MIXED SIGNAL FAULT SIMULATOR: COMPARISONS OF HARDWARE AND SIMULATION RESULTS

By

Aaron E. Case

A thesis submitted to the faculty of
The University of North Carolina at Charlotte.
in partial fulfillment of the requirements for the degree
of Masters of Science in the Department of
Electrical and Computer Engineering

Charlotte

2002

Approved by:

_____
Dr. Thomas Weldon

_____
Dr. Charles Stroud

_____
Dr. David Binkley

ABSTRACT

AARON EARNEST CASE. Mixed Signal Fault Simulator: Comparisons of Hardware and Simulation Results. (under the direction of DR. THOMAS PAUL WELDON)

This thesis describes a mixed signal fault simulator and compares simulated results to theoretical and hardware experimental results. The fault simulator takes a SPICE net list file as input and produces, as output, the circuit's behavior under certain fault conditions in order to establish the fault coverage of combinations of test waveforms and output metrics. The present form of the fault simulator includes catastrophic faults, typically open or short circuits. To demonstrate the efficacy of the fault simulator, simulated results were compared against hardware experimental results to verify the accuracy of the simulator for the specific case of a BiQuad filter. In this thesis, the focus is on the development of the analog portion of the mixed signal fault simulator, since the treatment of the digital portions of the circuit is well known and well defined.

ACKNOWLEDGEMENTS

I would first and foremost like to thank Dr. Thomas P. Weldon for affording me the opportunity to work under his direction on the DARPA neo-CAD project as well as all direction and tutelage involved therein. I would also like to thank Dr. Thomas P. Weldon for all the challenging and rewarding course work over the past few years. I would like to thank Dr. Stroud for material support. I would like to thank Dr. Makki for encouraging and convincing me to enter the graduate program at UNC Charlotte. I would like to thank my fellow research assistants Clark Hopper, Steve Tucker, Jason Morton, Ana Maria, Deepa Patel, Ramsey Hourani, and Konrad Miehle for their respective contributions and assistance. Supported by Defense Advanced Research Program Administration (DARPA) and managed by the Sensors Directorate of the Air Force Research Laboratory, USAF, Wright-Patterson AFB, OH 45433-6543.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
BIST	Built in Self Test
BJT	Bipolar Junction Transistor
CAD	Computer Aided Design
CUT	Circuit Under Test
DFT	Design for Test
DUT	Device Under Test
FET	Field Effect Transistor
FPGA	Field Programmable Gate Arrays
HBIST	Hybrid Built in Self Test
IC	Integrated Circuit
MOS	Metal Oxide Semiconductor
ORA	Output Response Analysis
ROC	Receiver Operating Characteristics
VCVS	Voltage Controlled Voltage Source
VLSI	Very Large Scale Integrated
SFA	Statistical Fault Analyzer
SPICE	Simulation Program with Integrated Circuit Emphasis
SoC	System on a Chip
TPG	Test Pattern Generator
pdf	Probability Density Function

CHAPTER 1: INTRODUCTION

This thesis presents a mixed signal fault simulator for built-in self-test (BIST) and gives experimental hardware and simulation results that verify the performance of the simulator. The focus of this effort is on the analog portion of the mixed signal fault simulator since the methodology for digital portions of a mixed signal system are well known. The thesis begins with an introduction of the relevance of fault simulation and how it fits into the overall product cycle. Next, the architecture and methods used to develop the fault simulator are discussed. Then, simulator results are compared with hardware results to validate the fault simulator models and techniques. Finally, directions for future work are offered. In the following, the background of the fault simulator is given. Details on the simulator and experimental results are given in subsequent chapters.

To place the need for a fault simulator in a larger context, consider a typical manufacturing process that has at some point a verification stage, whereby it is determined that a product meets specifications. This stage of production should seek to pass products that are good and eliminate the products that are bad, and only the products that are bad. In the process of product verification there are inevitably four categories of products found [21]. In the first category, bad units that are found to be good in production are known as false positives. This category can be costly to a firm through replacement costs, loss of reputation, and ensuing loss of market share. The false-

positive portion would ideally be zero for any enterprise as it is a liability in the mart of competitive commerce. In the second category, units that are found to be bad, and are bad, are minimally damaging to an enterprise in the sense that there is a loss of materials. In the third category, good units that are found to be bad in production are known as false negatives [21]. False negatives do not greatly damage a firm's reputation, but cause loss of resources and time. In the final category, good units that are found to be good in production are the yield of an enterprise. So, it can be seen that any product verification process should limit any erroneous prognosis as well as bad products. In this manufacturing context, the fault simulator is a tool for designing built in self test (BIST) to minimize losses due to false positives and false negatives.

In addition, the complexity of mixed signal and analog integrated circuits lead to complicated verification requirements [18]. In particular, it is difficult to predict how analog portions of a mixed signal system will behave under variable circumstances such as component variations, faults, and different test metrics. Such predictions of system behavior, either for use with external testing equipment or for use with built-in self-test architectures, can be generated by a fault simulator early in the design and test cycle. Through such fault simulation, fault coverage can be determined for a set of test patterns and output metrics to ensure that a circuit is free of manufacturing defects.

In the following, we first review the current state of the art. Then we give an overview of current challenges in fault simulation. Finally, the problem addressed in this thesis is stated.

1.1 Current State of the Art

In digital portions of a mixed signal circuit, fault modeling is well established. Digital fault modeling can be classified into two categories, gate level fault modeling and transistor level fault modeling [21, 8, 2]. In digital circuits there are two types of faults, stuck-at-one(sa1) and stuck-at-zero(sa0)[2]. Both levels of digital fault modeling utilize these types of faults. In gate level fault modeling, the stuck at faults are exercised at the gate inputs and outputs. In transistor level fault modeling required for CMOS technology, stuck at faults are modeled at the transistor level rather than just the gate level, giving a more complex yet comprehensive picture of fault conditions.

There has been much done in the area of digital fault simulation and modeling, but much less work in the realm of analog fault simulation [8]. The lack of a mature analog fault simulator can be partly attributed to the lack of mature fault models for analog circuits [7]. Furthermore, modeling and isolating faults in analog circuits is one if the most difficult tasks in diagnostic engineering [8].

The work on analog fault models can be classified into two categories, catastrophic faults and parametric faults [26]. Catastrophic faults, also referred to as hard faults, are the analog equivalents of stuck at faults in the digital domain. Analog hard faults occur when the terminal nodes of the component are stuck short or stuck open. Parametric faults are defined as any variation of the component values outside the acceptable performance range or tolerance limits [27].

The current models used for catastrophic faults of analog circuits include the use of high and low resistances to model shorts and opens at the component terminals as later shown in Figs. 2.5 and 2.6[27, 28]. The resistive and capacitive components have a 1 Ω

resistance in parallel ($R_p = 1\ \Omega$) for shorts and a 100 MΩ resistance in series ($R_s = 100$ MΩ) for opens. Similarly, the MOSFET is faulted in much the same manner across the source and drain only. In contrast, the BJT has a stuck-open and stuck-short fault between each of its three terminals, collector, emitter, and base [27].

Many different methods have been proposed for analog fault simulation. One notable method uses DC transfer function testing to test and isolate faults [10]. While DC testing is promising in cost and compact in layout size, the method has lower fault coverage at the macro level and is not viable at the transistor level [10]. DC testing does provide for simpler fault modeling but, does not give adequate parametric fault coverage for many types of analog circuits [10].

A second method, behavioral fault simulation works well for simplifying and expediting analog fault simulation [4]. In behavioral fault modeling methods, the basic algorithm is the same as that for event-driven logic simulation, except that the fault simulation algorithm propagates fault lists along with logic values through gate level hardware descriptions [4]. Behavioral modeling is generally accepted as being faster but less accurate than other methods and does not lend itself well to analog fault modeling without acceptable levels of complication. However, behavioral modeling only works at the gate level and therefore will not lend itself to analog micro-modeling [4].

A third method, functional fault models also work well at simplifying digital fault modeling but are often too complex and don't offer high fault coverage at the transistor level for analog faults [8]. Functional fault models work on blocks of analog circuits and can therefore only diagnose a faulty block of components not individual component

failures. Furthermore, functional fault models only offer a limited number of output conditions for each fault given a predefined test vector.

Another analog fault modeling technique is the test-oscillation methodology. The oscillation methodology treats every analog sub-circuit as an oscillator for verification [9]. The technique tests the frequency response of the fundamental analog blocks against known frequencies for fault-free case to detect faults [9]. While the method is promising with respect to cost and area overhead, in most cases, the method is not generally applicable to all types of circuits without considerable modifications [9, 2, 5].

For any methodology, fault simulation is used to identify the best set of test vectors and output metric to be capable of finding of identifying the maximum number of faults in a circuit. Fault simulation allows investigation of the efficacy of different test vectors and output metrics. By using the fault simulator, the best test vector and output metrics can be identified for a particular system.

Although not recommended, fault simulation could be bypassed in the design of a system. In this case, the verification stages of chip manufacturing consist of blindly applying a signal and measuring a "good output" for a fault-free unit. Without fault simulation data, selecting test vectors for any testing architecture would be not be possible due to a lack of prior knowledge of circuit behavior under faulty conditions. The fault simulator on the other hand allows investigation of input/output behavior of any possible fault conditions for a variety of test vectors and output metrics.

At present, the state of the art for analog and mixed signal fault modeling has yet to witness the same success as digital fault modeling and test pattern generation [8]. The more primitive state of analog and mixed signal fault simulations and techniques can be

attributed to the complexity of fault modeling in an analog circuit versus digital circuits. Research in IC testing has produced various methods and products to approach the problem of analog fault modeling in mixed signal circuits but has yet to arrive at any widely accepted standard or method. The aforementioned technologies offer potential for analog fault modeling to close the gap on much more advanced digital fault modeling [2]. Thus, this thesis focuses on initial steps toward mixed-signal fault simulation.

Analog fault simulation and fault modeling present different, and more complex, challenges than digital fault modeling, due to the nature of analog circuits [26, 27]. The performance of analog portions of a mixed signal circuit are subject to parametric variations, in addition to catastrophic faults such as an open or a short. Any simulation or testing procedure for an analog circuit must include parametric variation of components, as well as catastrophic faults, in verification and test vector generation. Therefore, analog portions of a mixed signal IC present much more complex problems in testing and simulation, and thus are more costly in verification. In the next section, we give further overview of issues in fault simulation.

1.2 Overview of Current Challenges and Solutions to Fault Simulation

Although there has been much work in the area of digital fault simulation and fault modeling, analog fault simulation lags far behind that of digital fault simulation due to the complexity of analog fault simulation and analog IC verification [8]. Analog and mixed signal IC's require more time and money investment in fault modeling and simulation to achieve the same level of fault coverage as their digital counterparts [8]. The nature of an analog circuit makes testing, and testing decisions, considerably more complex. With a digital circuit, a test vector will indicate that a certain gate or transistor

is stuck at 0 or 1, but an analog test vector needs to indicate if a analog circuit is within a predetermined error bound [16]. Thus, one complicating problem in analog testing is where to draw the line, or threshold, for fault-free or faulty analog circuits as well as how to quantify the yield of fault-free circuits. In this, fault coverage can be defined as the number of detected faults divided by the total number of possible faults (i.e., in a fault list) [2].

An analog or mixed signal fault simulation can consider normal parametric variations in conjunction with catastrophic faults in fault simulation. To accurately estimate the probability distribution of how a fault-free circuit behaves, a large number of samples taken from parametrically randomized circuits is required. The range of values for any given output metric that are obtained for the randomized good circuit will indicate natural variations due to the fabrication and manufacturing process of the system. This parametric randomization also must be done for each injected fault to determine accurately the probability distribution of output metric values for each fault. The foregoing randomization for fault-free and faulty circuits, combined with need to evaluate the circuit with a multitude of waveforms, lead to very large simulation times even for the simplest of circuits. Simulation times on the order of weeks can be encountered with as few as 15 components, 30 faults, 200 randomizations, and 30 waveforms. In this example, there are $(30+1)\times200\times30= 186{,}000$ combinations. Therefore, an important issue in fault simulation considered by this thesis is reducing simulation times and developing techniques to accelerate computation, and these issues are partly addressed in the future work section.

1.3 Statement of Problem

This thesis describes the design and testing of the analog portion of a mixed signal fault simulator. The simulator takes as input a SPICE file of a circuit under test (CUT) and produces, as output, statistical information on the behavior of the circuit under all fault conditions with a variety of candidate test pattern waveforms and output metrics. The raw output metric data of the simulator is post-processed into histograms that provide statistical fingerprints of a variety of output metrics for each potential catastrophic fault in the circuit. The output data and histograms can then be used to identify the best input stimuli, or test pattern waveforms, and the best output metrics for testing a hardware version of the circuit.

The present fault simulator is tailored to a specific built-in self-test (BIST) architecture for mixed signal systems. In the BIST architecture under consideration, test pattern waveforms are applied and output metrics are measured [2][13]. Although the current fault simulator is tailored for a specific architecture for BIST, the modular design of the fault simulator permits adaptability to future architectures in future work.

In Chapter 2, we first describe the design of the fault simulator. Then, simulation results for a BiQuad benchmark circuit are given in Chapter 3 and compared with hardware measurements and compared with hand-calculated theoretical results. Finally, Chapter 4 gives suggestions for future directions on the fault simulator.

CHAPTER 2: FAULT SIMULATOR DESCRIPTION

The focus of the fault simulator under consideration is with the analog portion of a mixed signal fault simulator. As discussed in the previous chapter, the more difficult issues are on the analog side, whereas there exist well known methodologies for fault simulation on the digital side. In the following, the analog fault simulator is described. Experimental results using the simulator are presented in the subsequent chapter.

2.1 BIST Framework

The fault simulator takes, as input, a spice net list describing a circuit and simulates randomized versions of the circuit with, and without, faults. The present fault simulator is designed to be used with the built-in self-test (BIST) architecture shown in Fig. 2.1. In the BIST architecture of Fig. 2.1, an input test pattern is generated in digital form in the test pattern generator (TPG), and then converted to an analog waveform in the digital to analog converter (DAC). The circuit under test (CUT) is then excited with this analog waveform. The output of the analog CUT is then converted back to digital format in the analog to digital converter (ADC) and analyzed in the output response analysis (ORA) portion of the system. The ORA then generates the output metrics, or output measures, from the raw data. The ORA data is then used to classify the analog circuit as fault-free or faulty.

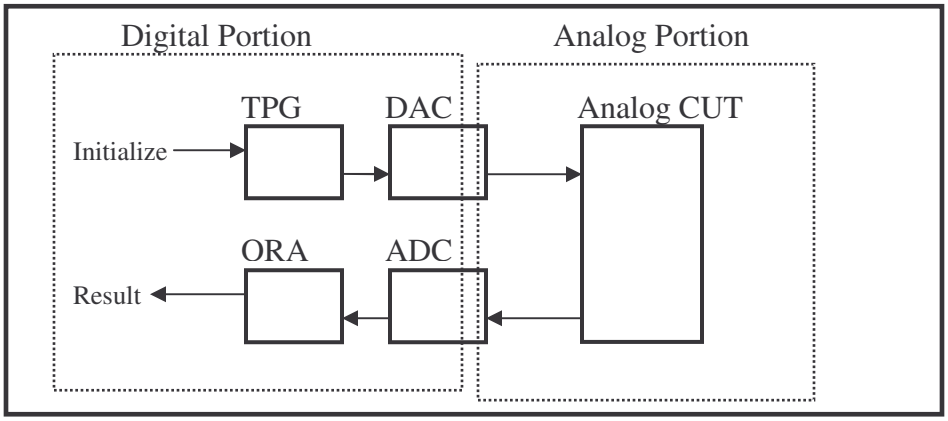


Figure 2.1 Built in self test (BIST) framework.

2.2 Fault Simulator Functional Flow

The fault simulator was designed to emulate the BIST framework of Fig. 2.1 and simulate the variations in thousands of randomized fault-free and faulty circuits with many different test waveforms (TPG) and different output metrics (ORA). The fault simulator generates the data needed to choose the best possible TPG test vector and best possible output metric, or to choose the best collection of TPG test vectors and output metrics.

In the fault simulator, the input circuit is first parsed into fundamental components, subsequently regenerating randomized versions of the circuit with, and without, faults. The randomizations emulate the normal variations of components in the circuit. The simulator also generates dozens of TPG waveforms to be combined with the randomized circuits, in all possible combinations. Lastly, a separate post processing program is used to convert the output data into histograms of the ORA output metric data to select the best vector and ORA metric and to determine the fault coverage it provides.

The fault simulator is written in the object oriented language of C++ for reusability, platform independence, and low maintenance requirements. The fault simulator is composed of a class library and an executable named faultsim. In addition, SPICE primitives are implemented as class objects in the class library. The C++ compiler and version used is the GNU complier version 3.1, a free ANSI package available to the public, without restrictions. Further details on the class library are found later in section 2.4.

A functional flow diagram of the simulator is given in Fig.2.2. In the first step of Fig. 2.2, the fault simulator takes a SPICE netlist as input. In the second step, the simulator parses the SPICE file into its fundamental components, such as FETS, resistors, capacitors, and inductors. [23].

In the third step of Fig. 2.2, the circuit file components are randomized parametrically and used to create a randomized batch of fault-free circuit files with no catastrophic faults. The randomization in this step is representative of normal variability in component values due to manufacturing processes. Then, faults are inserted by replacing components in the same set of parametrically randomized files with each catastrophic fault possible in the system.

In the “simulation” step of Fig. 2.2, the files are then simulated using the spice engine ELDO, the underlying component in the Accusim package from Mentor Graphics. The fault simulator currently uses Mentor Graphics ELDO, a proprietary product, but can potentially be run on any version of SPICE [24]. Additional inputs to the fault generator step include statistics describing the parametric variation of components and the set of TPG waveforms under consideration.

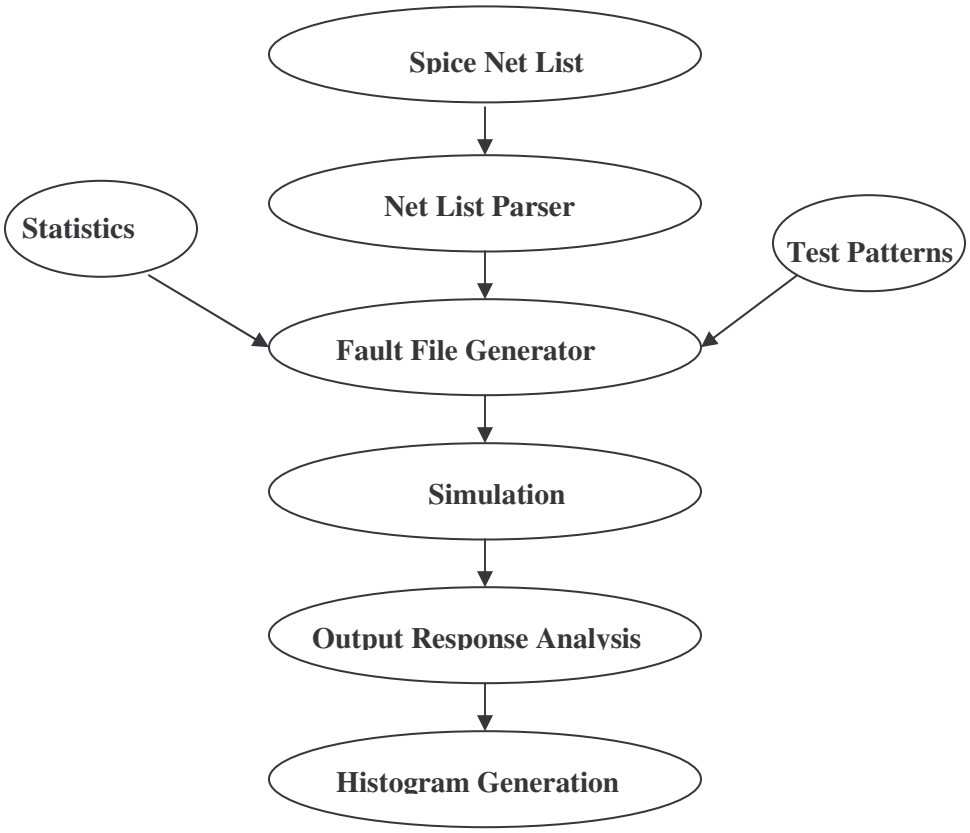


Figure 2.2 Fault simulator functional flow diagram.

In the next step of Fig. 2.2, Output Response Analysis (ORA), the output response from the simulations are used to compute the output metrics. The fault simulator produces output files including three ORA metrics, S_{out}, S_{del}, and S_{mag}, found in Table 2.1. In the Table, S_{out} is the sum of output voltages at each clock cycle for some number, N, of clock cycles. Similarly S_{del} is the sum of $V_{out} - V_{in}$ for some number, N, of clock cycles. Similarly, S_{mag} is the sum $|V_{out}-V_{in}|$. The number, N, of clock cycles for the summing of the ORA portion is variable and set at runtime.

Table 2.1
Floating point ORA metrics

$$S_{out} = \sum_{n=0}^{N-1} V_{out}[n]$$

$$S_{del} = \sum_{n=0}^{N-1} \left(V_{out}[n] - V_{in}[n]\right)$$

$$S_{mag} = \sum_{n=0}^{N-1} \left|V_{out}[n] - V_{in}[n]\right|$$

The summations in the ORA metrics of Table 2.1 are floating point summations that are often useful for the purpose of investigation. However, in actual implementation the output metrics of Fig. 2.1 are necessarily binary, with limited bit resolution. So, in addition to the floating-point values of Table 2.1, the binary equivalents of Table 2.2 are also computed during the fault simulation. These values are then representative of actual ORA data as would be measured in a hardware implementation.

