique to compare analog signals against expected waveforms. Java-based software allows designing and implementing tests which can be used in the testing environment.

Introduction

At ITC '97 we discussed a new approach to Analog BIST in which a histogram was made of a signal in the circuit under test, and the histogram was then analyzed by comparing with the expected or ideal signal histogram. Since then we've realized that the analysis should be tailored to the circuit under test, and we have devised an interactive approach for developing and verifying the HABIST test analysis. HABIST has received a patent and is being incorporated into Opmaxx BistMAXX™ suite of BIST technologies.

HABIST Fundamentals

Figure 1 illustrates the typical HABIST configuration. A test point of the circuit under test is connected to a sample and hold and analog to digital converter. The sample rate is controlled by its own clock which runs asynchronously with the circuit under test. A histogram is made of the samples. A program running on the tester or an embedded processor in device being tested takes the difference between the histogram and an expected histogram obtained from simulation or a "golden" device. A signature is generated, comprising of at the minimum the variance between the histograms, but typically includes relative amplitude, offset, noise level, clipping, and other values depending on the type of waveform. HABIST does not capture frequency domain statistics of sinusoidal waveforms. Because of sampling, HABIST circuits after the sample and hold can run at a much lower clock rate than the circuit under test making HABIST ideal for testing high speed circuits.

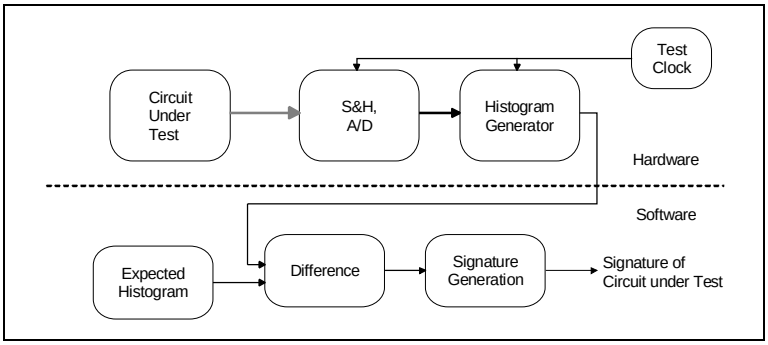

Figure 1— HABIST Testing Data Flow

The signature gives a figure of merit of the circuit, and can provide information useful for diagnosis and repair. However the size of the signature is much smaller than that of the histogram, which in turn is much smaller than that of the digitized sampled signal, which reduces the needed probing bandwidth, saving time and money.

The Testing Problem

To demonstrate the application of HABIST technology, demonstration hardware was built which generated sine waves of different amplitudes, DC offsets, peak clipping, and cross-over distortion, various combinations of which could be selected as the output of the "circuit under test." Testing such a circuit is different than the original HABIST scheme given in the original paper; in this test problem, no known good waveform was available. How could HABIST be used? The circuit waveform would be used to synthesize an "ideal" non-distorted version of itself. By comparing the relative signal amplitude of the observed waveform's histogram with that of a histogram of a perfect sine wave of a known amplitude, the amplitude of the observed signal, if it were an undistorted sine wave, could be extracted. This undistorted sine wave would then be used as the reference to calculate clipping and crossover distortion of the observed signal.

Let's jump ahead and look at the design solution.

The Solution

Figure 2 is the block diagram of the software solution. Histogram generation is performed in software, however ideally it would be performed in the device to reduce the data bandwidth. For that reason the signal cannot be analyzed except after it is converted to a histogram. The centering blocks shift the histogram so that either the mean or median data point is at the center histogram bin. Note that the P-P Amplitude block measures the signal amplitude as observed in the histogram, the spread of the histogram data, not the amplitude of the histogram itself. In fact, the histograms are manipulated in normalized form, disregarding the actual number of data points captured.