In Table 2.2, S_{16out}, S_{16del}, and S_{16mag} are binary 16-bit equivalents of the metrics of Table 2.1, where $((x))_y$ is x modulo y [25]. Hence, $((x))_{65536}$ is the 16-bit binary representation of x. In Table 2.2, S_{16out} is the 16-bit binary sum of output voltages and each clock cycle for some number, N, of clock cycles. Similarly S_{16del} is the 16-bit binary sum of $V_{out} - V_{in}$ for sum number of clock cycles and S_{16mag} is the 16-bit binary sum $|V_{out}-V_{in}|$. The number of clock cycle durations are variable and set runtime. The analog voltages corresponding to 00 hex and FF hex at the input of the ADC and the output of the DAC in Fig. 2.1 are variables set at runtime.

Table 2.2
Digital ORA metrics

$$S_{16out} = \left(\left(\sum_{n=0}^{N-1} ((V_{out}[n]))_{256} \right) \right)_{65536}$$

$$S_{16del} = \left(\left(\sum_{n=0}^{N-1} \left((((V_{out}[n]))_{256} - ((V_{in}[n]))_{256} \right)_{65536} \right) \right) \right)_{65536}$$

$$S_{16mag} = \left(\left(\sum_{n=0}^{N-1} \left(\left| ((V_{out}[n]))_{256} - ((V_{in}[n]))_{256} \right| \right)_{65536} \right) \right)_{65536}$$

For the case of the S_{out} metric of Table 2.1, the floating point output voltage at each rising edge of the clock is added to the output voltage of each successive clock cycle until the number, N, of user defined TPG clock cycles have elapsed. The S_{out} ORA metric is then stored. For the case of the S_{16out} metric of Table 2.2, the 16 bit digital output voltage at each rising edge of the clock is added to the digital output voltage of each successive clock cycle until the number, N, of user defined TPG clock cycles have elapsed. The BIST system has multiple settings for the resolution of the ADC, DAC, and the accumulator. For the purposes of this thesis, the bit resolution of the ADC, DAC, and the final sum are limited to 8, 8, and 16 bits respectively.

The floating point calculation for S_{del} in Table 2.1 computes floating point V_{out}-V_{in} at each clock rising edge and sums them for the predetermined number, N, of clock cycles. Similarly, the floating point calculation for S_{mag} computes floating point $|V_{out}-V_{in}|$ at each clock cycle and sums them for the predetermined number, N, of clock cycles.

The 16 bit binary calculation for S_{16del} in Table 2.2 computes 1's compliment V_{out}-V_{in}. at each clock cycle and sums them for the predetermined number, N, of clock

cycles. The subtraction V_{out}-V_{in} is implemented using ones compliment subtraction where the two inputs are 8 bit unsigned and the output is 16 bit signed. Similarly, the floating point calculation for S_{mag} computes floating point |Vout-Vin| at each clock cycle and sums them for the predetermined number, N, of clock cycles.

After the "Output Response Analysis" step of Fig. 2.2, the ORA metrics associated with each circuit file are stored in ORA files that are later post-processed in the histogram generation step. In this final step of Fig. 2.2, histograms are generated showing the distributions of ORA metric values for circuits with, and without, faults. The histogram generation step is a separate software program to post-process the ORA data. From the ORA data, the mean and standard deviation can also be calculated. The end result is the statistical data that can be used to select the most effective TPG test vector and ORA output metric (S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, S_{16mag}), or collection of test vectors and ORA output metrics, for maximum fault coverage.

2.3 Fault Simulation Architecture

The overall operation of the fault simulator has been described in the previous section in a functional flow form. This section provides more detail regarding each of the pieces and how they operate together using a more detailed functional flow graph given in Fig. 2.3.

2.3.1 SPICE Input

In the first step of Fig. 2.3, the fault simulator reads in the SPICE netlist of the device under test(DUT) of Fig. 2.1. The simulator requires standard SPICE net-list

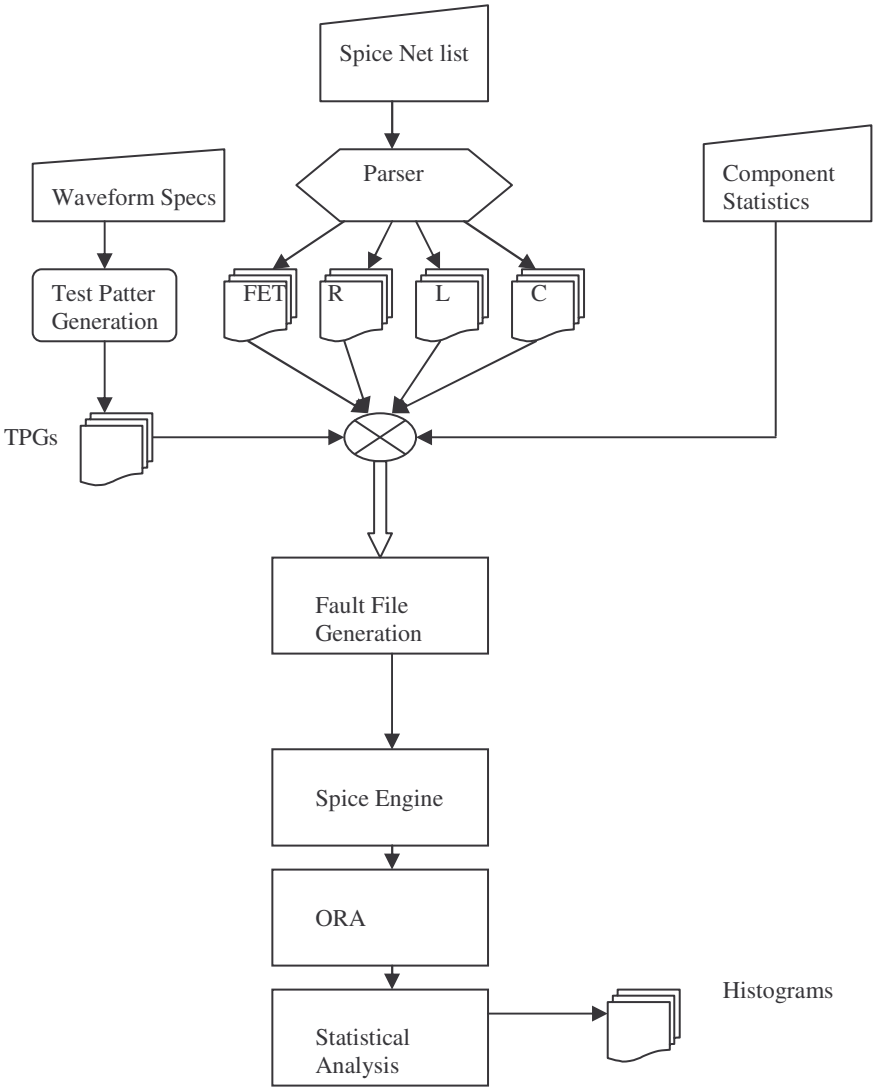


Figure 2.3 Fault simulator detailed functional flow graph.

format; schematic and VHDL entries cannot be read. SPICE netlist was chosen since it is widely used and standardized interface for describing the DUT of Fig. 2.1.

Spice was developed at the UC Berkley for circuit simulations and is available as open source to the public [23]. SPICE stands for Simulation Program with Integrated Circuit Emphasis. SPICE is a general purpose circuit simulation tool that will run

nonlinear DC, nonlinear transient and linear ac analysis. Circuits written using SPICE can include resistors, capacitors, inductors, mutual inductors, independent and dependent-current and voltage sources, lossy and lossless transmission lines, switches, uniform distributed RC lines, and five most common semiconductor devices; diodes, BJTs, JFETs, MESFETS, and MOSFETS. A SPICE net list file includes but is not limited to information about components, nodal connections, voltage levels, model information, various parameters, and input/output parameters.

2.3.2 Parser

In the second step of Fig. 2.3, the parser takes the circuit and reads each line of the SPICE file. The parser stores all of the information given by the file such as node connections, component values, and other parameters. The parser recognizes components such as R, L, C and converts them into corresponding C++ class library objects, as indicated by the FET, R, L, and C objects below the parser in Fig.2.3

2.3.2.1 Component Statistics

In the upper right of Fig. 2.3, the fault simulator then loads the statistical data required to introduce the parametric variations into the component values. The SPICE file is randomized the number of times specified, with components varied using a uniform or Gaussian distribution according to the specified component statistics. The SPICE netlist file is reconstituted for each catastrophic fault and randomized with the fault by the number of parametric randomizations in the simulation. In this, phase of the simulation, the statistics module of the Faultsim library calculates the appropriate Gaussian distributed values for the components reflecting the tolerances of the corresponding manufacturing processes.

Resistive, inductive, and capacitive components are randomized parametrically according to the default parameters shown in Table 2.3. The components are randomized based on process (i.e., lot-to-lot) statistics and single-chip (i.e., within-a-chip) statistics. A given process will have a Gaussian probability distribution function (pdf) as shown in the top plot of Fig. 2.4. The second plot in Fig. 2.4 shows the pdf of component values on a single chip where the values of the components of a single chip track each other as a result of the processing. This distribution is much tighter, since devices on a single chip will tend to track each other. The bottom plot shows that a distribution for another chip may have a different mean value, but again with the tighter distribution.

Table 2.3
Process statistics

Component	Process(chip-to-chip) statistics			Single-chip(within-a-chip) statistics		
	PDF	μ	σ	PDF	μ	σ
Resistors	Gaussian	1	0.1	Gaussian	1	0.04
Capacitors	Gaussian	1	0.11	Gaussian	1	0.03
Inductors	Gauss	1	0.12	Gaussian	1	0.02

In Table 2.3, the process or, lot-to-lot, variation of resistors are defined with a mean of 1 and a standard deviation of 0.1. This defines a Gaussian distribution of resistor values that vary around their nominal value with a standard deviation of 10 percent. For any given single chip, resistors have a Gaussian distribution center around their nominal value of four percent as indicated by the single-chip statistics column of Table 2.3 with mean 1 and a standard deviation of 0.04 for the resistors. Similarly, capacitor-inductor variation is defined by Table 2.3 for lot-to-lot and single chip.

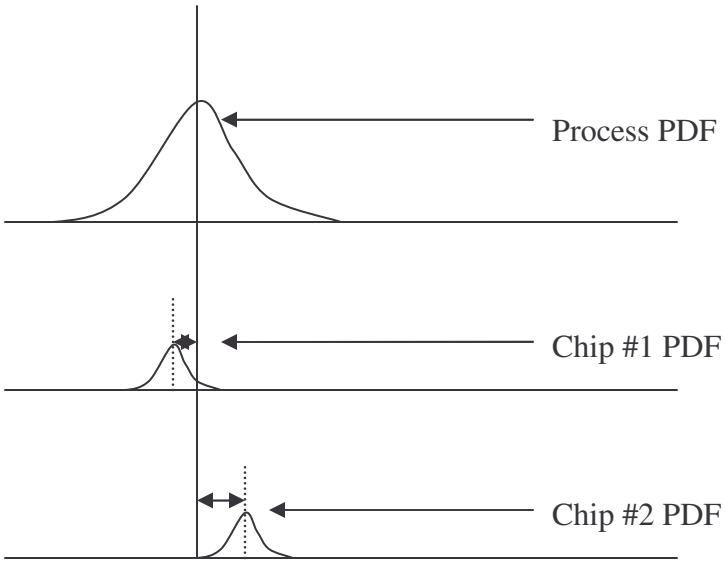


Figure 2.4 Statistical models showing process pdf (probability density function) in upper trace and two lower traces illustrating single chip pdf's for two different chips. Lower two traces indicate component variations within a single chip or integrated circuit.

In this, it is not likely for the resistors of a chip to be scattered across the full range of lot-to-lot variation. The processing of chips tends to bias component values on a single chip in the same direction giving each chip its own tighter distribution with a standard deviation smaller than lot-to-lot variance. The software implements randomization of chips as illustrated in the two lower plots of Fig. 2.4.

The parser uses as many library modules as needed for reading and faulting the spice net-list files. The library contains classes for implementing resistors, capacitors, inductors, transistors, and different types of sources. When reading a file, if a resistive component is encountered the parser instantiates a Resistor class object from the Faultsim library, which will store the device information for later parsing. Similarly, other SPICE components are implemented as C++ objects in the fault simulation library.

2.3.3 Test Pattern Generation

The upper left of Fig. 2.3 illustrates generating TPG patterns for the fault simulation. The candidate test patterns can be found in Appendix B. These patterns include saw-tooth, frequency sweep, and random waveforms of variable amplitude and frequency. TPG patterns are generated by the TPG class object of the Faultsim library as a piece-wise linear waveform. During simulation, the TPG files are then combined with the aforementioned randomized circuit files generating all possible combinations of parametrically randomized circuits, and TPG waveforms, and faults. The TPG waveforms are stored in files according to waveform, amplitude, and frequency. The clock included in each simulation takes 256 clock cycles(to count through 2^8 bits), to drive the DAC of Fig. 2.1, making the effective waveform frequency equal to the clock frequency divided by 256. This simple relationship between the clock and waveform frequency however is not true for certain waveforms such as frequency sweep. In addition, options in the fault simulator allow for varied number of repetitions for the test pattern waveform, and hold off and time before collecting ORA data.

2.3.4 Fault File Generation

In the fault file generation step of Fig. 2.3, hard faults are injected into copies of the original circuit for resistors, capacitors, inductors, and MOSFETS. A resistive catastrophic fault is emulated by placing a 100MΩ in place of the original resistor value for an open, and a 1Ω resistor for a short. At present, capacitive opens and inductive shorts are simulated as a 2 fF capacitor and 2 fH inductor, respectively. However, capacitive shorts and inductive opens are temporarily implemented as 2 Farad capacitor and 2 Henry inductor, respectively, and remain for future implementations otherwise.

For transistors, the stuck-off transistor level fault emulated by disconnecting the transistor from the circuit with a 100 Meg ohm resistor as shown in Fig. 2.5. The stuck-on transistor level fault is emulated by a 1 ohm resistor between the source and drain of the transistor as shown in Fig. 2.6. [2]

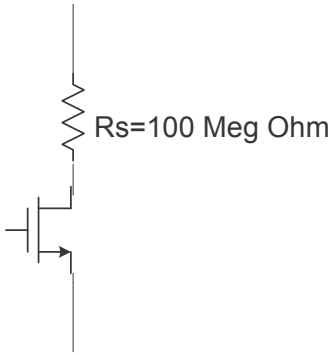


Figure 2.5 Stuck-off fault for transistor illustrating 100 MΩ series resistor used to implement fault.

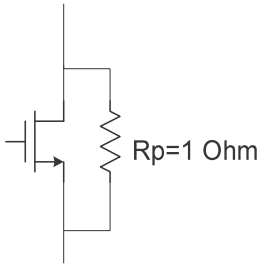


Figure 2.6 Stuck-on fault for transistor illustrating 1 Ω parallel source drain resistor used to implement fault.

2.3.5 SPICE Engine

Following the fault file generation step of Fig. 2.3, the SPICE engine used to do the simulation is the Mentor Graphics ELDO tool [24]. The main fault simulator, Faultsim, executable calls ELDO for all simulations using the command "ELDO filename.cir", where filename.cir is the spice file to be simulated. The ELDO simulation

then produces an output file that contains all of the voltage and time data that will be used to calculate the ORA metrics.

2.3.6 ORA

In the next step of Fig. 2.3, once a simulation is complete the six ORA metrics, S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, S_{16mag} shown in Table 2.1 and 2.2 respectively, are then computed and stored in an ORA file. The ORA files are organized by TPG waveform for later classification of test vector efficacy and fault coverage. Each ORA file contains all the ORA data for the particular TPG pattern and for all fault conditions. Each line of an ORA file contains eight items; the circuit filename, the test vector name, and the ORA metric values (S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, S_{16mag}) for that combination.

2.3.7 Statistical Analysis

In the final step of Fig. 2.3, statistical analysis, the ORA files are post processed using a separate executable program named anarun. The anarun executable is used to produce histograms of the ORA metrics ($S_{out,}$ $S_{del,}$ $S_{mag,}$ $S_{16out,}$ $S_{16del,}$ and S_{16mag}) of Table 2.1 and 2.2 which provide a graphical and statistical insight into the efficacy of the test vector. This post-processing tool also computes the mean, standard deviation, and variance of the ORA data, as well as, arrange the data for importation to spreadsheet tools for graphical interpretation as a histogram.

2.4 Class Objects

As mentioned beforehand most of the functionality in the fault simulator is implemented as C++ class objects in the Faultsim library. The following sections give brief descriptions of each of the classes and their functions.

2.4.1 Class Resistor

Class Resistor implements the resistor and contains data for the name, resistance value, and nodes of the device. The object also includes statistical parameters, and member functions for reading, writing, and randomizing a resistor.

2.4.2 Class Inductor

Class Inductor works much in the same way that class resistor does. Class inductor contains data for the name, inductance value, and nodes of the devices. The object also includes statistical parameters, and member functions for reading writing, and randomizing a inductor.

2.4.3 Class Capacitor

Class Capacitor, much like classes inductor and resistor, contains data for the name, capacitance value, and nodes of the devices. The object also includes statistical parameters, and member functions for reading writing, and randomizing a capacitor.

2.4.4 Class Ora

This class object processes the output files from the spice engine, the .chi files, to produce the ORA metrics found in the ORA files. This class contains member functions to search the .chi files for the time and voltage data to compute the output metrics in Table 2.1 and Table 2.2. This class computes the floating point analog ORA metrics as well as the digital metrics. The class object also deletes the rather bulky .chi files after storing the results of the ORA metrics in the ORA files.

2.4.5 Class Gsrc

This class object implements the G-model of spice, or a voltage controlled current source. This class contains member functions to process the nodes of the input and

output along with the transconductance of the model. At present the object does not introduce faults or randomize this component.

2.4.6 Class Vsrc

This class object implements the independent voltage and stimulus source in SPICE. This class contains the data for reading, storing, and writing the nodes and voltage values of independent voltage sources. At present these components do not implement fault or randomization.

2.4.7 Class Circuit

This class handles the circuit level functions of the parser. The class objects loads circuit level objects and segments the tasks of to lower level objects contained therein. This class contains the data for the complete netlist in the form of a collection of objects appropriate to the corresponding components of the original.

2.4.8 Class DotEnds

This class implements the ".ends" statement that signals the end of a sub-circuit in spice. This object performs the task reading, storing, and writing the ".ends" statements found in SPICE files.

2.4.9 Class Other

This class implements any SPICE token that is not implemented in one of the other class libraries. This class contains the functions and data for reading, storing, and writing all SPICE statements listed in the Other class member functions. These statements include, but are not limited to .ac, .dc, .plot, .print, .probe, .step, .temp etc. There are no functions for faults or randomization in this class.

2.4.10 Class Xsubckt

This class is used for SPICE sub-circuit level statements which typically correspond to one line in a SPICE file. This class contains the member functions for reading, storing, and writing the sub-circuit level statements found in SPICE files.

2.4.11 Class CircuitStats

This class contains the all the functions and data to implement the process (lot-to-lot) and single-chip (within a single Integrated circuit) statistical characteristics of the randomizer as described in section 2.3.2. This class contains a wide array of statistical functions to implement within-chip as well as process (chip-to-chip) pdf's into the randomization of the SPICE file.

2.4.12 Class DotSubckt

Class DotSubckt implements the SPICE statement ".subckt," a definition for a sub-circuit in a SPICE file. This class contains the functions and data for reading, storing, and writing the nodes and names of the sub-circuits of SPICE. This class object has no faults or randomization.

2.4.13 Class Isrc

Class Isrc implements the SPICE independent current source statements found in SPICE files. This class contains the functions and data for reading, storing, and writing the nodes and values of independent current sources. This class has no faults or randomization.

2.4.14 Class Comment

Class Comment implements the comments of SPICE that are denoted by the "*" character. This class contains the member functions to identify comments, read

comments, and write comments. This class deals only with the contents of comments and therefore has no statistical functions.

2.4.15 Class Esrc

The Esrc implements the SPICE voltage controlled voltage source statements. This class contains the member functions to recognize, read, store, and write E models found in SPICE files. At present, there are no faults or randomization built into this class.

2.4.16 Class Statistics

Class Statistics implements statistical functions that are used in CircuitStats class to generate randomized values from the statistical mean and standard deviation. This class and its member functions do not operate directly on any given component but do serve as support to the process of randomization.

2.4.17 Class Component

Class Component identifies each component level object that would be found in a SPICE netlist and acts as a container for particular components. Class component contains the member functions to identify components and execute the proper member classes according to the type of component. This class does not implement statistical randomization or faults.

2.4.18 Class Mos

Class Mos implements the MOSFET component and contains data for the name, channel size, and nodes of the device. The object also includes statistical parameters, and member functions for reading, writing, and randomizing a MOSFET transistor. This class also implements the stuck-on and stuck-off faults illustrated in Figs. 2.5 and 2.6.

2.4.19 Class Faultlist

Class Faultlist implements the generation of fault lists based on the circuit being simulated. This class contains the functions for generating fault lists for shorts and opens of R, L, C, and MOS components.

2.4.20 Class Tpg

Class Tpg contains the member functions that create TPG test waveforms based on input parameters of clock frequency, waveform, amplitude, repetitions, and hold off for ORA calculations. The Tpg class generates piecewise linear stimulus in SPICE format and can store the SPICE code in files.

2.4.21 Class Data

Class Data implements the various functions needed for processing generic data arrays used in various places throughout the fault simulator.

2.5.1 Computational Complexity

The task of fault simulation presents formidable computational complexity for even modest circuits. To illustrate this complexity, consider the operational amplifier shown in Fig. 4.1, with only 11 components. If each component has two faults, an open and a short, there are 22 potential faults for the circuit. Two hundred randomized netlist files for each fault plus the fault-free file would result in (22+1)×200=4600 circuit files. If each file is simulated at three frequencies, 10 waveforms, and two amplitudes there are (22+1) ×200×3×10×2= 276,000 netlist files to be simulated.

2.5.2 Computational Time

The foregoing complexity leads to long computational times. In experiments, the operational amplifier circuit of the previous section has an average simulation time of

eight seconds based on current computing equipment. The computing environment available is a Sun Microsystems SPARC Ultra 80 with four 450 MHz processors and 2 GB of main memory. If the 276,000 circuit files were simulated sequentially this would amount to 25.5 days to simulate the circuits. Also, some waveforms take considerably longer to simulate than others depending on faults and various other conditions such as frequency and amplitude. Therefore, an approximation of the time for simulating the operational amplifier of Fig.4.1 is nearly one month. The linear nature of the problem dictates that doubling the number of components would double the simulation time, meaning a circuit with 22 components would require nearly 2 months to simulate.

2.5.3 Parallel Processing

The huge simulation times associated with even small circuits (as outlined in the previous section) provides impetus to use parallel processing in the spice engine portion of Fig. 2.3. This parallelization effort provided for a linear reduction in the amount of time based on the number of parallel threads up to the number of processes on the parallel processing machine. A simulation with two threads ran twice as fast with a simulation with one thread. Experimental results show this to be true when simulating the BiQuad filter shown in Fig. 3.7. The BiQuad filter with 15 components, 160 randomizations, 3 frequencies, and one waveform can be simulated in one day with 4 threads versus 3 days with one thread on a 4-processor machine.

The fault simulator was parallelized by making changes to the main executable faultsim. Parallel threads were implemented with the functions fork() and wait() which spawn parallel processes and close them when they are finished respectively. The program allows a user definable number of parallel processes, threads to be used for a

simulation. The program then creates a process subdirectory for each thread. After the parallel threads finish, the results are then collected and stored in a separate common ORA subdirectory incrementally as the circuit simulations complete for all possible combinations of TPG waveforms, faults, and parametric randomizations.

2.6 Faultsim

Faultsim is the main executable of the fault simulator. Faultsim controls the flow of Fig. 2.2 and Fig. 2.3. The only flow not controlled by faultsim is the post processing executable anarun, the program that generates the histogram for the ORA results as the last steps of Figs. 2.2 and 2.3.

```
%faultsim
faultsim
faultsim usage:
== faultsim ckt.cir numproc numrand inpos inneg outpos outneg vbias
vamp maxcpu
vomin vomax
      - ckt.cir is the circuit spicefile
      - numproc is number of processes that are forked()
         to run in parallel on a multi-cpu machine
      - numrand is number of randomizations per fault
      - inpos,inneg are pos and neg differential input nodes
      - outpos,outneg are pos and neg differential output nodes
      - vbias test pattern dc bias, where
      inpos=vbias+(vampl/2),inneg=vbias-(vampl/2)
         where inpos=vbias+/-(vampl/2),inneg=vbias
         if vbias=0, input is true floating input
      - vamp test pattern amplitude in volts
      - maxcpu is max number cpu seconds allowed per spice run
      - vomin to vomax is differential output voltage range
      - numrep number or repetitions to execute TPG

 Example:
      faultsim benchmark.cir 4 4 17 18 16 0 2.5 0.2 120 0 5 1
%
```

Figure 2.7 Faultsim help screen showing description of command-line parameters and example command-line.

Typing faultsim at the command line, as shown in Fig. 2.7, will display a help screen that lists the command-line parameters and a brief explanation of each. Faultsim is controlled by the following parameters: ckt.cir, numproc, numrand, inpos, inneg, outpost, outneg, vbias, vamp, maxcpu, vomin, vomax, and numrep. The first parameter ckt.cir is the name of the SPICE netlist file that must be in the directory in which faultsim is running. The next parameter, numproc, is the number of processes, or threads that faultsim will spawn in the simulation stage. The parameter numrand is the number of randomizations for each fault and for the good circuit. The parameters inpos and inneg are the node names of the positive and negative input voltages of the SPICE file. These are used to merge the TPG waveforms (as SPICE commands) to the circuit input. The parameters outpos and outneg are the positive and negative output nodes at which the output will be taken and analyzed. The parameter vbias is the DC bias level of input TPG waveform. The parameter vamp is the amplitude of the input TPG waveform to be tested. The parameter maxcpu sets the timeout of a single ELDO simulation. This is often necessary as certain faults will cause simulations not to converge or converge slowly. The parameters vomin and vomax set the voltage range of the ADC in the BIST framework. The last parameter, numrep, is the number of repetitions that the simulation will process of each waveform (each repetition being 256 clock cycles).

CHAPTER 3: EXPERIMENTAL RESULTS

The fault simulator results were compared to theoretical and experimental hardware results for a BiQuad filter. The hardware BiQuad filter was designed and built by Clark Hopper and Steven Tucker working under the direction of Dr. David Binkley.

In this chapter, the fault simulator is validated by comparison to theoretical analysis (hand calculations) and experimental hardware. The focus is on validating the ability of the simulator to predict hardware functionality and therefore a particular TPG waveform is employed which may or may not be the best test vector for this circuit. Nevertheless, the chosen waveform suffices for the purpose of validating the fault simulator.

3.1 Biquadratic Filter Circuit

The circuit used for the verification of the fault simulator was a Kerwon-Huelsman-Newcomb biquadratic filter shown in Fig. 3.1[22]. The BiQuad filter has band pass, low pass, and a high pass outputs. The cutoff frequency for all three filter types is set by R and C from the components in feedback of the circuit shown in Fig. 3.1.

$$\omega_o = \frac{1}{RC} = \frac{1}{10k \times .015\mu} = 6.7 \times 10^3 \qquad (3.1)$$

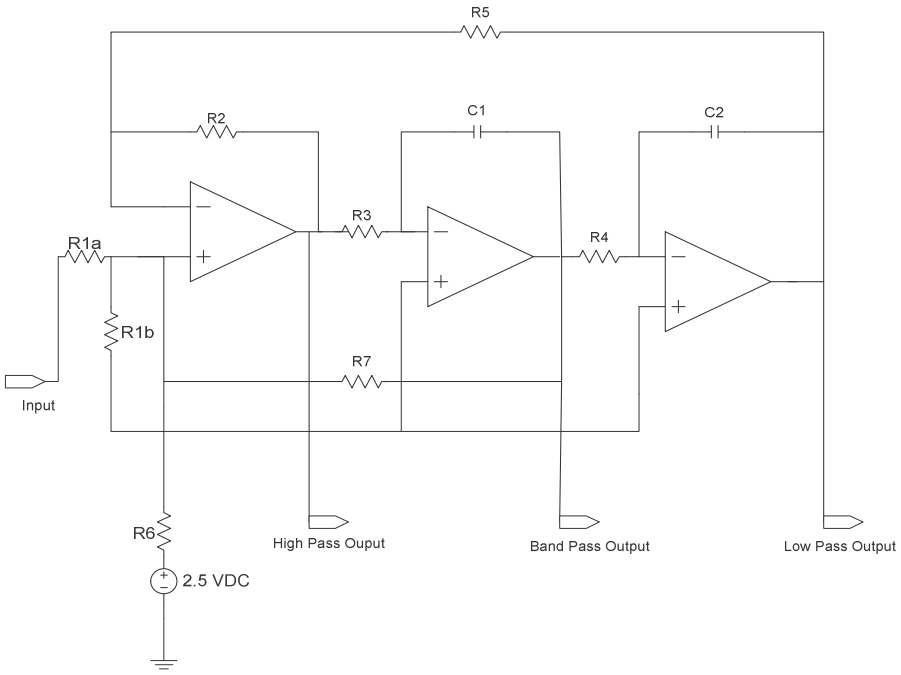


Figure 3.1 BiQuad filter circuit used for benchmarking fault simulator against hardware.

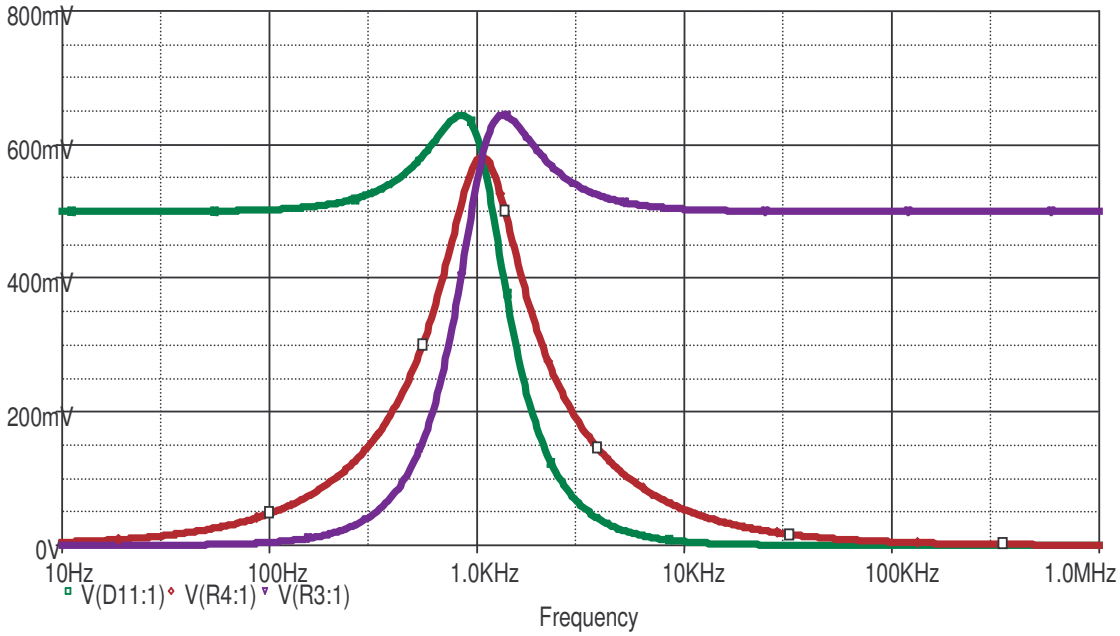


Figure 3.2 Frequency response of BiQuad filter shown in Fig. 3.1 with 1 kHz cutoff frequency for high-pass, low-pass, and band-pass.

The cutoff frequency for the experimental hardware is 1 kHz as shown by the frequency response plot of Fig. 3.2. Fig. 3.2 is the frequency response of the circuit of Fig. 3.1, showing the band-pass, low-pass, and high-pass cutoff frequency of 1 kHz. The frequency response plot also shows the circuit has a gain of approximately one half for all the pass band regions. The gain of the BiQuad filter is set by the components R1b and R7 of the circuit shown in Fig. 3.1[22].

$$gain = 2 - \frac{2}{\frac{R_2}{R_3} + 1} = 2 - \frac{2}{\frac{7.5k\Omega}{20k\Omega} + 1} = .55 \qquad (3.2)$$

3.2 Modeling the BiQuad Efficiently

The BiQuad filter circuit of Fig. 3.1 used to collect hardware experimental results was implemented with AD820A/AD operational amplifier, shown in the operational amplifier circuit of Fig. 3.3.

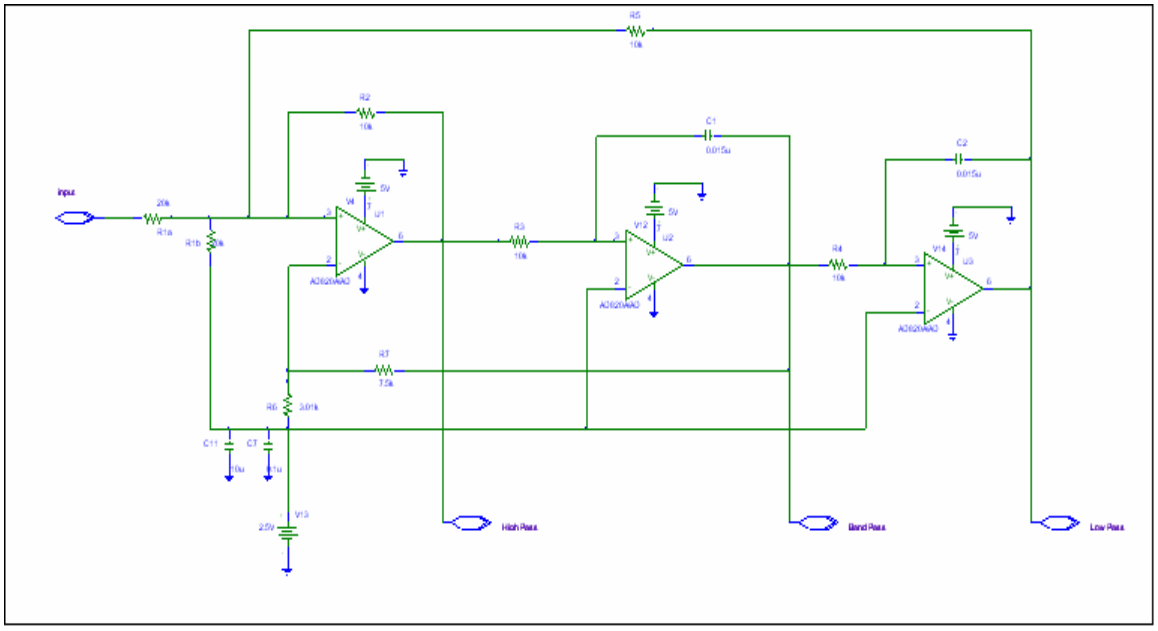


Figure 3.3 BiQuad hardware implementation using AD820 amplifier.

The AD820A/AD contains 26 transistors each, making the fault simulation of all internal components of the three AD820A/AD amplifiers far too time consuming. Furthermore, the test lab would be unable to test faults at the transistor level for the AD820A/AD package for comparison against fault simulator data. For these reasons, a reduced order model was implemented to emulate the AD820/AD amplifier with a voltage controlled voltage source (VCVS), a current limiting resistor, two diodes and a five volt supply as shown in Fig. 3.4.

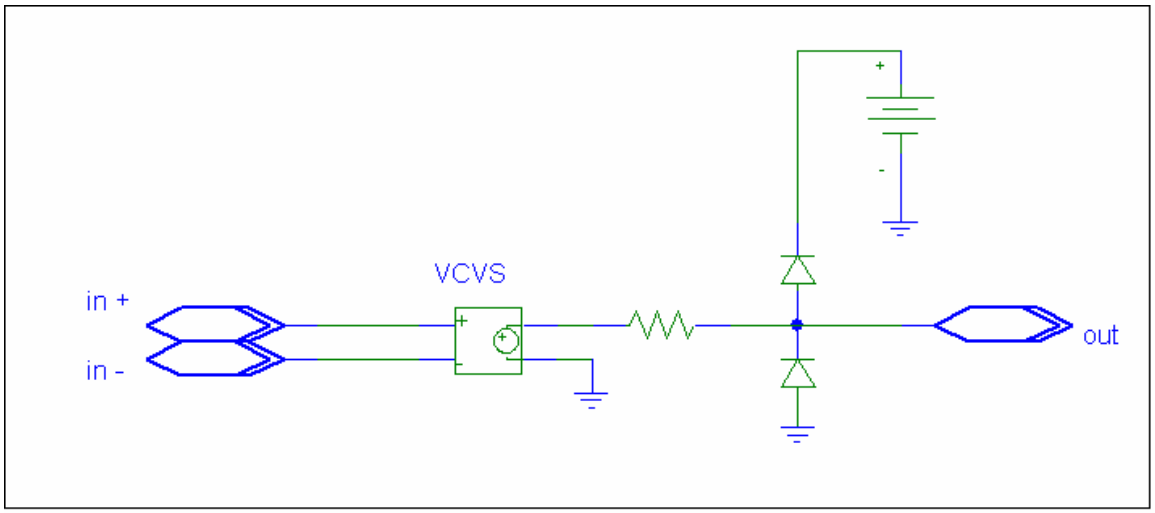


Figure 3.4 Emulation of AD820A amplifier used in BiQuad circuit design ideal voltage controlled voltage source, two diodes and output impedance resistor.

In Fig. 3.4, the AD820A/AD is modeled by a (VCVS) along with diodes to cause clipping, and resistive output impedance. In Fig. 3.4, the VCVS emulation of the AD820/AD, the input signals are applied to the positive and negative input nodes of the VCVS. The gain of the VCVS is set to 10^6. One problem with using the ideal VCVS is the model has no mechanism for clipping, and so the diode network is added to induce clipping. When the output voltage of Fig. 3.4 goes below -.5 volts, which is below the

threshold voltage for the diode, the diode to ground turns on and shorts the output of the circuit to ground. When the output voltage goes above 5.5 volts and breaks the threshold of the diode to the 5-volt supply, the diode limits the output of the E-source to the 5-volt supply. These diodes keep the output of the model of the amplifier, in Fig. 3.4, in the range of 0 to 5 V similar to the AD820A/AD. In future work the 5 V source will be dropped to 4.5 V so the voltage clips at 5 V, similarly the grounded terminal should be set to .5 V so the other rail clips at 0 V.

Another modification to the circuit of Fig. 3.1 included adding an inverting VCVS to the front-end of the BiQuad filter to account for the inverting amplifier used in the experimental hardware, shown following the DAC in the schematic of Fig. 3.18. The VCVS with gain of one, shown in Fig. 3.5, has the input signal connected to the positive input node of the VCVS with the negative input of the VCVS tied to the 5 volt supply.

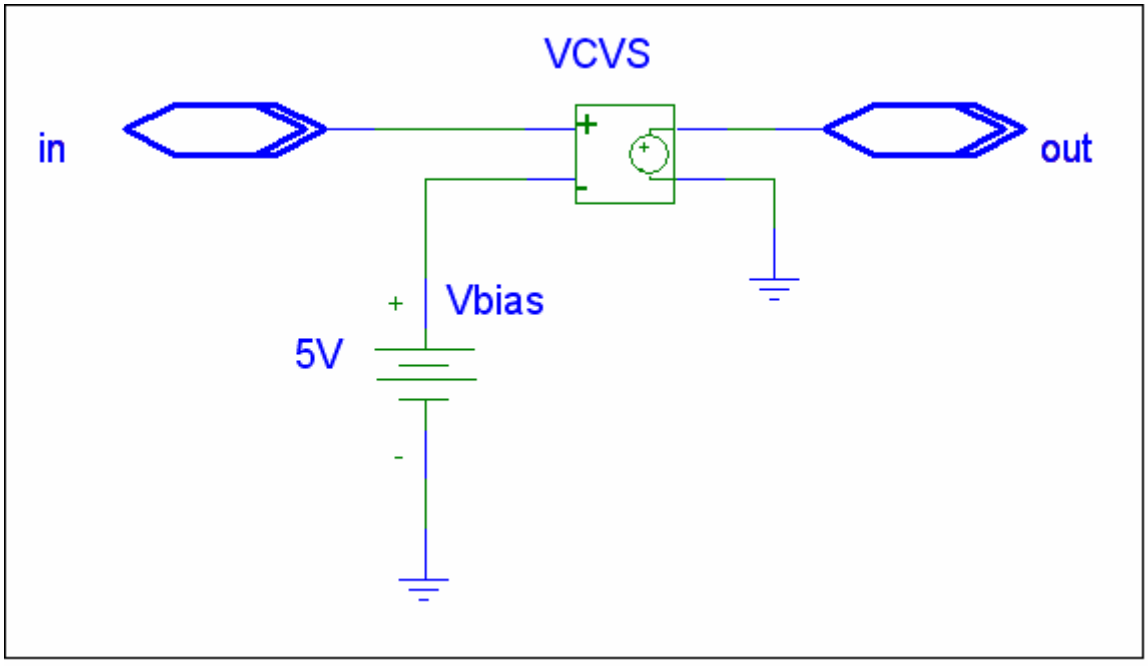


Figure 3.5 Inverting amplifier for input of BiQuad circuit used to translate 0 to 5 volt input to 5 to 0 volt output. Gain of VCVS is -1.

The circuit of Fig. 3.5, inverts the input signal and adds a 2.5-V DC offset. When the input is 0 volts the output becomes 5 volts, when the input is 5 volts the output will be 0. The negative terminal of the VCVS output is connected to ground and the final output signal of Fig. 3.5 is taken from the positive output. The aforementioned changes to the circuit are to expedite fault simulation by faster circuit simulation times and to provide models representative of the behavior of the original hardware experimental circuit.

Figure 3.18 contains hardware realization used the BIST system of Fig. 2.1. This system was used to obtain the hardware experimental results and was also the model used for the fault simulator. In Fig. 3.18, found at the end of the chapter, the full hardware realization of the fault simulator containing the inverting amp of Fig. 3.5, the DAC and ADC of Fig. 2.1, the BiQuad of Fig. 3.1, and the TPG discussed in section 2.3.3.

3.3 Theoretical Results (Hand Calculations)

In this section, theoretical results are presented for several faults, so later these results may be compared with simulation results to validate the fault simulator. In particular, theoretical results are calculated for all six ORA metrics for three faults which produced the same output condition. These three faults were selected because of the simplicity of the hand calculations. The calculations illustrate ORA computation and provide baseline theoretical values for the ORA metrics. For the purposes of discussion and verification, the waveform employed for testing was the count-up waveform (Cup waveform Appendix B) at 19.5kHz effective waveform frequency (5MHz TPG clock frequency), 5 V amplitude, 2.5V DC offset, for one cycle (256 clock cycles) with no hold-off for initialization. Unless otherwise indicated, the remainder of this chapter

discusses results for this TPG waveform with the aforementioned conditions and refer to the high pass output of Fig. 3.1.