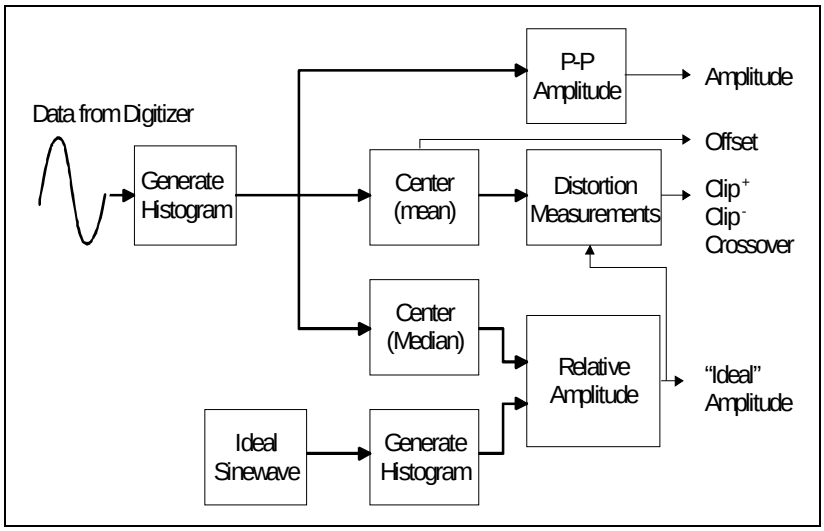

Figure 2— HABIST Software Solution

Relative amplitude is calculating by minimizing the variance between two histograms by scaling the spread of one of the histograms. Because the two histograms could also vary in their DC signal offset, effecting the variance, and the histogram spreading operation must be centered on the median data point, both histograms must first be adjusted so that their median data point is center bin or between the two central bins if there are an even number of bins. The ideal sine wave can be created such that its histogram is centered, so a separate centering step is not needed.

The clipping and crossover distortion measurements are made by observing the deviation in the distribution compared to an ideal sine wave of a specified amplitude. The amplitude is that obtained from the preceding relative amplitude algorithm. The DC offset of the test signal is obtained from calculating the mean bin of the signal histogram. This offset is also used to center the histogram for the distortion measurements, which require that the mean bin be as close to the center as possible.

While not part of this example, algorithms have also been developed for measuring noise level, amplitude modulation depth, sawtooth wave linearity, and square wave rise/fall times, overshoot, and low frequency roll-off.

Deriving and Verifying the Solution

A few sample waveforms were captured, to be used in the development of the HABIST test prior to the construction of the test fixture. To aid in development of HABIST tests, algorithms for HABIST analysis as well as waveform synthesis were encapsulated in the Java™ language as JavaBeans™. A bean is a software component which has a well defined interface allowing it to be utilized in graphical programming environments. Figure 3 shows VisualAge® for Java programming environment being used to generate a program to test the HABIST algorithm described above on sample waveforms. The beans have been found to work in all other bean-aware Java programming environments tested, including products from Borland, Sun, and Net Beans. The following beans are some of those implemented for HABIST development:

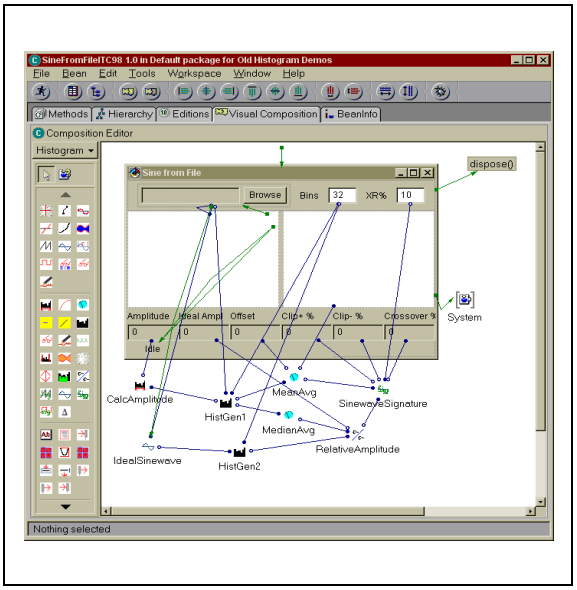

Figure 3— HABIST Algorithm Verification

Histogram Processing Beans

ADCLinearity	DNL/INL measurements of an ADC
Amplitude	Amplitude of signal
Average	Mean/median bin, optional shift
Difference	Generate difference histogram
HistogramGenerator	Generate histogram from data array
HistogramReader	Read histogram from file
Iterator	Iterate on data and collect result
ModulationDepth	Calculate modulation depth
NoiseLevel	Measure relative noise level and compensate
Positioner	Position a histogram
RelativeAmplitude	Calculate amplitude based on ideal histogram
SawtoothSignature	Sawtooth histogram measurements
SinewaveSignature	Sine wave histogram measurements
SquarewaveSignature	Square wave histogram measurements
Variance	Calculate variance between two histograms

Waveform Synthesis Beans

AddNoise	Add noise to waveform

Adjustment	Gain and offset adjustment
Clipping	Clip waveform
CrossoverDistort	Add cross-over distortion
Modulator	Amplitude modulate waveforms
SawtoothGenerator	Basic sawtooth wave generator
SimulatorWaveformReader	Read a waveform from simulator output
SinewaveGenerator	Basic sine wave generator
SquarewaveGenerator	Square wave generator

While the histogram processing beans represent algorithms that are used for testing, the waveform synthesis beans are helpful in developing tests. Other beans, not listed, include “visible” beans that provide a user interface for the created Java program. Creating additional beans for new histogram processing algorithms is a straightforward process, taking only minutes after the algorithm has been coded and tested.

A Java program was created to test the HABIST solution. Figure 3 shows the graphical user interface components at the top, and icons for the various histogram beans at the bottom. Figure 4 shows a close-up view of the connections to the histogram beans. The programming environment allows creating and placing these beans and then making connections between them. The histogram beans support a data flow architecture; when the output of one bean is connected to the input of another, any change in the first beans output signals the second bean to process the new data presented by the first. Sequential operation is implied by the order of the connections. The inputs of some of the histogram beans come from fields in the user interface to provide adjustable parameters and selection of the test waveform, while the outputs of some beans connect to numeric or graphical display fields in the user interface.

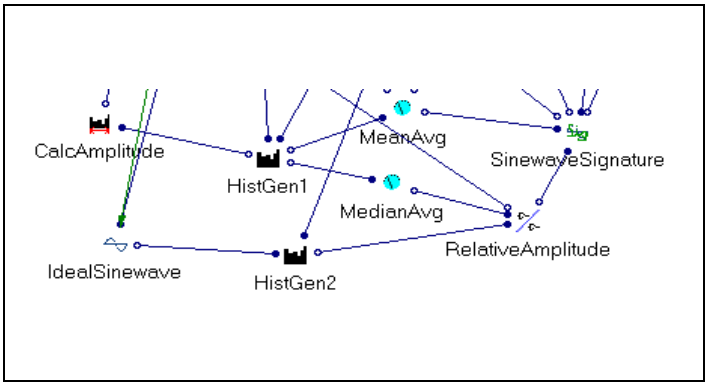

Figure 4— Connecting Histogram “Beans”

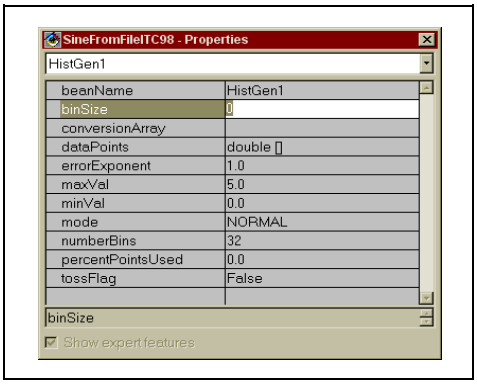

Figure 5—Bean Properties

Constant parameters of the histogram processing algorithms can be specified by setting the bean properties (see figure 5). The HistogramGenerator bean allows setting the input levels for the minimum and maximum bins, setting the number of bins, the maximum bin count (at which point additional data is ignored), and whether to ignore data out of range or add to the outermost bins. Thus it can simulate hardware histogram generators. It has the ability to introduce errors typifying faults of analog to digital converters. The *dataPoints* property here is not a constant as we are using it, but is an input array which is read in from a file. This connection is made graphically and appears

in figure 3 as a line from the user interface control in the upper left corner of the display to the HistGen1 bean.