3.3.1 Calculation for Specific Fault Conditions

For the case of the circuit of Fig. 3.1 without faults, a SPICE simulation showing the input as the lower trace and high pass output as the upper trace is given in Fig. 3.6.

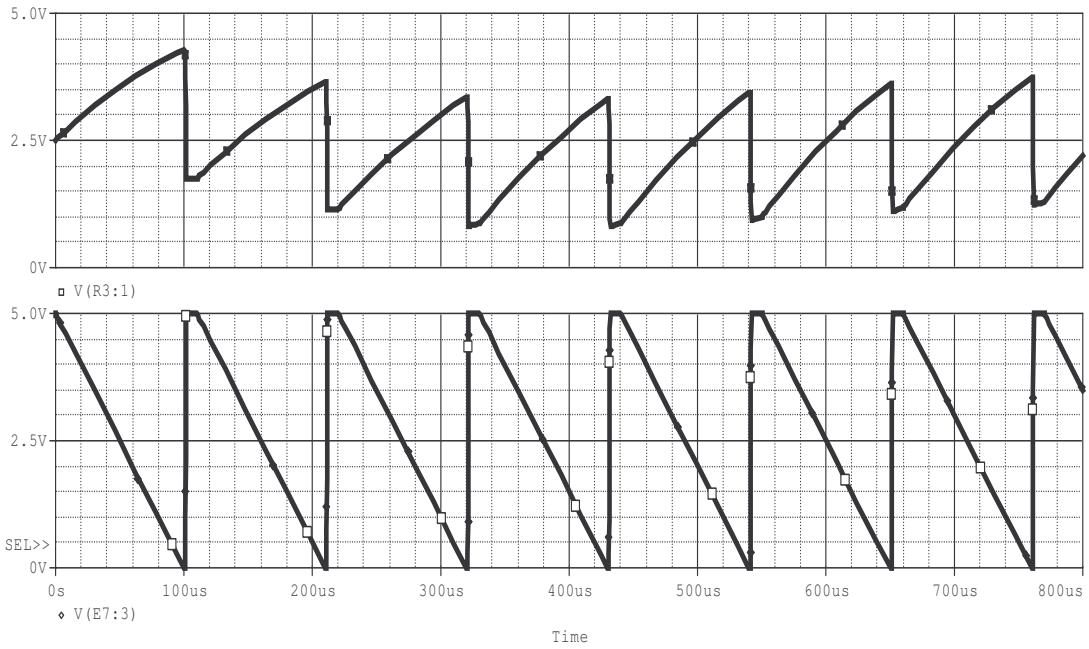


Figure 3.6 SPICE simulation of circuit of BiQuad showing input (lower trace) and output (upper trace). Transient output response due to initial conditions is visible in the output plot.

The saw-tooth waveform in the lower trace in Fig. 3.6 going from 0 to 5 volts at a frequency of 19.5 kHz is the input signal to the BiQuad circuit of Fig. 3.1. The high pass output is the saw-tooth signal in the upper trace of Fig. 3.6, with 2.5 V peak-to-peak centered around 2.5 volts. The 2.5 V peak-to-peak value of the output is consistent with the predicted gain of .5 for the BiQuad circuit. The DC offset of 2.5 in the upper trace of

Fig. 3.6 is also consistent with the expected operation of the BiQuad filter circuit shown in Fig. 3.3, given the 2.5-V virtual ground shown biasing the filter. Although a high-pass circuit should ideally have no DC offset, it can be seen in Fig. 3.6 that the output of the circuit is initially offset above the 2.5 V DC virtual ground, and then levels out centered at 2.5 V around 750µs. This can be attributed to the transients of the circuit, and these effects are later considered in section 3.6 when comparing the ORA metrics with the experimental hardware. Since the simulation was run with no hold off and for only one cycle, these transient effects can be expected to contribute to the measured ORA metrics.

Table 3.1 lists the faults for the BiQuad circuit of Fig. 3.3. From the fault list shown in Table 3.1, three faults were selected that give the output condition of a constant 2.5 volts, as in Fig. 3.7.

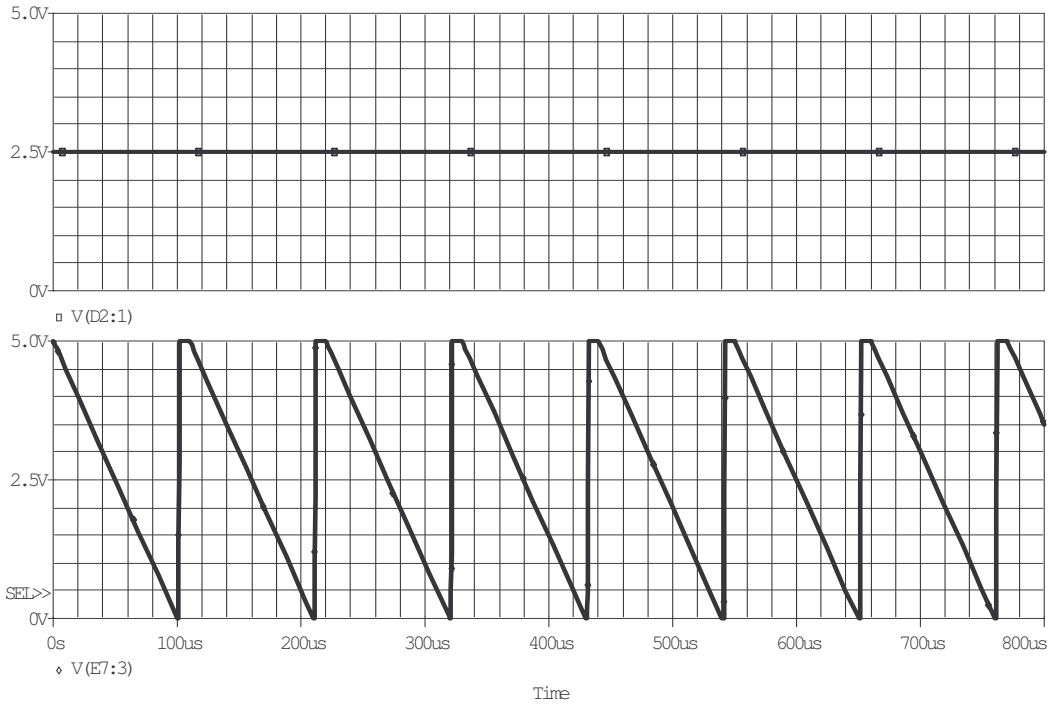


Figure 3.7 SPICE plot showing input (lower trace) and output (upper trrace) of faulty BiQuad circuit with 2.5-V DC output condition.

The three faults that produce this 2.5-V constant output condition are R3 short, R2 short, and R1b short shown in Table 3.2 below.

Table 3.1
Fault List for BiQuad filter circuit simulation

Fault List	
Component	Fault
R3	Open
R3	Short
R4	Open
R4	Short
R5	Open
R5	Short
R6	Open
R6	Short
R2	Short
R7	Open
R7	Short
C1	Open
C2	Open

Table 3.2
List of Specific Components and Faults for ORA Confirmation were V_{out} is a constant 2.5 VDC

Component	Fault	Output Condition
R2	Short	2.5 volts
R3	Short	2.5 volts
R1b	Short	2.5 volts

The SPICE simulation of the three fault conditions shown in Table 3.2 produced the output (2.5-VDC) shown in the upper trace of Fig. 3.7, with a saw-tooth input shown in the lower trace. These faults were chosen because they offer easily calculated ORA metrics.

3.3.1.1 S_{out} Floating Point Calculation

The cases of Table 3.2 with an output voltage stuck at 2.5 V and with the input going from 0 to 5 volts over 256 clock cycles is a useful condition for validation since it provides simple hand calculations. In this case, S_{out} becomes:

$$S_{out} = \sum_{n=0}^{N-1} V_{out}[n] = \sum_{n=0}^{255} 2.5 = 256 \times 2.5 = 640 .$$

For more complex signals, the summation can be viewed as an integral or area under the V_{out} curve. For the present example, the S_{out} metric is analogous to the area of the rectangle (i.e. the integral of the rectangle) shown in Fig. 3.8. Given that the result is

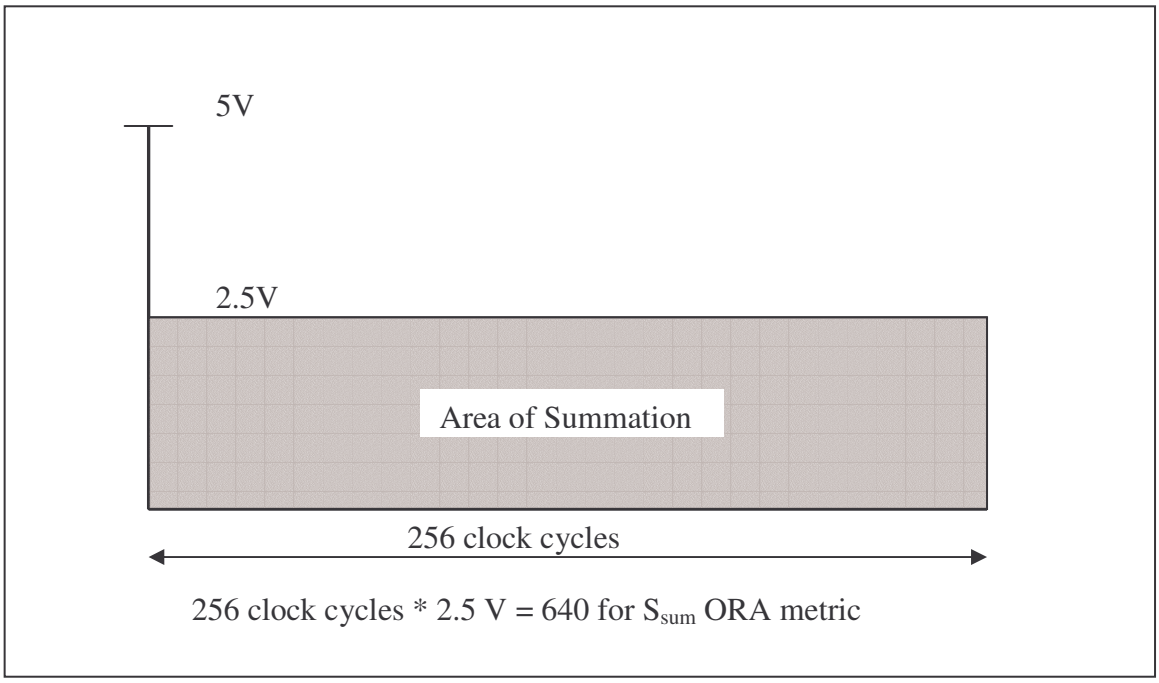


Figure 3.8 Illustration showing area corresponding to the fault for 2.5-V output condition of Fig. 3.7 over 256 clock cycles for computing S_{out} metric. The area is 2.5x256=640.

for one TPG waveform cycle only (256 clock cycles), S_{out} would equal the area of a

rectangle with base=256 clock cycles and height=2.5 V with area 256x2.5=640. This is the area of the rectangle in Fig. 3.8.

3.3.1.2 S_{del} Floating Point

The cases of Table 3.2 with an output voltage stuck at 2.5 V are used for S_{del} since it provides simple hand calculations. The S_{del} metric subtracts the voltage of the input from the output voltage at each clock cycle and sums the result over the number of specified clock cycles. Only one cycle of the TPG waveform (256 clock cycles) is used for the simulation under consideration. In the upper left of Fig. 3.9, V_{out} is represented as a 2.5-VDC constant for 256 clock cycles. In the upper right, V_{in} is represented as a ramp from 0 to 5 volts for those same 256 clock cycles. At the bottom is the difference of the upper two figures representing $V_{out} - V_{in}$ for same 256 clock cycles. In this case, S_{del} becomes:

$$S_{del} = \sum_{n=0}^{N-1} \left(V_{out}[n] - V_{in}[n] \right) = \sum_{n=0}^{255} \left(2.5 - \frac{5n}{255} \right) = 256 \times 2.5 - \sum_{n=0}^{255} \frac{5n}{255} = 0 .$$

For more complex signals, the summation can be viewed as an integral or area under the V_{out} - V_{in}. For the present example, the S_{del} metric is analogous to the area under the curve at the bottom of Fig. 3.9. Given that there is equal positive and negative area under the curve of Fig. 3.9, the net result should be at or near 0.

3.3.1.3 S_{mag} Floating Point

The cases of Table 3.2, with an output voltage stuck at 2.5 V, are used for S_{mag} since it provides simple hand calculations. The S_{mag} metric sums the absolute value of the difference of the input subtracted from the output. Only one cycle of the TPG waveform (256 clock cycles) is used for the simulation of discussion. In the upper left

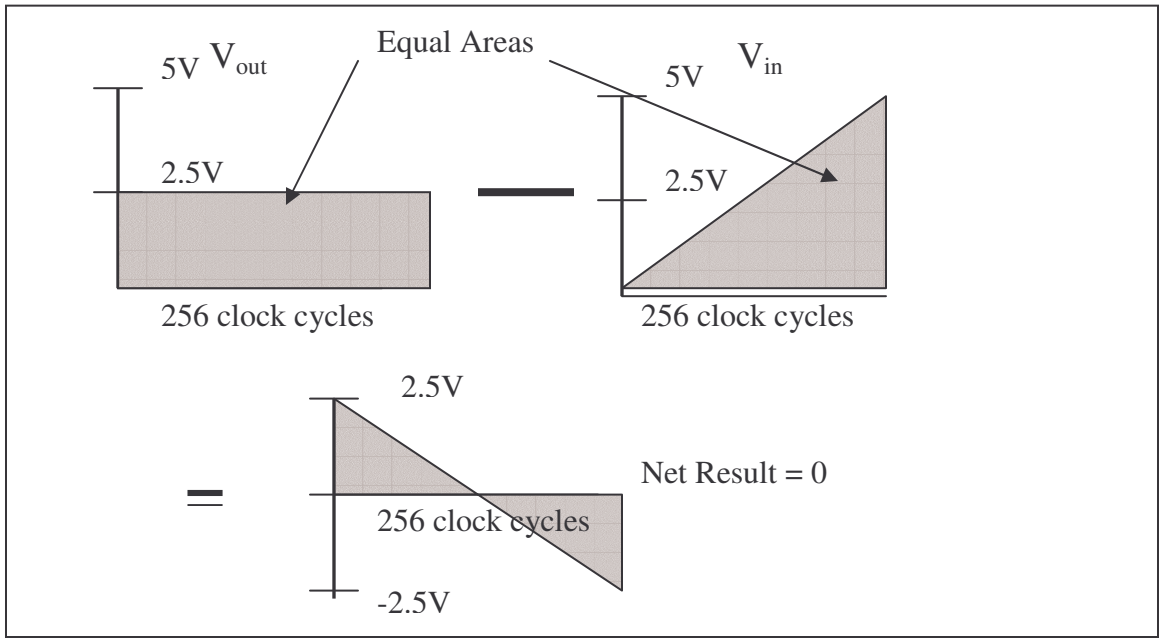


Figure 3.9 Illustration showing area corresponding to the fault for 2.5-V output condition over 256 clock cycles for computing S_{del} metric. The top left plot is the output waveform, the top right plot is the input waveform, and the bottom waveform is the resultant subtraction of the two upper plots. The lower plot with equal areas above and below the time axis have a net result of zero.

of Fig. 3.10 V_{out} is represented as a 2.5-VDC constant for 256 clock cycles. In the upper right, V_{in} is represented as a ramp from 0 to 5 volts for those same 256 clock cycles. In the lower left is the difference of the upper two figures representing $V_{out} - V_{in}$ for same 256 clock cycles. In the lower right is the magnitude of the difference of the upper two figures representing $|V_{out} - V_{in}|$ for same 256 clock cycles.

In this case, S_{mag} becomes:

$$S_{mag} = \sum_{n=0}^{N-1} \left| V_{out}[n] - V_{in}[n] \right| = \sum_{n=0}^{255} \left| 2.5 - \frac{5n}{255} \right| = 320 .$$

For more complex signals, the summation can be viewed as an integral. For the present example, the S_{mag} metric is analogous to the area of the two triangles in the lower right of Fig. 3.10. Given that the result is for one saw-tooth cycle only, S_{mag}

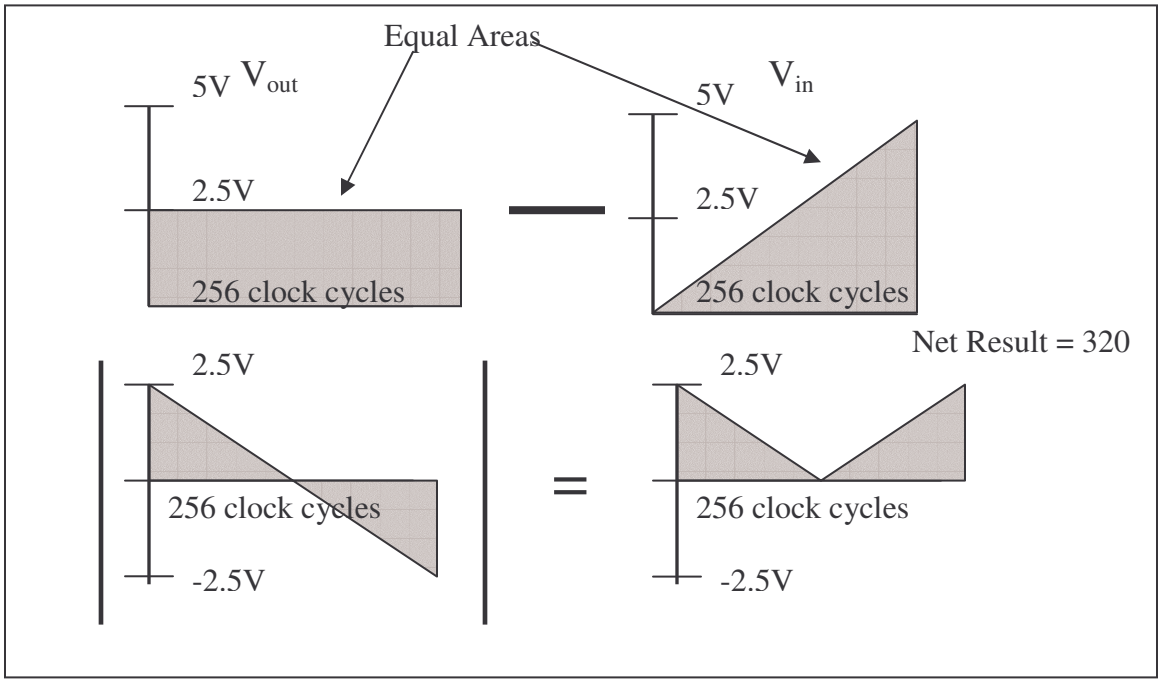


Figure 3.10 Illustration showing area summed for 2.5-V output condition over 256 clock cycles for S_{mag} metric. Top left is output waveform, top right is input waveform, bottom left is result of subtraction of output from input, and bottom right is magnitude of bottom left.

would equal the area of the two triangles, with base=128 clock cycles and height=2.5 volts for each triangle, with area 128x2.5/2=160 for a total area of 320 in both triangles.

3.3.1.4 Summary of Floating Point Calculations

Table 3.3 summarizes the theoretical predicted values for the three floating point ORA metrics S_{out}, S_{del}, and S_{mag} for the case of Table 3.2 where the output is stuck at 2.5V.

Table 3.3
The theoretical floating point values for ORA metrics S_{out}, S_{del}, and S_{mag} for the fault in Table 3.2 with 2.5-VDC output with saw-tooth input over one cycle

ORA metric	Expected Floating Point Value
S_{out}	640
S_{del}	0
S_{mag}	320

3.3.2 Digital ORA Metrics

Whereas the floating point ORA metrics of the previous sections are useful for investigation, the digital ORA metrics S_{16out}, S_{16del}, and S_{16mag} are used for direct comparison between the fault simulator and the experimental hardware. The BIST system under consideration has an ADC and a DAC of 8-bits, and accumulator of 16-bits, which performs the summation of Table 2.2. The ADC and the DAC analog voltage ranges are 0 to 5 volts, with 00 hex corresponding to 0 V and FF hex corresponding to 5V.

The test waveform of one cycle of cup was selected because it would not overflow the accumulator regardless of the output voltage. In the worst case, one would have 256 clock cycles of FF hexadecimal ADC output added in the accumulator. This would result in a final accumulator value of FFFF in hexadecimal. For the case of the mid rail voltage of 2.5 V(as in the faults of Table 3.2), or a 7F in hexadecimal, times 256 clock cycles gives 7F00, or 32767 in decimal, which is half of the maximum value of the accumulator. Therefore, the accumulator will not overflow for the cases under consideration.

3.3.2.1 S_{16out} Digital Value

The cases of Table 3.2 with an output voltage stuck at 2.5 V are used to calculate S_{16out} since it provides simple hand calculations. The S_{16out} metric adds the digital output voltage at each clock cycle and sums the result over the number, N, of specified clock cycles. Only one cycle of the TPG waveform (256 clock cycles) is used for the simulation under discussion. In Fig. 3.8, the 2.5-VDC output V_{out} would correspond to an ADC digitized value of 7F hexadecimal, or 127 decimal. In this case, S_{16out} becomes:

$$S_{16out} = \left(\left(\sum_{n=0}^{N-1} ((V_{out}[n]))_{256} \right) \right)_{65536} = \left(\left(\sum_{n=0}^{255} ((127))_{256} \right) \right)_{65536} = 32512 ,$$

where 32512 decimal is 7F00 hexadecimal. This can also be confirmed by summing 7F hexadecimal 256 times decimal to get 7F00 hexadecimal, or 32512 decimal.

3.3.2.2 S_{16del} Digital Value

As in the calculation of S_{16out}, the cases of Table 3.2 with an output voltage stuck at 2.5 V are used to calculate S_{16del} since it provides simple hand calculations. The S_{16del} metric subtracts the digitized voltage of the input from the digitized high pass output voltage at each clock cycle and sums the result over the number, N, of specified clock cycles. Only one cycle of the Cup TPG waveform (256 clock cycles) is used for the simulation under discussion. In the upper left of Fig. 3.9, V_{out} is represented as a 2.5-VDC constant, corresponding to digitized value of 7F hex, for 256 clock cycles. In the upper right of Fig. 3.9, V_{in} is represented as a ramp as 5 volt for those same 256 clock cycles, corresponding to a ramp from 00 hexadecimal to FF hexadecimal after digitization. The subtraction is a 1's compliment subtraction with V_{out} and V_{in} unsigned 8-bit and with the output 1's compliment signed 16-bit. In this case S_{16del} becomes:

$$S_{16del} = \left(\left(\sum_{n=0}^{N-1} ((V_{out}[n] - V_{in}[n]))_{65536} \right) \right)_{65536} = \left(\left(\sum_{n=0}^{255} ((127 - n))_{65536} \right) \right)_{65536} = 0 .$$

The digital result of 0 for S_{16del} corresponds to the analog answer of 0 for S_{del}. The digital method of subtraction varies from the floating point in that the subtraction is done using the ones complement and the input values V_{out} and V_{in} are unsigned.

3.3.2.3 S_{16mag} Digital Values

The cases of Table 3.2 with an output voltage stuck at 2.5 V are used to calculate S_{16mag} since it provides simple hand calculations. The S_{16mag} metric at each clock cycle subtracts the digitized voltage of the input from the digitized high pass output voltage, and then takes the absolute value and sums the result over the number, N, of specified clock cycles. Only one cycle of the TPG waveform (256 clock cycles) is used for the simulation of discussion. The digitization of the corresponding signals in Fig. 3.10 follows in the manner as digitization of Fig. 3.9 described for S_{16del} in the prior section. Again, the subtraction is a 1's compliment subtraction with V_{out} and V_{in} unsigned 8-bit and with the output 16-bit 1's compliment signed. In this case, S_{16mag} becomes:

$$S_{16mag} = \left(\left(\sum_{n=0}^{N-1} ((|V_{out}[n] - V_{in}[n]|))_{65536} \right) \right)_{65536} = \left(\left(\sum_{n=0}^{255} ((|127 - n|))_{65536} \right) \right)_{65536} = 16256 .$$

The digital result of 16256 for S_{16mag} corresponds to the analog answer of 320 for S_{mag}. The digital method of subtraction varies from the floating point in that the subtraction is done using 16-bit 1's complement, the input values of V_{out} and V_{in} are 8-bit unsigned and the output is 16-bit signed. The value 16256 for S_{16mag} should be half of the S_{16out} value of 32512 as evident by comparing Figs. 3.9 and 3.10.

3.3.2.4 Summary of Digital Calculations

Table 3.4 summarizes the theoretical predicted values for the three digital ORA metrics S_{16out}, S_{16del}, and S_{16mag} for the case of Table 3.2 where the output is stuck at 2.5V.

Table 3.4
The theoretical digital values for ORA metrics S_{16out}, S_{16del}, and S_{16mag} for the fault in Table 3.2 with 2.5-VDC output with saw-tooth input over one cycle

ORA Metric	Expected Digital Value (decimal)
S_{16out}	32512
S_{16del}	0
S_{16mag}	16256

3.4 Simulation Data for BiQuad Filter Circuit

The BiQuad circuit of Fig. 3.1 was simulated using the reduced order model of Fig. 3.3 (replacing the operational amplifiers with the reduced order models of Fig. 3.4) for comparison against the theoretical values from section 3.3. In section 3.5, simulation results are compared to hardware experimental results.

In the simulations, the BiQuad filter circuit of Fig. 3.3 was simulated with count-up(Cup) ramp waveform with no initialization cycles and one repetition of the TPG pattern (256 clock cycles). The simulation was run for 160 randomizations per fault and for the fault-free circuits. The fault simulation was executed with the following command line:

faultsim biquad.cir 8 160 11 0 2 0 2.5 5 60 0.5 4.5 1

In this command line, biquad.cir is the circuit SPICE file for the reduced order model version of the circuit of Fig. 3.3. The second field, numproc=8, is the number of parallel processes that are forked to run in parallel on a multi-cpu machine. The third field, numrand=160, is number of randomizations due to parametric variations of normal components that are done per fault. In the fourth and fifth fields, inpos=11 and inneg=0, are positive and negative differential input nodes in the SPICE file. In the sixth and seventh fields, outpos=2 and outneg=0 are positive and negative differential output nodes for the SPICE file. The eighth field, vbias=2.5 is the TPG waveform DC bias, where

inpos=vbias+(vamp/2), inneg=vbias-(vamp/2) and if vbias=0, the input is true floating differential input. The ninth field, vamp=5, is the test pattern amplitude in volts. The tenth field, maxcpu=60, is maximum number cpu seconds allowed per SPICE run. The eleventh and twelfth fields, vomin=.5 and vomax=4.5, is the differential output voltage range where 00 hexadecimal corresponds to vomin and FF hexadecimal corresponds to vomax for ADC converter of Fig. 2.1. The last field, numrep=1, is the number or repetitions to execute TPG waveform (1 repetition= 256 clock cycles). By default, three frequencies are simulated and the hold-off for collecting ORA data is 0. In addition, waveforms are hard-coded in faultsim at this present version of the software. The waveforms and frequencies must be selected before recompiling the faultsim executable.

After completion of fault simulation, the raw ORA data is contained in the ORA files. The ORA files are then processed in a post-processing executable named anarun. The post-processing takes the ORA files and generates excel spreadsheets for histograms of the six ORA metrics for each TPG waveform. The post-processing program also calculates the mean, variance for each fault for the six analog and digital ORA metrics S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, and S_{16mag}.

3.4.1 Analog Results for Fault Simulator

In this section, simulation results for the analog metrics, S_{out}, S_{del}, and S_{mag}, are compared against the theoretical results. The histogram for the S_{out} analog output metric is shown in Fig. 3.11. In Fig. 3.11, the histogram for a fault-free unit is shown as a solid line and the faulty circuit histogram is shown as a dotted line. The faulty circuit histogram is a composite of all the faults formed by summing all the faulty histograms and dividing by the number of faulty histograms. (Histograms showing good circuits and

a single fault, for all faults listed in Table 3.1, can be found Appendix F.) The vertical axis gives the number of units falling within a certain ORA metric range bin and the horizontal axis is the ORA metric value, i.e., S_{out}. This convention will be used for all histograms contained hereafter. The figure shows faulty circuits clustered around S_{out}=640 mark. These faulty circuits correspond to the three resistive shorts given in Table 3.3 that produce the mid-rail DC voltage of 2.5 volts.

The histogram of Fig. 3.11 also shows a cluster of faults that fall in the same range as the good circuits. Because faults in many components affect the output of the count-up waveform only slightly, these other faulty circuits have S_{out} values that are clustered around the fault-free circuit histograms. These faults, which affect the count-up output only slightly, are listed in Table 3.5. Although these faults are not detectable with the present waveform, previous work on the BiQuad filter has shown that these faults can be detected with other waveforms [28].

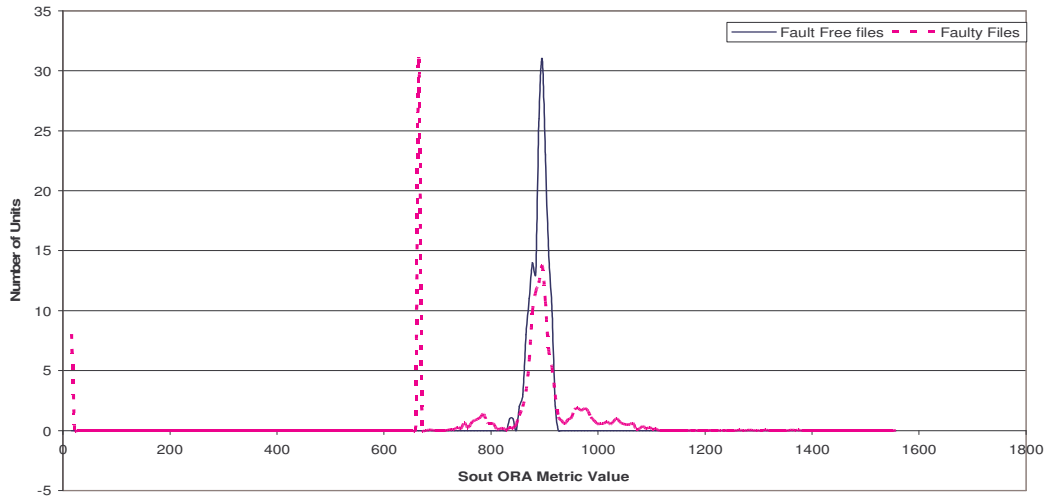


Figure 3.11 Fault simulator results for analog S_{out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective waveform frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram which is the histogram for fault-free circuits.

For the cases of Table 3.2 that were used to calculate theoretical results, Table 3.6 shows percent error between theoretical and simulated results for analog S_{out}. In Table 3.6, the first column lists the fault, the second column gives the theoretical value of S_{out}, and the third column gives the mean value of the simulated S_{out} for 160 randomized circuits for that particular fault (160 randomizations set by the faultsim command line). In Table 3.6, the percent error is calculated by dividing the difference between the mean, μ, in column two and the theoretical value, t, in column one and dividing the theoretical value, t, in column one, and finally multiplying by 100 to obtain the percentage. This calculation is shown in equation 3.3 below.

$$e = \frac{\mu - t}{t} \times 100 \qquad (3.3)$$

Table 3.6 shows, for the analog S_{out} metric, that the fault simulator results are within 5 percent of the theoretical results.

Table 3.5
Faults with only slight effect on the output of BiQuad filter for count-up waveform at 19.5 kHz

Component	Fault
R3	Open
R4	Open
R4	Short
R5	Open
R5	Short
R6	Open
R6	Short
R7	Open
R7	Short
C1	Open

Table 3.6
S_{out} comparison of analog theoretical values against fault simulator results

Fault	Theoretical Value of S_{out}	Fault Simulator mean value	% error(eq 3.3)
R2 short	640	665.629	4.1
R3 short	640	665.658	4.1
R1b short	640	665.786	4.1

The histogram for the S_{del} analog output metric is shown in Fig. 3.12. In Fig. 3.12, the fault-free circuit histogram is shown as solid line and the composite faulty circuit histogram is shown as a dotted line. The faulty circuit histogram is a composite of all the faults formed by summing all the faulty histograms and dividing by the number of faults (See Appendix F for histograms of single faults and fault-free circuit for each fault). The vertical axis gives the number of units within a certain ORA metric range bin and the horizontal axis is the ORA metric value, S_{del}. The figure shows a cluster of faulty circuits cluster around S_{del}=0. These faulty circuits correspond to the three resistive shorts given in Table 3.3 that produce the mid-rail DC voltage of 2.5 volts.

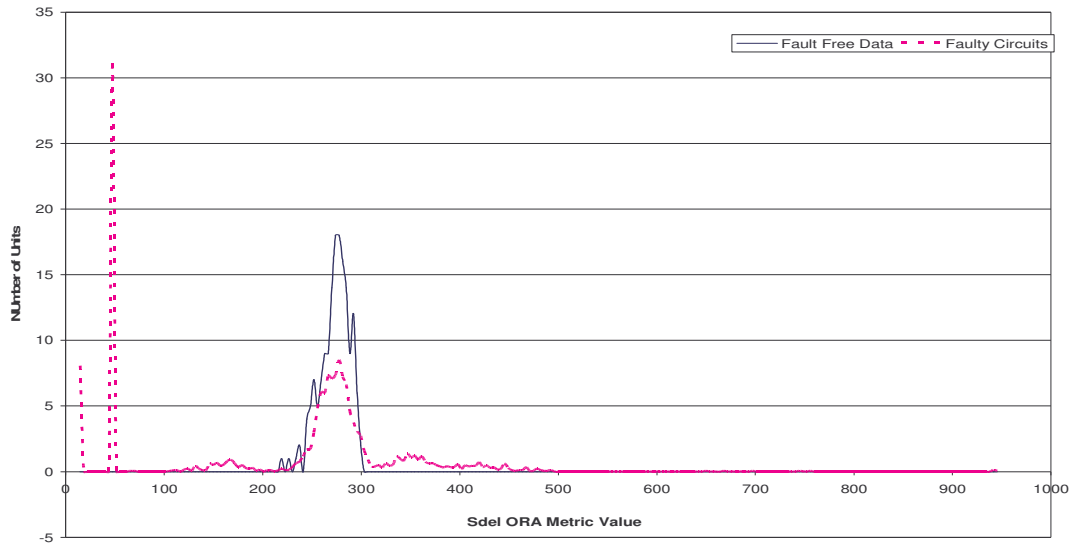


Figure 3.12 Fault simulator results for analog S_{del} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective waveform frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram, which is the histogram for fault-free circuits.

Table 3.7 shows the error for the theoretical values for the S_{del} metric. The percent error calculation in equation 3.3 is not appropriate for Table 3.7 since the theoretical value is 0. Therefore, the error is simply defined as the difference.

Table 3.7
S_{del} comparison of analog theoretical values against fault simulator results

Fault	Theoretical value of S_{del}	Fault Simulator mean value	Error
R2short	0	47.97	47.97
R3short	0	48.00	48.00
R1bshort	0	48.12	48.12

The histograms in Fig. 3.12 also show the overlap of faults listed in Table 3.5 with the fault-free histogram, corresponding to undetectable faults for this waveform.

Finally, the histogram for the S_{mag} analog output metric is shown in Fig. 3.13. In Fig. 3.13, the fault-free circuit histogram is shown as solid line and the composite faulty circuit histogram is shown as a dotted line. The vertical axis gives the number of units within a certain ORA metric range bin and the horizontal axis is the ORA metric value, S_{mag}. The figure shows a cluster of faulty circuits around S_{mag}=320. Unlike S_{out} and S_{del}, it is not possible to resolve a separate cluster corresponding only to the three faults given in Table 3.3 that produce the mid-rail DC voltage of 2.5 V.

Although the histogram does not show a separate cluster corresponding to the theoretical results for the faults of Table 3.3, comparisons between theoretical and simulated values may yet be made. Table 3.8 shows that the simulated values for the S_{mag} ORA metric for analog data have approximately 1 percent error from the theoretical value. The percent error calculation in Table 3.8 uses equation 3.3, with corresponding μ, and t.

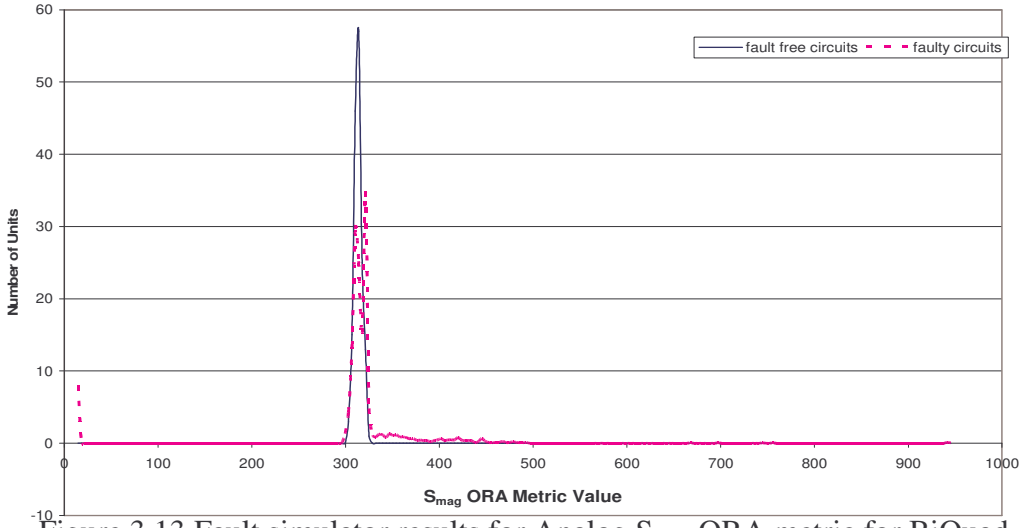


Figure 3.13 Fault simulator results for Analog S_{mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective waveform frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram, which is the histogram for fault-free circuits.

Table 3.8
S_{mag} analog theoretical values compared with fault simulator results

Fault	Theoretical Value of S_{mag}	Fault Simulator mean value	% error(eq3.3)
R2short	320	322.65	.80
R3short	320	322.85	.80
R1bshort	320	322.81	.80

3.4.2 Digital Results for Fault Simulator

In this section, simulation results for the digital ORA metrics S_{16out}, S_{16del}, S_{16mag} are compared against theoretical results. The histogram for the S_{16out} digital output metric is shown in Fig. 3.14. In Fig. 3.14, the fault-free circuit histogram is shown as solid line and the faulty circuit histogram is shown as a dotted line. The faulty circuit histogram is a composite of all the faults formed by summing all the faulty histograms and dividing by the number of faults. The vertical axis gives the number of units within a certain ORA metric range bin and the horizontal axis is the decimal ORA metric value,

S_{16out}. This convention will be used for all histograms contained hereafter. The figure shows a cluster of faulty circuits around S_{16out}=34000. These faulty circuits correspond to the three resistive shorts given in Table 3.3 that produce the mid-rail DC voltage of 2.5 volts. The histogram also shows a cluster of faults that fall in the same range as the fault-free circuits. Because faults in many components affect the output of the count-up waveform only slightly, many faulty circuit S_{16out} value fall within the cluster of the fault-

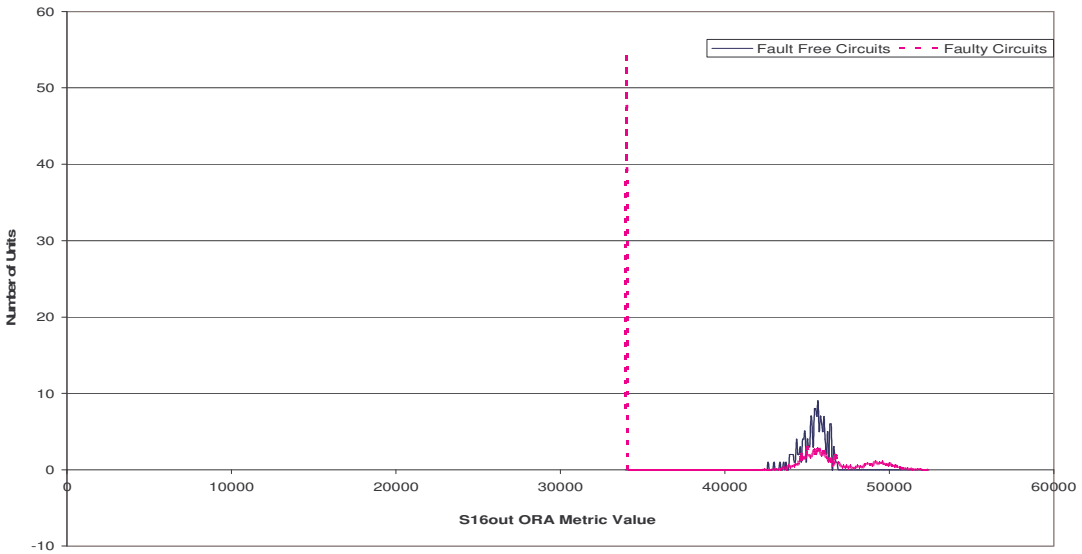


Figure 3.14 Fault simulator results for analog S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram, which is the histogram for fault-free circuits.

free circuit histograms. These faults, which affect the count-up output only slightly, are listed in Table 3.5.

For the cases of Table 3.2 that were used to calculate theoretical results, Table 3.9 shows percent error between theoretical and simulated results for S_{16out}. In Table 3.9, the

first column lists the fault, the second column gives the theoretical value of S_{16out} and the third column gives the mean of the simulated value of S_{16out} for 160 randomized circuits, for that particular fault (160 randomizations set by the faultsim command line). In Table 3.9, the percent error is calculated by using equation 3.3. Table 3.9 shows that the simulated value of S_{16out} metric is less than 5 percent form the theoretical values.

Table 3.9
S_{16out} digital theoretical values against fault simulator results.

Fault	Theoretical Value of S_{16out}	Fault Simulator mean value	% error(eq 3.3)
R2short	32512	34048	4.72
R3short	32512	34048	4.72
R1bshort	32512	34048	4.72

The histogram for the S_{16del} digital output metric is shown in Fig. 3.15. In Fig. 3.15, the fault-free circuit histogram is shown as solid line and the composite faulty circuit histogram is shown as a dotted line. The faulty circuit histogram is a composite of all the faults formed by summing all the faulty histograms and dividing by the number of faulty histograms. The vertical axis gives the number of units within a certain ORA metric range bin and the horizontal axis is the ORA metric value, S_{16del}. The histogram of Fig. 3.15 shows a cluster of faulty circuits around S_{16del}=2500. These faulty circuits correspond to the three resistive shorts given in Table 3.3 that produce the mid-rail DC voltage of 2.5 volts.

Table 3.10 shows the error relative to the theoretical values for the simulated S_{16del} metric. The percent error calculation in equation 3.3 is not appropriate for Table 3.10 since the theoretical value is 0. Therefore, the error is defined as the difference. The two histograms in Fig. 3.15 also show overlap of faults listed in Table 3.5, again indicating undetectable faults for this waveform.