All of the histogram beans have properties to customize the algorithms for specific requirements. For example, the Amplitude Bean, labeled CalcAmplitude, has an input histogram property that is connected to the output histogram property of HistGen1. When a new histogram is generated, the Amplitude Bean will automatically recalculate the histogram amplitude. It can do this in two different ways, depending on the test designers preference. The bean properties are shown in figure 6. The *mode* property is a boolean value. If it is true, the signal amplitude is calculated as the distance between the two bins with the greatest content. If it is false, then a peak to peak amplitude is calculated, after deleting high and low points to ignore noise. The percentage of points to ignore are also expressed as the bean's properties.

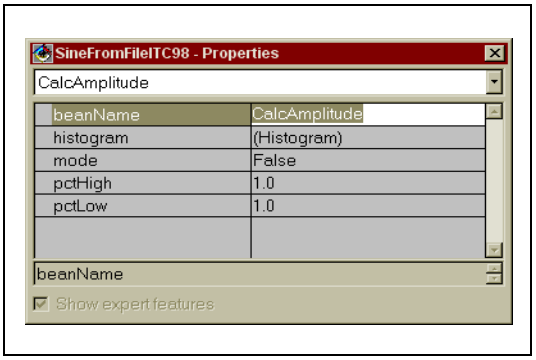

Figure 6— Amplitude Bean

The finished algorithm test program was executed. When given a sample waveform with heavy clipping, the results shown in figure 7 were obtained. The number of bins was set to 32, corresponding to a 5 bit Analog To Digital Converter. The *XR%* setting of 25 means that the waveform will be examined for crossover distortion within the central 25% of signal amplitude range. The graphs show the input waveform for reference purposes, with a scale of 0 to 5 volts, and the histogram of the waveform.

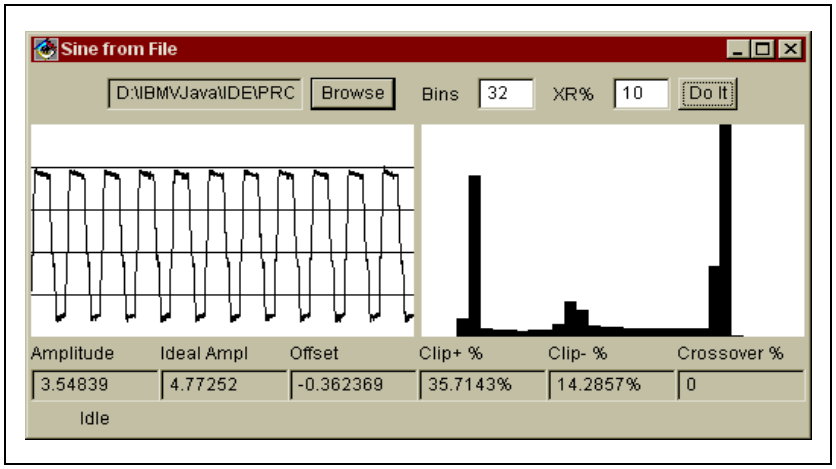

Figure 7— Histogram Analysis Test In Operation

While the amplitude measured roughly 3.5 volts, the analysis concludes that if the signal were not to have been clipped, it would be roughly 4.8 volts, with 36% of the positive and 14% of the negative sides clipped. The mean amplitude is -.36 volts from the midpoint of 2.5 volts, and 3% of the signal amplitude is lost via crossover distortion. This closely matches measurements made of the original signal.