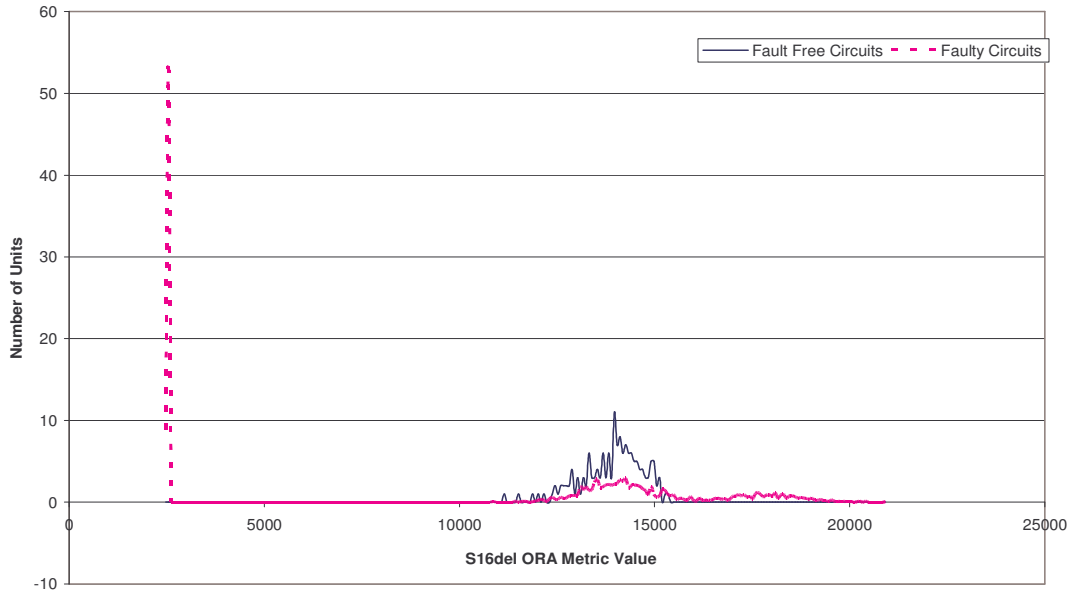


Figure 3.15 Fault simulator results for analog S_{16del} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram, which is the histogram for fault-free circuits.

Table 3.10
S_{16del} digital theoretical values compared with fault simulator results

Fault	Theoretical Value of S_{16del}	Fault Simulator mean value	error
R2short	0	2532	2532
R3short	0	2532	2532
R1bshort	0	2532	2532

The histogram for the S_{16mag} digital output metric is shown in Fig. 3.16. In Fig. 3.16, the fault-free histogram is shown as solid line and the composite faulty circuit histogram is shown as a dotted line. The faulty circuit histogram is a composite of all the faults formed by summing all the faulty histograms and dividing by the number of faulty histograms. The vertical axis gives the number of units with a certain ORA metric range bin and the horizontal axis is the ORA metric value, S_{16mag}.

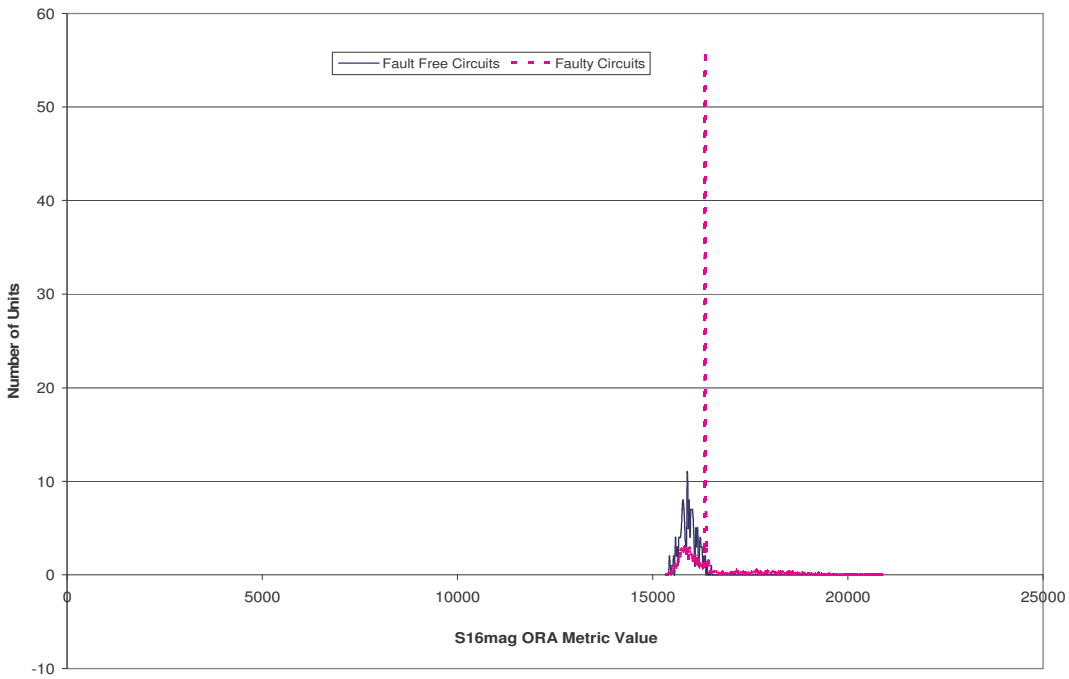


Figure 3.16 Fault simulator results for analog S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The dotted histogram is a composite of all faults rescaled and normalized relative to the solid histogram, which is the histogram for fault-free circuits.

The histogram of Fig. 3.16 shows a cluster of faulty circuits around S_{16mag}=16346. Unlike S_{16out} and S_{16del}, it is not possible to resolve a separate cluster corresponding only to the three resistive shorts given in Table 3.3 that produce the mid-rail DC voltage of 2.5 V, since the values of faulty circuits are so close to the values of fault-free circuits.

Table 3.11 shows the error for the S_{16mag} metric of the fault simulator is approximately 1 percent relative to the theoretical value. The percent error calculation in Table 3.11 uses equation 3.3, with corresponding μ, and t.

In Figs. 3.11 through 3.16, it can be seen that the fault-free circuit histograms and the faulty circuit histograms are often times very close to each other for each of the six analog and ORA metrics. The histograms of Figs. 3.11 through 3.16 show that with the

count-up 19.5 kHz, 5 V peak-to-peak TPG waveform many faults are undetectable for this the ORA metric and TPG waveform. The faults listed in Table 3.5 are undetectable with the TPG waveform. However, they may be detectable with other waveforms and are therefore classified as potentially detectable.

Table 3.11
S_{16mag} digital theoretical values against fault simulator results

Fault	Theoretical Value	Fault Simulator mean value	% error(Eq.3.4)
R2short	16256	16346	0.5
R3short	16256	16346	0.5
R1bshort	16256	16346	0.5

3.5 Comparison with Experimental Hardware

In addition to confirming the results of the fault simulator against theoretical results in the previous section, this section compares fault simulator results to experimental hardware results. The experimental data was produced by Jason Morton in the VLSI-FPGA Design and Test Lab under the direction of Dr. Charles Stroud of the University of North Carolina at Charlotte. This hardware experimental data was generated using the BiQuad filter (detailed schematic included in Figs. 3.1 and 3.3). The VLSI-FPGA Design and Test Lab used the BIST system of Fig. 2.1, designed and built at the VLSI-FPGA Design and Test Lab, was implemented per Fig. 3.18. Faults are injected into the circuit of Fig. 3.3 by switches in series for opens, and by short-circuit jumpers in parallel for shorts, for the faults listed in Table 3.1. The hardware does not produce the ORA metric for the S_{16del}, so the comparison of the hardware to the fault simulator will be limited to the S_{16out} and S_{16mag} ORA metrics.

Table 3.12 is a summary of the results for the ORA metric S_{16out} for the fault simulator and the hardware. Table 3.12 shows the percent error, as defined in equation

3.4 below, between the results for the fault simulator and the experimental hardware results collected by Jason Morton from the VLSI-FPGA and Test Lab working under the direction of Dr. Charles Stroud.

$$e = \frac{\mu_{hardware} - \mu_{simulator}}{\mu_{simulator}} \times 100 \qquad (3.4)$$

Table 3.12
Comparison of experimental hardware and simulations for S_{16out} ORA metric showing difference between means and percent difference

S16out, 5 Megahertz clock, Count-up waveform

Faults	Hardware Results			Software Results			Comparison	
Fault	μ	σ	variance	μ	σ	variance	μ difference	%error of
nofaults	42275	16.97	287.981	45420	783.96	614593	-3145	-6.92
R2short	35512	0	0	34048	0.3405	0.1159	1464	4.30
R3open	60830	2.07	4.2849	49726	856.7	733935	11104	22.33
R3short	29602	97	9409	34048	0.3405	0.1159	-4446	-13.06
R4open	43203	6.39	40.8321	45451	775	600625	-2248	-4.95
R4short	35000	4991.7	2.5E+07	33974	13	169	1026	3.02
R5open	43166	31.3	979.69	46654	193	37249	-3488	-7.48
R5short	32841.9	5881.9	3.5E+07	34048	0.3405	0.1159	-1206.1	-3.54
R6open	41385	10.5	110.25	45289	722	521284	-3904	-8.62
R6short	29818	5370	2.9E+07	49001	800	640000	-19183	-39.15
R7open	29600	5039	2.5E+07	49492	762	580644	-19892	-40.19
R7short	41375	11.3	127.69	45325	13.95	194.6	-3950	-8.71
C1open	32847	49	2401	34048	0.3405	0.1159	-1201	-3.53
C2open	34350	4610	2.1E+07	34048	0.3405	0.1159	302	0.89

Table 3.12 shows error ranging from less than 1 percent to 40 percent. The results for the fault simulator tend to agree with the hardware experimental results for most faults, with only a few faults having considerable differences.

Table 3.13 contains the same data as Table 3.12 except for the S_{16mag} ORA metric. The error in this Table can be seen to vary over a wider range than Table 3.12. The range of error for the ORA metric S_{16mag} ranges from a half percent to 75 percent.

Table 3.13
Comparison of experimental hardware and simulations for S_{16mag} ORA metric showing difference between means and percent difference
S16mag, 5 Megahertz clk, Count-up waveform

Faults	Hardware Results			Fault Simulator Results			Comparison	
	μ	σ	variance	μ	σ	variance	μ difference	%error of
nofaults	13191	24.1	580.81	15913	213.84	45727.5	-2722	-17.11
R2short	16257	0	0	16346	0.1634	0.0267	-89	-0.54
R3open	25410.7	3705.83	13733176	18212	851	724201	7198.71	39.53
R3short	16998.2	132.93	17670.38	13346	0.1634	0.0267	3652.23	27.37
R4open	13485.5	5.88	34.5744	15893	241	58081	-2407.5	-15.15
R4short	28059	753.32	567491	16370	139.23	19385	11689.01	71.41
R5open	13233	20.7	428.49	16177	353.84	125203	-2943.99	-18.20
R5short	28919.4	12.097	146.3374	16346	0.1634	0.0267	12573.42	76.92
R6open	14203.3	7.79	60.6841	15871	213	45369	-1667.66	-10.51
R6short	13191.8	34.57	1195.085	17530.9	740	547600	-4339.15	-24.75
R7open	11580.6	1158	1340964	17987	828	685584	-6406.42	-35.62
R7short	14200.8	5.84	34.1056	15883	218	47524	-1682.19	-10.59
C1open	18465	96	9216	15908	221	48841	2557	16.07
C2open	18685	78	6084	16346	0.163	0.02657	2339	14.31

3.6 Good Circuit Result Confirmation

In addition to confirming the results of the fault simulator against theoretical results for the specific fault conditions of Table 3.2, comparisons were made for a fault-free circuit. This section compares the fault-free circuit ORA data, comparing simulation results for the fault-free circuit to hardware results for the fault-free circuit.

A high-pass filter, such as the BiQuad filter should by definition have no DC offset in the output. For the BiQuad circuit of Fig. 3.1, the high pass output should be symmetrical about the 2.5-VDC virtual ground of the output. In this event, S_{16out} and S_{out} ORA metrics would both be expected to have ORA values for good circuits that would be effectively the same as a 2.5-Vconstant DC output.

Transient effects that skew ORA metric results from their expected values are present in the hardware and simulation. Fig. 3.17 shows an oscilloscope trace capturing

the first cycles of the cup TPG waveform at 5 MHz clock frequency with 5 V peak-to-peak and 2.5 V offset in the hardware system. The upper trace of Fig. 3.17 is the input to the hardware BiQuad circuit, after inversion of the DAC output by the inverting amplifier of Fig. 3.18. The lower trace is the high pass output of the hardware BiQuad circuit.

The oscilloscope trace of Fig. 3.17 was taken by the VLSI-FPGA Design and Test Lab by Steve Tucker and Jason Morton. The behavior of the output in the hardware circuit shown in Fig. 3.17 matches roughly the SPICE simulation of Fig. 3.6. The oscilloscope's lower trace shows the high pass BiQuad output starting higher than 2.5 VDC, going below 2.5 VDC, and then settling out to the expected 2.5 V virtual ground. Therefore, this transient is present in both the hardware and simulation, and should also influence the ORA results for both the fault simulator and the hardware experiments.

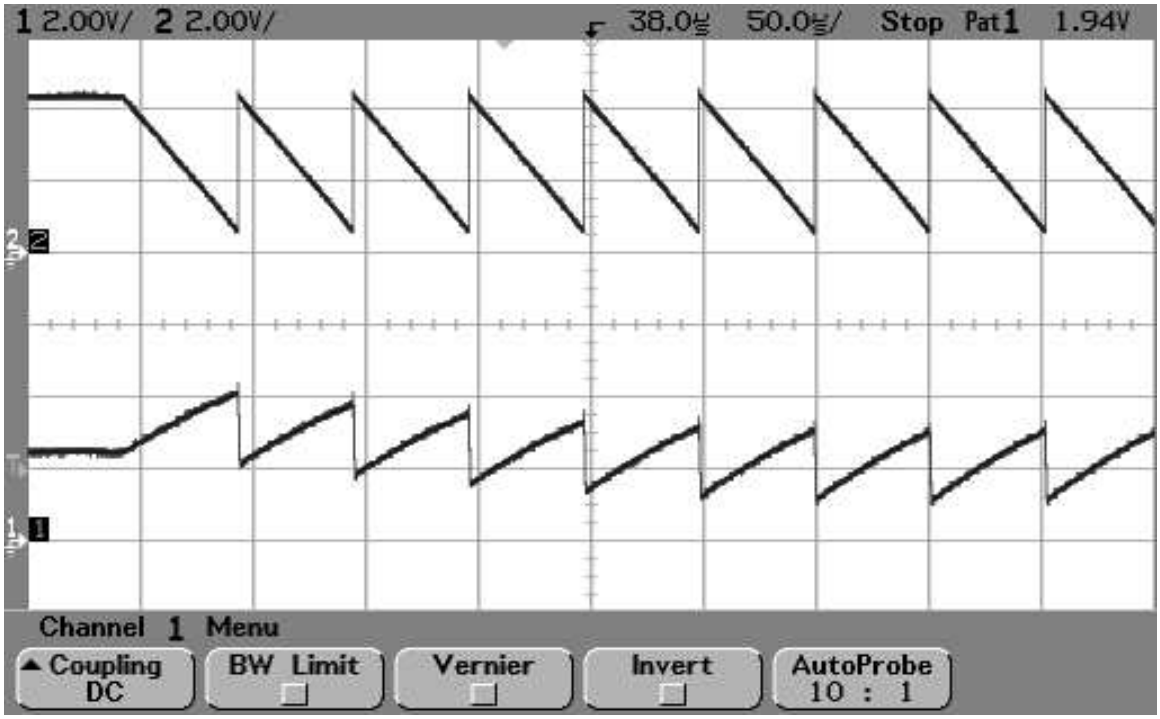
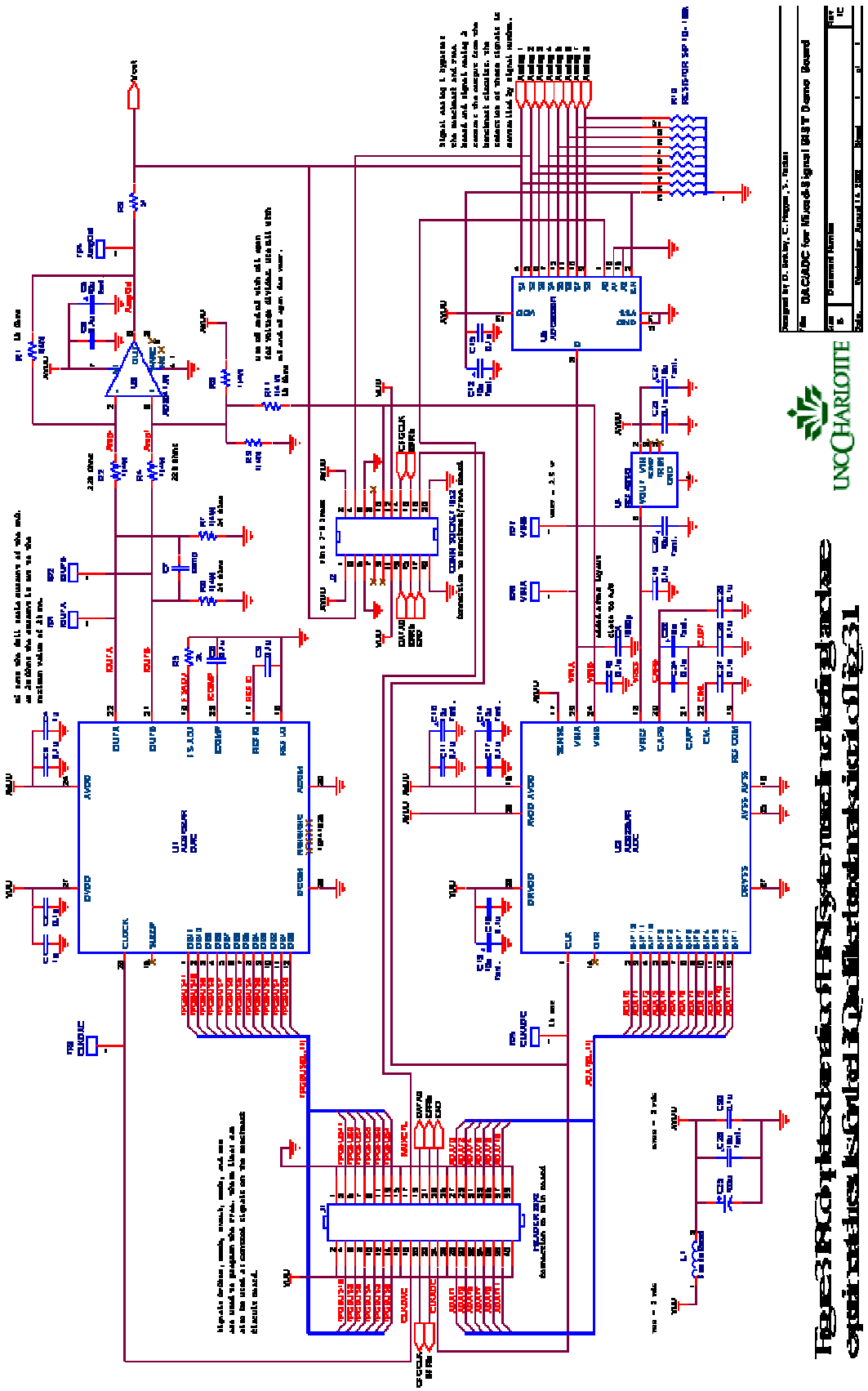


Figure 3.17 Oscilloscope plot showing presence of transient effect on 5MHz Cup waveform with 5Vpp input on BiQuad filter (compare to SPICE plot Fig. 3.6). Upper trace is 0 to 5 V Cup input TPG waveform (after inverting amp of Fig. 3.5), lower trace is high pass output showing transient behavior within first 4 or 5 cycles of saw-tooth waveform.

As is evident by inspection of the histograms of Figs. 3.11 and 3.14, the fault-free data is clustered around a value that is shifted up form the corresponding theoretical values of S_{out}=640 and S_{16out}=32512 decimal. From Fig. 3.17, the high pass output in the lower trace is initially shifted up by approximately 1 volt, corresponding to a shift in S_{out} of 256×1=256 and corresponding to a shift of S_{16out} of 33 hex times FF hex equals 32CD hex or 13005 decimal. And so, this would result in shifted histograms for a S_{out} and S_{16out} centered at 640+256=856 and 32512+13005=45517. And so, the fault-free S_{out} histogram of Fig. 3.11 is centered near 870, and the fault-free histogram for Fig. 3.14 is centered around 46,000.

3.7 Conclusion

In conclusion, the fault simulator has been validated against both theoretical and experimental hardware results for a simple waveform. For the simple faults which permit theoretical analysis, the simulator data was close to the theoretical data for $S_{out.}$ In addition, S_{16out} ORA data of Table 3.12 shows good agreement between the simulator results and the experimental hardware results for many faults. It is not clear, at present, the cause of the discrepancies between the simulator results and the experimental hardware results for S_{16mag} in Table 3.13. Nevertheless, there is good agreement for several faults between the simulator results and the experimental hardware results for S_{16mag} in Table 3.3.



CHAPTER 4: POTENTIAL FUTURE DIRECTIONS

One potential area of future investigation includes reducing the margin of error between the fault simulator and the experimental hardware results of Tables 3.12 and 3.13. In addition, more waveforms from Appendix B could be simulated and checked.

Other potential areas of future work include methods of speeding up the selection of TPG waveforms and improving the methods by which TPG waveforms are selected. Section 4.1 suggests possible ways to evaluate TPG waveforms for mixed signal BIST Microsystems. Section 4.2 addresses a second area of future investigation, issues of fault coverage for the BiQuad. Then section 4.3 considers receiver operating characteristics as a tool in selecting metric thresholds. Finally section 4.4 considers Bhattacharyya methods for selecting the most promising TPG waveforms to speed up the simulator.

4.1 Speeding up Fault Simulation

The task of choosing the best TPG waveform and ORA metric with the maximum fault coverage requires the simulation of many different TPG waveforms on many versions of a circuit. To illustrate the number of combinations, consider a very simple circuit with 11 components to demonstrate the time requirements of such a simulation. A circuit with 11 components will have 22 different faults, assuming two hard faults per component. Also assume each circuit with and without faults is randomized 250 times to simulate normal component variations. In addition, let there be 14 TPG waveforms at three frequencies and three amplitudes. This produces $11\times(22+1)\times250\times14\times3\times3=725{,}000$

circuits to be simulated with only modest coverage of frequency and amplitude. The particular case of the OpAmp1 circuit of Fig. 4.1 takes 10 seconds to simulate and would then give a simulation time of 7,250,000 seconds, or 84 days. This large amount of time even for a

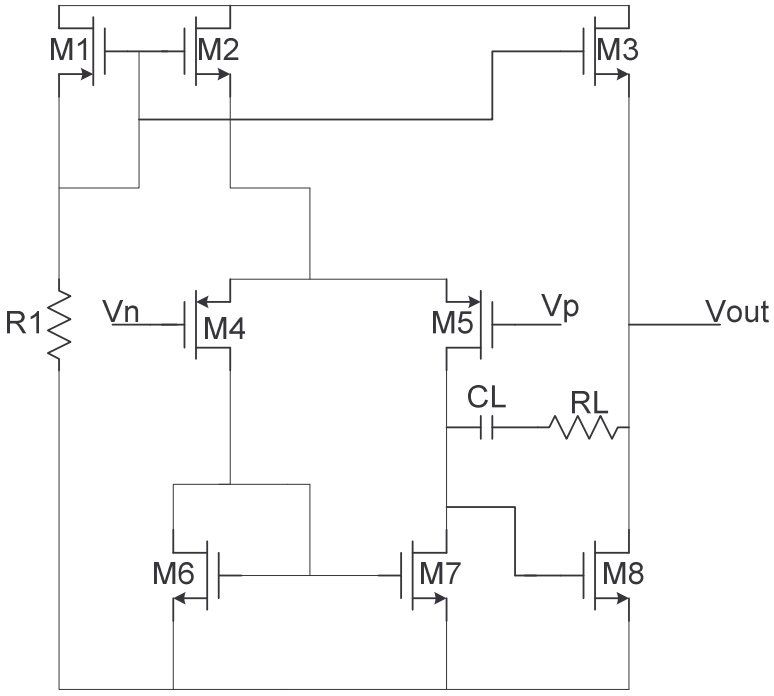


Figure 4.1 Operational amplifier circuit (OpAmp1).

very simple circuit of eleven components suggests a need for speeding up the fault simulator.

There are several possible approaches to speeding up the fault simulation ranging from changes in software architecture to sensitivity analysis. The possible changes in software architecture include altering the flow of the simulation module to work in multiple threads, a number of threads to be specified by the user based on the available

computing resources. Changes in the software to simulate 4 circuits simultaneously throughout the simulation cut the time by 75 percent, or linearly to one fourth. The reduction in simulation time t, from N threads, is N/t times faster.

The fault simulator was converted into a multi-threaded application by using the fork function in C++ to run SPICE simulations spawned by the fault simulator in parallel. This function will allow a program to diverge into multiple threads and allowing processes that don't depend on each other and don't fully consume computing resources, to run in parallel. The following is a code sample to show how a process can be forked. The fork() function call returns a 0 for a child process, positive integer for the parent or calling process and a negative integer in case of failure. This function allows for the program to split but, still have access to the same variables and functions of the parent program. The following code shows how simple controls can be used to spawn off child processes and increase the speed of an application.

```
 /*some statements before fork()*/
int pid = fork();
if( pid < 0 )
{
 perror("..."); exit(EXIT_FAILURE);
}
else if( pid ) //parent process
{
   /*some code here*/
}
else // child process
{
  /*child code here*/
}
```

In a second approach to speeding up simulations, more promising test vectors and ORA metrics can be simulated, pruning less promising TPG vectors and metrics from the search tree. Toward this end, a measure of how good a particular TPG waveform and

ORA metric is at isolating faults is proposed. This method will utilize Gaussian characteristics of the output data to prune ineffective TPG waveforms and ORA methods from the search tree.

4.2 Fault Coverage

A second area for future consideration is estimates of fault coverage for the BiQuad circuit. In earlier versions of the fault simulator, some fault coverage issues were investigated for the circuit of Fig. 4.1. The operational amplifier of Fig. 4.1 OpAmp1 was simulated in the fault simulator. (The net-list used in testing the operational amplifier can be found in Appendix C) The circuit was simulated using seven of the eleven TPG the test patterns found in Appendix B. All six of the aforementioned ORA metrics, S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, and S_{16mag} were evaluated. The OpAmp1 circuit was simulated with the following list of arguments as described in section 3.4:

faultsim benchmark.cir 4 160 17 18 16 0 2.5 0.2 120 0 5

A simulation of test patterns 1, 2, 3, 4, 5, 6, and 7 from Appendix B were used in the simulation at clock frequencies 10kHz, 100kHz, and 1MHz. All waveforms in the operational amplifier tests used two hundred milli-volt input amplitude. The amplitude was set to two hundred millivolts to be well in the range of the bias conditions such that the amplifier output would not clip too severely. It was observed that test patterns 8, 9 10, 11, and 12 had convergence problems with the circuit during simulation. Even after increasing simulation time limits, the convergence problems persisted and failed to yield useful ORA data. For purposes of the remainder of this section, only the 10 kHz cup TPG waveform and the S_{del} ORA metric will be considered.

Figure 4.2 is the analog histogram of S_{del} for the count-up (Cup) analog ramp function waveform at 10 kHz clock frequency and two hundred milli-volt amplitude. The 10 kHz clock frequency gives an effective frequency of 39 Hz, as derived by dividing the clock frequency by the number of clock cycles it takes to complete one cycle, 256. The histograms of Fig. 4.2 show the histogram created by the fault-free circuit as a solid line and the composite for all faults as a dotted line.

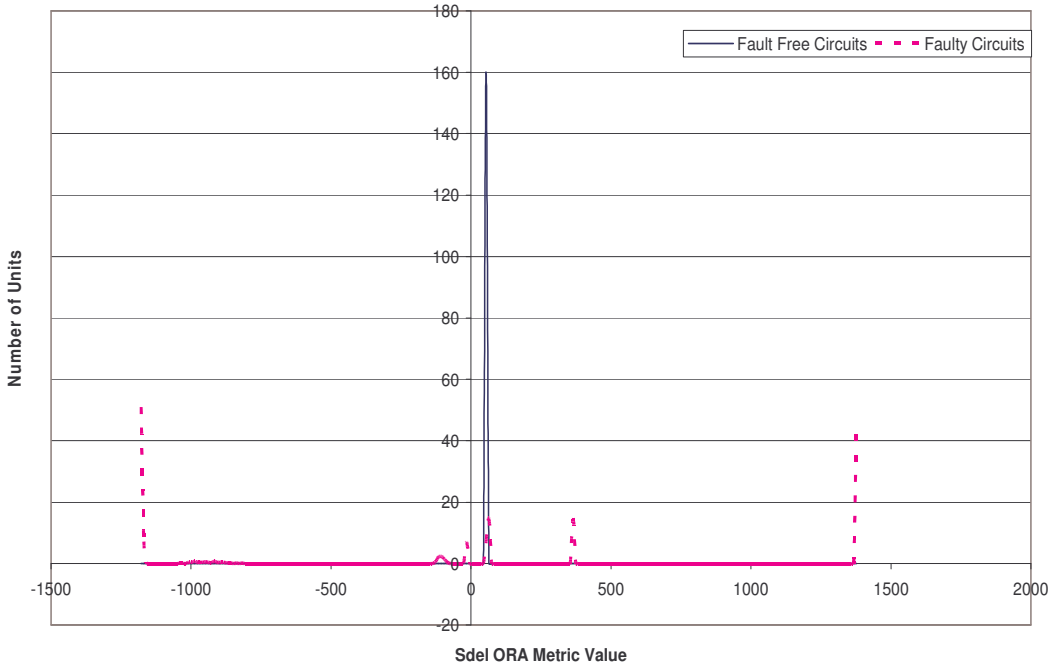


Figure 4.2 Histogram of fault-free circuits and faulty circuits for OpAmp1 with 200 mV Cup waveform at 10 kHz clock frequency.

Figure 4.3 shows a histogram containing only good circuits, the range at which a fault-free circuit would fall when tested with the cup waveform at 10 kHz, two hundred milli-volt amplitude, and S_{del} ORA metric. Figure 4.4 shows the histogram produced when an open is introduced into M1, creating a suck-off fault along with the histogram of fault-free circuit (solid).

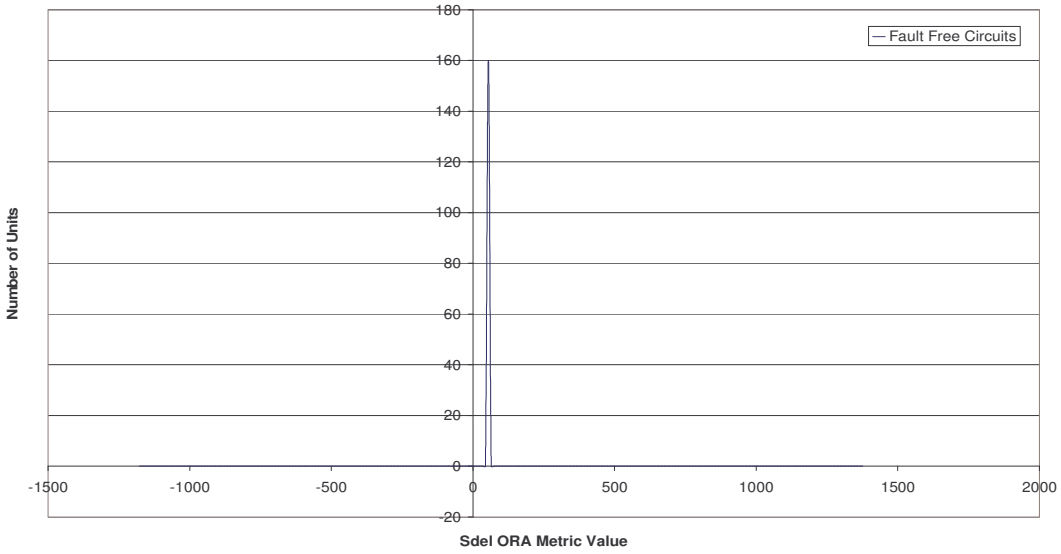


Figure 4.3 Histogram of fault-free circuits for OpAmp1 with 200 mV Cup waveform at 10 kHz clock frequency.

The stuck-off fault condition for the transistor M1 of Fig. 4.1 is modeled as an open as discussed in section 2.3.4, preventing the transistor from turning on and operating normally.

The M1 open creates the histogram in Fig 4.4 that is clearly shifted off to the right, distinguishing the faulty circuit from the fault-free circuit. The figure illustrates how the count-up waveform at 10 kHz can be used to find a faulty circuit with an open in M1 using the S_{del} ORA metric. This fault is said to be detectable and identifiable by this waveform as illustrated in the separation between the two histograms of Figure 4.4. The histogram of Fig. 4.5 outlines how the TPG waveform cup at 10 kHz and two hundred millivolts does not distinguish an open in capacitor cl and an open in resistor rl. In this figure, the solid histogram of the fault-free circuits overlap the composite histogram of the faulty circuits (shown by the barely visible dotted curve). Therefore, the

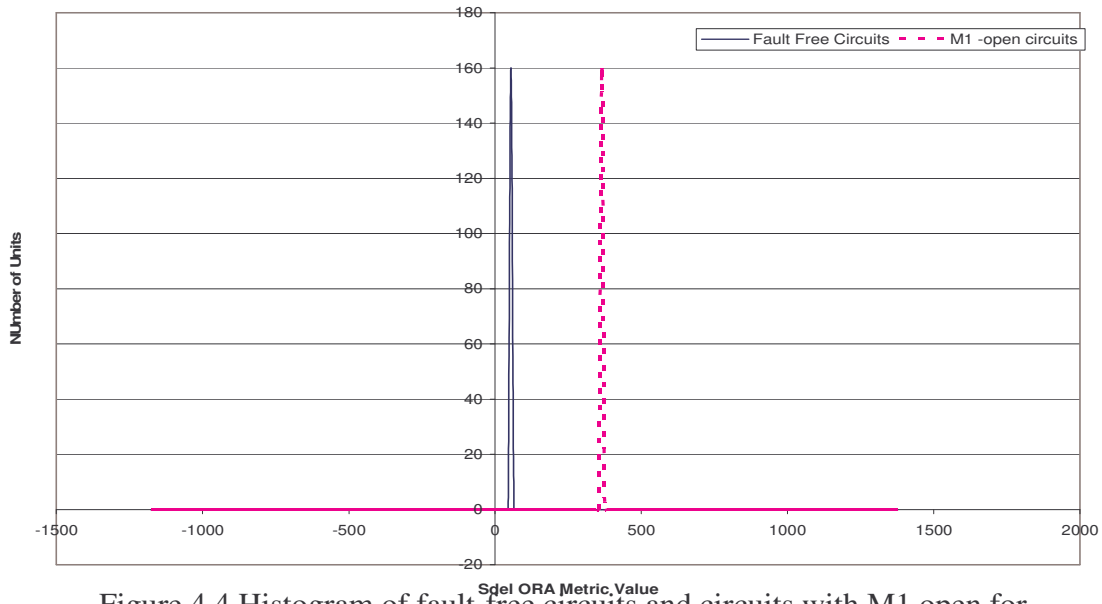


Figure 4.4 Histogram of fault-free circuits and circuits with M1 open for OpAmp1 with 200 mV Cup waveform at 10 kHz clock frequency.

TPG waveform Cup did not result in well separated histograms for faulty and fault-free circuits with the S_{del} ORA metric. These faults are considered to be undetectable and therefore bring the fault coverage of the test vector down. It can be seen from the schematic of the operational amplifier that these two components, rl and cl don't effect circuit operation at a low clock frequency of 10 kHz when they have open circuit faults.

4.3 Receiver Operating Characteristics

To address the issue of fault coverage and selection of ORA metric decision thresholds for deciding the presence of faults, we draw upon earlier work on receiver operating characteristics (ROC). ROC is considered as a method to analyze the fault coverage illustrated in the previous section. Consider the simplified probability density function(pdf) of Fig. 4.1, in which two Gaussian pdf's occur with different means and variances where the Gaussian pdf is:

$$p(x) = \frac{1}{\sigma_x \sqrt{2\pi}} e^{\frac{-x^2}{2\sigma_x^2}} \qquad (4.1)$$

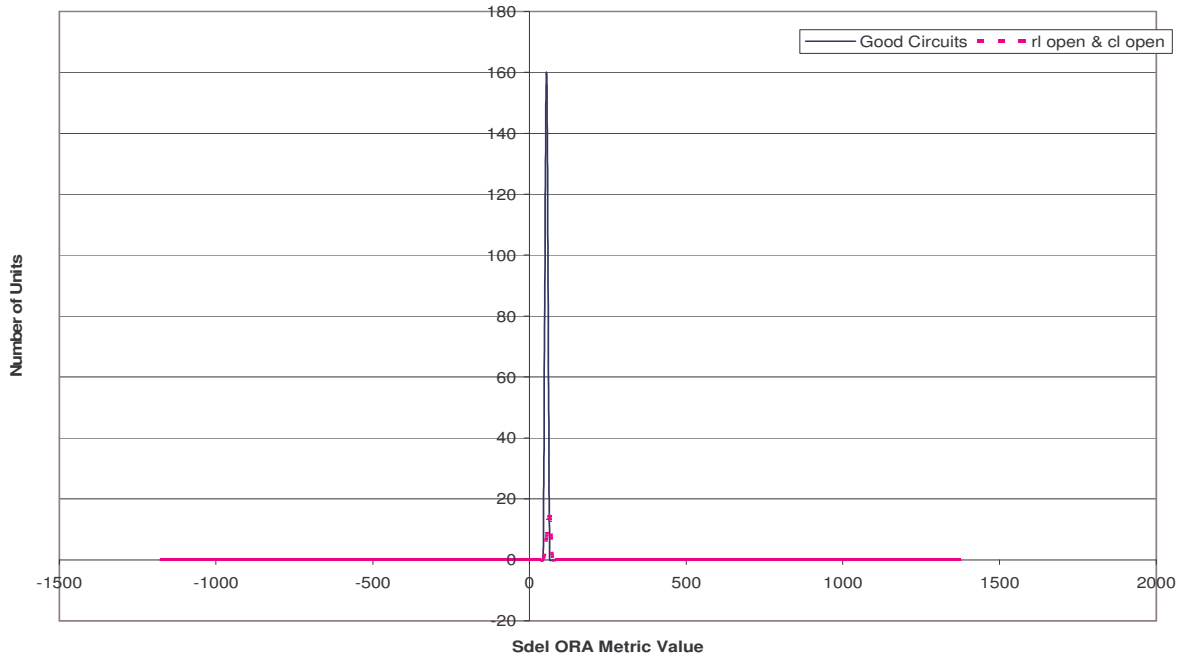


Figure 4.5 Histogram of Fault Free Circuits and Circuits with Rl open and Cl open for OpAmp1 with 200 mVolt Cup waveform at 10 KHz Clock Frequency

The pdf on the left of Fig. 4.6 represents a Gaussian distribution of some ORA metric for fault-free circuits and the distribution on the right represents the distribution for some particular fault. Let boundary A, be the decision threshold between fault-free and faulty circuits. All of the units to the left of the boundary are classified fault-free and all of the circuits to the right of boundary A are classified bad. ROC is used to set this threshold and analyze what happens when the threshold is varied.

The false alarm rate (equivalent to the sum of false positives and false negatives discussed in chapter 1) can be modeled by the receiver operating characteristics, or the receiver operating curve, as shown in Figure 4.7.

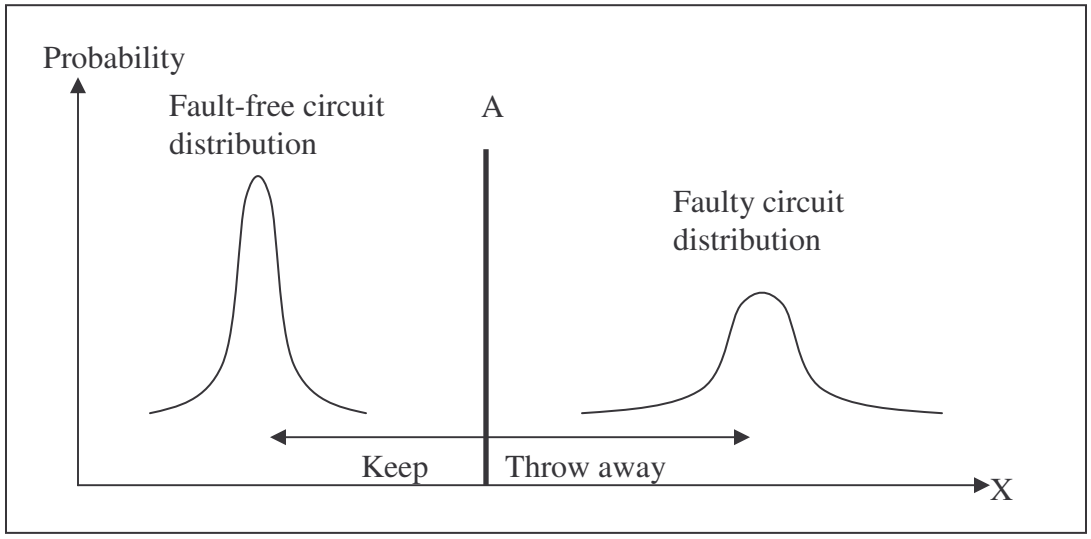


Figure 4.6 Histogram illustrating false positives and false negatives.

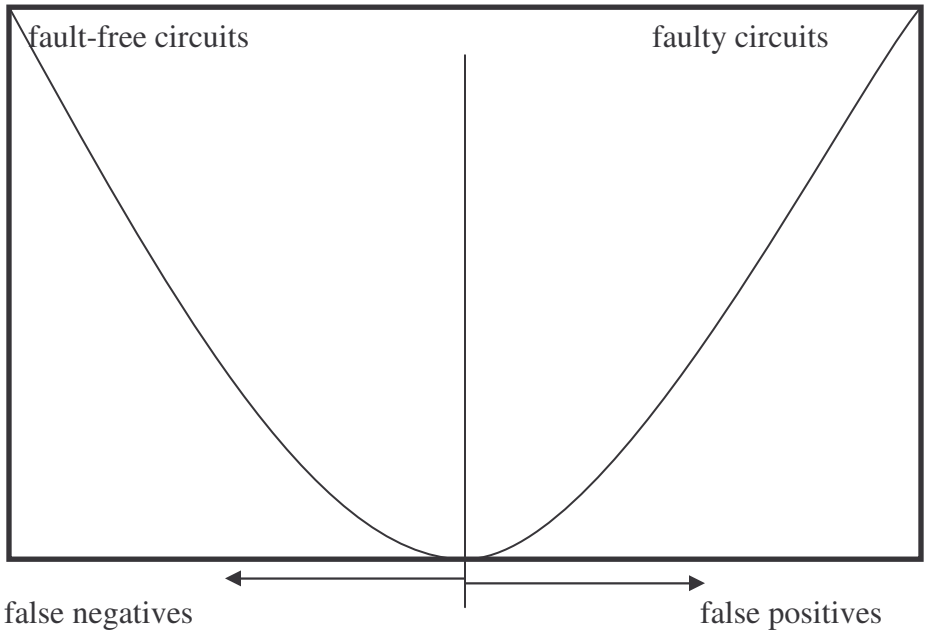


Figure 4.7: Receiver operating curve.