Figure 8 shows the results with a different sample waveform, that of the sine wave generator without additional distortion. The number of bins has been increased to that needed for a 6 bit A/D converter. The test results show a much smaller amount of distortion than the preceding example. The suspiciously identical measurements of positive and negative clipping is a result of the resolution of the measurements. The absolute magnitude of clipping can only be measured to the amplitude

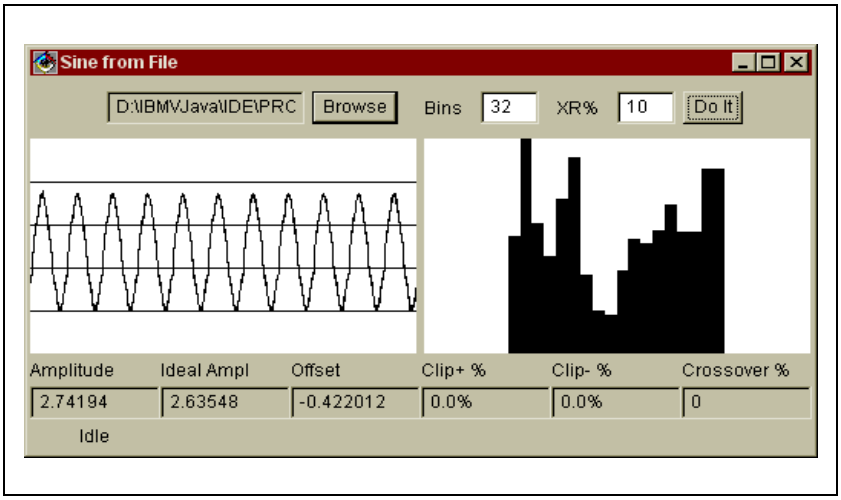

Figure 8— A Good Sine Wave Histogram

difference represented by adjacent bins. Although not shown here, the signature algorithm does report the resolution as the percentage of signal amplitude.

Conversion To Library Routines

The actual circuit testing was to be performed using LabVIEW™ (see figure 9). While the existing histogram analysis algorithm could conceivably be used in test situations, it was much easier to interface LabVIEW to C++ routines.

The modular design of the HABIST Java Beans allows them to be easily used in graphical design

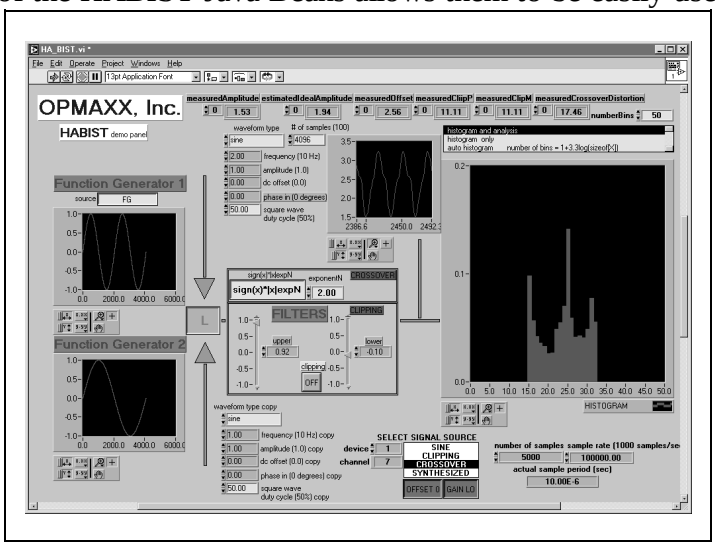

Figure 9— HABIST under LabVIEW

environments, however they also can be used in traditional, non-graphical, procedural programming. A C++ class was written which functionally matches the Histogram Processing Java Beans. The analysis algorithm was tested again, first using Java without the user interface. Then the Java code was converted to C++ and retested. Finally the C++ code was converted to a form accessible from LabVIEW. The commented C++ code representing the connections of the Java Beans was 86 lines long:

```
#include"histo.h"
#include"math.h"
#include"stdio.h"
#ifndef M_PI
#define M_PI        3.14159265358979323846
#endif

const double DELTA=0.01;    // gain calculation delta
const double SPREAD=35.0;   // crossover spread (% of whole)
const double IDEALPEAK=1.0; // Peak amplitude of ideal waveform
const double MAXBIN=5.0;    // Maximum bin value for input data
const double MINBIN=0.0;    // Minimum bin value for input data
const double PCTIGNORE=1.0; // Percentage of points to ignore (amplitude calc)
extern "C" {
void __declspec(dllexport) processHistogram(short *data,
                                            int dataSize,
                                            double *histData,
                                            int numberBins) {
    HISTOGRAM h(data, dataSize, numberBins, 0, 0, 4095);
    h.normal();
    for (int i=0; i < numberBins; i++)
        histData[i] = h[i];
}
void __declspec(dllexport)
        demoHistogram1(double *data,
                       int dataSize,
                       double *histData,
                       int numberBins,
                       double *measuredAmplitude,
                       double *estimatedIdealAmplitude,
                       double *measuredOffset,
                       double *measuredClipP,
                       double *measuredClipM,
                       double *measuredCrossoverDistortion) {
    int i;  // Temp
    HISTOGRAM h(data, dataSize, numberBins, 0, MINBIN, MAXBIN);
    h.normal();
    for (i=0; i < numberBins; i++)
        histData[i] = h[i];
    *measuredAmplitude = h.ppAmplitude(PCTIGNORE,PCTIGNORE);
    double *idealData = new double[dataSize];
    for (i=0; i < dataSize; i++) {
        idealData[i] = sin((i*2.0*M_PI)/dataSize);
    }
    HISTOGRAM ideal(idealData, dataSize, numberBins, 0,-IDEALPEAK,IDEALPEAK);
    ideal.normal();
      // Calculate estimated ideal amplitude
    {
        double adjustment = h.median();
        *measuredOffset = adjustment*(h.getMaxVal()-h.getMinVal())/numberBins;
```

```
        double shiftAmount = numberBins/2.0 - adjustment;
        HISTOGRAM sh1 = h.position(0, shiftAmount, 1.0);
        sh1.normal();
        double estGain = ideal.amplitude()/sh1.amplitude();
        double gain;
        gain = sh1.gain(ideal, estGain, DELTA, 0);

        *estimatedIdealAmplitude = ((IDEALPEAK*2.0)/gain)*
                            (sh1.getMaxVal()-sh1.getMinVal())/
                            (ideal.getMaxVal()-ideal.getMinVal());
    }

    // Generate histogram for signature analysis,
    // then perform analysis
    {
        double dummy;
        double adjustment = h.mean();
        double shiftAmount = floor(0.5 + numberBins/2.0 - adjustment);
        HISTOGRAM sh2 = h.position(0, shiftAmount, 1.0);
        sh2.normal();
        *measuredCrossoverDistortion =
                sh2.xover(*estimatedIdealAmplitude, SPREAD)*100.0;
        *measuredClipP =
                (1.0-sh2.clipp(*estimatedIdealAmplitude, dummy))*100.0;
        *measuredClipM =
                (1.0-sh2.clipm(*estimatedIdealAmplitude, dummy))*100.0;
        }
        delete [] idealData;

    }
}
```

The LabVIEW test program worked as desired, with only one iteration to correct an error in the C++ class. This was a clerical error introduced when converting Java to C++.

The analysis algorithm executed in 4.6 milliseconds in the C++ version, and 5.1 milliseconds under Java. This was using a Pentium™ II 400MHz PC running Windows NT. In this case the time to capture the data (and generate the histogram) is greater than that for processing. Since these can be overlapped, the processing time is effectively zero.

In cases where an embedded processor is available for testing, the HABIST algorithms can be easily adapted to micro-controller assembly language, using fractional integer arithmetic instead of the floating point used in the C++ and Java versions. While execution time would be longer, it would not be longer to the extent suggested by the relative processor power. The C++ design is inefficient as it was designed with ease of use and safe memory management (memory leaks are basically not possible) in mind.

Conclusions

HABIST has been shown to be a low overhead approach to testing analog circuitry. HABIST test development is simplified by having a graphical environment to design tests, and verify their operation. Testing can be performed with various mixes of on-chip and external elements, with analysis performed in C++ or Java.