Figure 4.7 depicts how moving the boundary A to the left will decrease the number of false positives but will increase the number of false negatives at a much

greater rate given Gaussian data. The ROC curve shows the trade off between the selectivity and sensitivity. The ROC curve can be used to select the best operating point as a trade off between selectivity and sensitivity. The threshold would be chosen so that the threshold, A, gives the best trade off between the total number of false positives and false negatives. The threshold can be calculated by equation 4.2 where the average expected cost of placing threshold A at point x, which takes the cost of a false positive, α, the cost of missing a positive, β, with proportion of cases, ρ, and the location of the boundary x[24].

$$\cos t = (1-\rho)\alpha x + \rho\beta(1-x) \qquad (4.2)$$

Equation 4.2 can be used to determine the cost of choosing the threshold of boundary A, of Fig. 4.6. The equation shows the complementary relationship of ρ, the cost proportion, to the position of the boundary x. The relationship of Fig. 4.2 shows that moving the boundary too close to the fault-free circuit pdf will cause an increase in costs proportional to the cost proportion constant ρ. False positives and false negative do not incur the same cost in the relationship of equation 4.2, allowing further flexibility.

4.4 Bhattacharyya Distance and Fast TPG Pattern Searching

A simple way for quantifying the false alarm rate is the Bhattacharyya Error or B-distance. This metric is not only useful in the one dimensional case such as Fig. 4.6, but also multi-dimensional cases. Also, it can be used to help speed up fault simulation by providing a metric for choosing more promising TPG waveforms in the branches of a tree structured search for the best waveform in fault simulation.

The solution to finding a TPG waveform efficiently presents many challenges. One solution would be to first run a full batch of fault-free parametrically randomized

circuits for every waveform. From such a simulation the statistical parameters needed for Gaussian statistical characterization of fault-free circuit ORA metrics can be derived. The parameters needed include variance, correlation, standard deviation σ (sigma), and μ (statistical mean) for all ORA metrics. This data would then provide a statistical model for each of the six ORA metrics, S_{out}, S_{del}, S_{mag}, S_{16out}, S_{16del}, and S_{16mag}.

The next step is to take a small statistical sample of circuits with each fault, with each test vector, and ORA metric for comparison with the fault-free circuit data that was collected in the first step. The statistical sample of runs would then be analyzed to determine which test vectors were most promising for exposing faults. There are numerous potential approaches once the statistical data for the fault-free circuits and the faulty circuits is estimated for all the TPG waveforms and ORA metrics. The next question is how to use this statistical data to determine which test vectors will be committed to complete simulation.

Some possible techniques for using the statistical sample to choose promising waveforms included using a six-sigma distance of each test vector to eliminate the test vectors with mean that lie within the six-sigma distance of the fault-free circuits. The problem with this strategy was that some of the TPG waveforms provided excellent fault coverage with certain ORA metrics while other ORA metrics from the same TPG waveform did not. Thus requiring retaining a test vector even if two out of three digital ORA metrics performed poorly.

In a more powerful approach, the efficacy of TPG waveforms and ORA metrics can be ranked by calculating the Bhattacharyya distance between two multi-variate Gaussian distributions and thus estimate error bound, or equivalently, the fault

coverage[1]. The requirements for the Bhattacharyya distance are two sets of multivariate data that have a Gaussian pdf[1]. To the extent that the data is Gaussian, the Bhattacharya distance and error bound techniques can be used to generate a TPG waveform and ORA ranking.

Bhattacharyya distance is a measure of the distance between two sets of multivariate Gaussian pdf's used to calculate the Chernoff error bound when quantifying the hypothetical statistical differentiation between classes. The Bhattacharyya distance is found from the mean and standard deviation of the two sets of multivariate Gaussian data. The multivariate mean and multi-variant standard deviation from two sets of Gaussian pdf's are used for the calculation of the B-distance and to estimate the error.

In the more simple case of one dimensional data, the scalar means and standard deviations are first needed for each set of data [2]. The scalar mean is found from the limit,

$$\mu = \frac{1}{n}\sum_{k=0}^{k=n} x_k \qquad (4.3)$$

The scalar standard deviation and variance for the values can also be derived from data. There are six sets of ORA metric values for each TPG pattern in the ORA files that can used as the multivariate data for the Bhattacharyya distance algorithm outlined above [2]. The next step in the process of ranking the TPG patterns by the best possible error of any of three ORA values lies in calculating the Bhattacharyya distance to estimate the error for each fault, for each ORA metric and TPG waveform. The Bhattacharyya

distance [3]:

$$B(S_1,S_2)=\frac{1}{8}(\bar{u}_1-\bar{u}_2)^T\left[\frac{\sum_1-\sum_2}{2}\right]^{-1}(\bar{u}_1-\bar{u}_2)+\frac{1}{2}\ln\left\{\frac{\left|\frac{1}{2}\left(\sum_1+\sum_2\right)\right|}{\left|\sum_1\right|^{\frac{1}{2}}\left|\sum_2\right|^{\frac{1}{2}}}\right\} \quad (4.4)$$

Where μ_1 and μ_2 are the vector means of 2 classes, $\sum_1$ and $\sum_2$ are covariance matrices [3]. For the purposes of illustration, a scalar form will be considered. The B-distance formula then becomes in scalar form,

$$B(S_1,S_2)=\frac{1}{4}\frac{(\mu_1-\mu_2)^2}{\sigma_1^{\,2}+\sigma_2^{\,2}}+\frac{1}{2}\ln\left\{\frac{\frac{1}{2}\left(\sigma_1^{\,2}+\sigma_2^{\,2}\right)}{\sigma_1\sigma_2}\right\} \quad (4.5)$$

Where μ_1 is the mean for the fault-free circuits and μ_2 is the mean for the data, and σ_1 and σ_2 are the standard deviations. The error bound is then found through the following relationship [2].

$$Error=\frac{1}{N}\sum_{faultS_1}\exp\left(B(S_1,S_2)\right) \quad (4.6)$$

In this scalar example, of equation 4.4 produces three error values per TPG pattern, one for each ORA value from which the data was derived. This is then used to determine the effective error rates of each individual feature of the Gaussian data, or the fault coverage of different TPG waveforms and ORA metrics.

The TPG waveforms are then sorted according to the lowest error bound. The end result is a list ranking the most promising TPG waveforms and ORA metrics for maximum fault coverage. Future research may consider using the vector form of the Bhattacharyya measure instead of the scalar form of this discussion.

REFERENCES

[1] Pratt, Willaim, *Digital Image Processing*, 2nd ed. New York: John Wiley and Sons Press, 1991.

[2] Stroud, Charles, *A Designers Guide to Built in Self Test*, Boston: Kluwer Academic Publishers, 2000.

[3] T. P. Weldon, Y. A. Gryazin, and M. V. Klibanov, "Comparison of 2D and 1D Approaches to Forward Problem in Mine Detection", *Proceedings of the SPIE*, vol. 4038, pp. 1140-1148, 2000.

[4] A. Khocke, S.D. Sherlekar, G. Venkatesh, R. Venkateswaran, "A Behavioral Fault Simulator for Ideal", *IEEE Design and Test of Computers*, Vol. 4 No. 2, pp14-20, 1992.

[5] P. N. Variyam, A. Chatterjee, " Digital-Compatible BIST for Analog Circuits Using Transient Response Sampling", *IEEE Design and Test of Computers,* vol.6, no.3, 106-115, 2000.

[6] P. N. Variyan, A. Chatterjee. "Test Generation for Comprehensive Testing of Linear Analog Circuits Using Transient Response Sampling" *Proceedings of 1997 International Conference on Computer-Aided Design*, Washington, DC, pp 1146-1151, Nov, 1997.

[7] A. Chatterjee, B. C. Kim, N. Nagi, "Design for Testability and Built-In Self-Test of Mixed-Signal Circuits: A Tutorial" *Proceedings of the 10th International Conference on VLSI Design*, , Hyperbad, India, pp 388-392, Jan, 1997.

[8] Y. V. Malyshenko, "Functional Fault Models for Analog Circuits" *IEEE Design and Test of Computers,* vol. 6, No.5 pp. 80-85 Apr 1998 pp 80 85

[9] K. Arabi, B. Kaminska, " Parametric and Catastrophic Fault Coverage of Analog Circuits in Oscillation-Test Methodology" *Proceedings of the 15th IEEE VLSI Test Symposium,* Monterey, Ca. pp. 166-171, Apr. 1997.

[10] Chatterjee, B. C. Kim, N. Nagi, "DC Built-In Self-Test for Linear Analog Circuits" *IEEE Design and Test of Computers* vol.5, no.3 pp 26-33, Summer 1996.

[11] Grochowski, D. Bhattacharya, T.R. Viswanathan & K. Laker "Integrated circuit testing", *IEEE Trans. on Circuits and Systems II Analog and Digital Signal Processing* vol.44, no.8, pp. 610-633, 1997.

[12] Baker K., Richardson A.M., Dorey A.P., "Mixed Signal Test - Techniques, Applications and Demands"*IEE Proc.-Circuits Devices Systems*, vol.143, no.6 pp 86-102 , Dec. 1996.

[13] Agreement No. F30602-97-1-0042, *Mixed Signal Based Built-In Self-Test for Analog Circuits*, Final Report, Stroud, C., Bradley, E.

[14] Sunter J., "Mixed-Signal BIST - Does Industry Need it?" *Proc. 3rd IEEE International Mixed Signal Workshop*, Seattle Wa, Tutorial 2, June 1997.

[15] Wang, C.-J., Wey C.-L., "Efficient Testability Design Methodologies for Analog/Mixed-Signal Integrated Circuits" *3rd Int. Mixed Signal Testing Workshop,* Seattle, pp. 68-74, June 1997.

[16] Novak, F., Mozetic I., Santo-Zarnik M., Biasizzo A., "Enhancing Design-for-Test for Active Analog Filters by Using CLR(R)" *Analog Integrated Circuits and Signal Processing*, vol.4 pp. 215-229, Sept. 1993.

[17] Vazquez D., Rueda A., Huertas J. L., Peralias E., "Unified off- and on-line testing in analogue circuits: concept and practical demonstrator" *Proc. 3rd IEEE International Mixed Signal Workshop*, Seattle, pp. 169-174, June 1997.

[18] Bratt, A.H., Richardson, A.M., Harvey, R.J. & Dorey, A.P. "A Design-For-Test Structure for Optimising Analog and Mixed Signal IC Test", *European Design and Test Conference*, Paris, France, pp. 24-34, Mar. 1995.

[19] Mir, S., Lubaszewski, M., Courtois, B., "Unified Built-In Self-Test For Fully Differential Analog Circuits" *Journal of Electronic Testing: Theory and Applications*, vol.9, no.1-2, pp. 135-151, Sept. 1996.

[20] Wey, C.-L, "Built-In Self-Test (BIST) Structure for Analog Circuit Fault Diagnosis" *IEEE Trans. on Instrumentation and Measurement*, vol.39, no.3, pp.517-521, 1990.

[21] Arabi, K., Kaminska, B., "Oscillation Built-In Self Test (OBIST) Scheme for Functional and Structural Testing of Analog and Mixed-Signal Integrated Circuits"*Proc. 1997 International Test Conference*, Washigton DC, pp786-796, Nov.1997.

[22] Sedra, A., Smith, K., *Microelectronic Circuits*, 4th ed. New York, Oxford University Press, 1998.

[23] University of California at Berkley: *BSIM3 Information Guide* www.device.eecs.berkeley.edu/~bsim3/,

[24] Mentor Graphics Corporation Website: *Produce Information Guide* http://www.mentor.com/eldo/overview.html

[25] Oppenheim, A., Schafer, R., *Discrete-Time Signal Processing*, 2nd ed., Upper Saddle River, 1998.

[26] Kaminska, B. et. al., "Analog and Mixed Signal Benchmark Circuits-First Release", *Proc. The 1997 International Test Conference*, Washington DC, Nov. 1997.

[27] Kondagunturi, R., et. al., "Benchmark Circuits for Analog and Mixed Signal-Signal Testing", *Proc. 1999 IEEE Southeast Conference,* Lexington, Ky, Mar. 1999.

[28] Maggard, K., et. al. "Built-In Self-Test for Analog Circuits in Mixed-Signal Systems", *Proc. 1999 IEEE Southeast Conference,* Lexington, Ky, Mar. 1999.

APPENDIX A: CLASS LIBRARIES AND THEIR FUNCTIONS

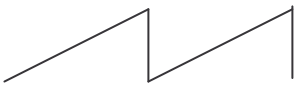
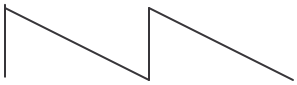
Module	Function
Ora	Processes SPICE output files and generates the ORA metric data
Gsrc	Processes G models in SPICE
Vsrc	Processes V components
Circuit	Loads, parses, and writes circuit files
DotEnds	Concludes processing with instance of .end statement
Other	Processes statements not included present Library i.e. diode, .probe, .ac, .dc etc
Inductor	Processes inductors
Xsubckt	Processes sub-circuit statements in SPICE
CircuitStats	Processes parametric variations of R, L, C and mos.
DotSubckt	Processes .subckt statements in SPICE
Resistor	Processes resistor statements in SPICE
Isrc	Processes current source statements in SPICE
Comment	Processes commented lines in SPICE any line with *
Esrc	Processes SPICE voltage controlled voltage source statements
Statistics	Processes Statistical parameters for CircuitStat's parametric variations
Component	Loads circuit and processes all known components
Mos	Processes MOS components and their faults
Faultlist	Generates faultlist and for faulty files
Tpg	Generates test patterns for fault simulator
Faultsim	Main program executable
Anarun	Processes ORA output and generates PDF histograms

Capacitor	Processes capacitors and their faults
Data	For processes various data arrays in file writing

APPENDIX B: LIST OF TPG WAVEFORMS

TPG Waveform	**Abbreviation**
1. Count Up	cup
2. Count down	cdwn
3. Count Up Down	cud
4. Count Up w/bit reversal	cuR
5. Count down w/bit reversal	cdR
6. Count Up Down w/bit reversal	cudR
7. Linear Frequency Shift Register	lsfr
8. Frequency Sweep	fswp
9. Frequency Sweep w/bit reversal	fswpR
10. Frequency Sweep w/ Constant Amplitude	fswpC
11. Frequency Sweep w/ Constant Amplitude and Bit Reversal	fswpRC

The following are pictorial diagrams illustrating the shape or appearance of TPG waveforms.

Pattern	**Pictorial**
cup	
cdwn	
cud	
cuR	noise-like
cdR	noise-like
cudR	noise-like
lsfr	noise-like
fswp	
fswpR	random amplitude, random period
fswpC	
fswpRC	constant amplitude, random period

APPENDIX C: SPICE NET LIST FOR OPERATIONAL AMPLIFIER

```
*Operational amplifier Hspice Netlist
*tpw opamp: out vin+ vin- +5volt -5volt
.subckt OpAmp 9 11 12 13 14
R1 1 14 110E3
M1 1 1 13 13 PMOS L=4U W=150U
M2 3 1 13 13 PMOS L=4U W=35U
M3 9 1 13 13 PMOS L=4U W=100U
M4 4 12 3 3 PMOS L=4U W=60U
M5 5 11 3 3 PMOS L=4U W=60U

cl 5 16 1.27E-12

rl 16 9 8750
M6 4 4 14 14 NMOS L=4U W=27.5U
M7 5 4 14 14 NMOS L=4U W=27.5U
M8 9 5 14 14 NMOS L=4U W=100U

.MODEL NMOS NMOS (                                LEVEL  = 3
+ TOX    = 3.1E-8          NSUB   = 1.763642E15     GAMMA  = 0.721254
+ PHI    = 0.7             VTO    = 0.5944737       DELTA  = 0.913057
+ UO     = 652.3781644     ETA    = 9.998788E-4     THETA  = 0.0712612
+ KP     = 7.319728E-5     VMAX   = 2.51124E5       KAPPA  = 0.5
+ RSH    = 0.0981893       NFS    = 4.760633E11     TPG    = 1
+ XJ     = 3E-7            LD     = 0               WD     = 7.519702E-
7
+ CGDO   = 1.67E-10        CGSO   = 1.67E-10        CGBO   = 1E-10
+ CJ     = 2.879473E-4     PB     = 0.8976295       MJ     = 0.5
+ CJSW   = 1.18445E-10     MJSW   = 0.05            )

.MODEL PMOS PMOS (                                LEVEL  = 3
+ TOX    = 3.1E-8          NSUB   = 1E17            GAMMA  = 0.4794113
+ PHI    = 0.7             VTO    = -0.8594243      DELTA  = 0.4719726
+ UO     = 100             ETA    = 0.9984189       THETA  = 0.1358457
+ KP     = 2.489648E-5     VMAX   = 1.052858E5      KAPPA  = 0
+ RSH    = 35.4503246      NFS    = 5.538975E11     TPG    = -1
+ XJ     = 2E-7            LD     = 9.78062E-15     WD     = 1E-6
+ CGDO   = 1.98E-10        CGSO   = 1.98E-10        CGBO   = 1E-10
+ CJ     = 2.872176E-4     PB     = 0.7469896       MJ     = 0.4224801
+ CJSW   = 1.402728E-10    MJSW   = 0.0702615       )

.ends OpAmp

Xop1 16 17 18 2 10 OpAmp
VDD 2 0 5
VSS 10 0 0

.end
```

APPENDIX D: SPICE NET LIST FOR REDUCED ORDER BIQUAD FILTER

```
* Schematics Version 9.2.2
* Fri Sep 13 14:55:32 2002

* From [PSPICE NETLIST] section of h:\apps\pspice\PSpice\PSpice.ini:
* HPO:Node 2, BPO:Node 9, LPO: Node 13
* Input nodes: 5 & 2

R3         2 1  10000
R2         3 2  10000
R7         8 9  7.5000
R4         9 10  10000
R6         12 8  3010
R1a        3 4  20000
R1b        12 3  20000
R5         3 13  10000
R3a        7 2  100
R3b        16 9  100
R3c        15 13  100

E2         0 7 8 , 3 1000000
E6         0 15 12 , 10 1000000
E5         0 16 12 , 1 1000000

D2         2 6 Dbreak
D3         0 2 Dbreak
D8         0 9 Dbreak
D9         9 11 Dbreak
D10        0 13 Dbreak
D11        13 14 Dbreak

C1         1 9  0.015U
C2         10 13  0.015U
C5         5 4  1
C11        0 12  10U
C7         0 12  0.1U



V15        6 0 5V
V18        11 0 5V
V19        14 0 5V

.MODEL Dbreak D(IS=1E-15)

*Vin        5 0 DC 0V
V13        12 0 2.5V

*.PROBE
.END
```

APPENDIX E: FAULTSIM MANUAL

The following pages are manual pages from the fault simulator software, faultsim, class libraries, and anarun. This appendix contains detailed information for each class library, the main executable faultsim, and the post processing executable anarun. The detailed information includes program flow, usage information, class functions, and variable descriptions along with various data members.

```
. ------------------------------------------------------------------------
.
.                                   faultsim.cpp
.
.                                   Rev 2.0
.
. ------------------------------------------------------------------------
. faultsim.cpp is a c++ main() executable program
.
.  it uses classes from ../classes directory
.  for help, run faultsim without arguments and it will
.   print a help screen
.
.
. NOTE: for latest usage information, run faultsim at the
.                   command line without any arguments!!
.
.
. Usage:
.
.  faultsim ckt.cir numproc numrand inpos inneg outpos outneg vbias vamp maxcpu
vomin vomax repnum
.
.          - ckt.cir is the circuit spice file
.          - numproc is number of processes that are forked()
.            to run in parallel on a multi-cpu machine
.          - numrand is number of randomizations per fault
.          - inpos,inneg are pos and neg differential input nodes
.          - outpos,outneg are pos and neg differential output nodes
.          - vbias is test pattern dc bias, where
.                           inpos=vbias-(vampl/2) to inpos=vbias+(vampl/2)
.                           and inneg=vbias
.                      if vbias=0, then a true floating input is used
.          - vamp test pattern amplitude in volts
.          - maxcpu is max number cpu seconds allowed per spice run
.          - vomin to vomax is differential output voltage range
.          - repnum num or repetitions to run waveform
.
.     Example:
.          faultsim benchmark.cir 2 5 17 18 16 0 2.5 0.2 30 0 5

. Functional summary of faultsim
.
.          1. controls parallel execution on a multi-cpu machine
.
.          2. first creates a directory "rundataxx" for the run
```

```
.
.       3. then, creates directory "rundataxx/circ" and stores
.               randomized template spice circuit files there,
.               with one ".cir" file per randomized faulty circuit
.
.               the original spice file is entered as an argument
.               in the faultsim command line
.
.               the template file has a comment line "tpwtpgheretpw"
.               used as a marker for the location of the test pattern
.               to be inserted
.
.               numrand command line argument determines number of
.               randomizations per fault (i.e, 100 random circuits with
.               R1 open-circuited, 100 with R2,  ... etc.)
.
.       4. then, faultsim spawns (using fork()) several executables
.               "runtpg" that make use of multiple cpu's
.
. FLAGS ------------------------------------------------------------
.
.       Check the header file for any useful debug flags
. initial simulation set-up:
.
.       - create data directory rundatxxxx where xxx is date
.       - create subdirectory,
.               creates directory "rundataxx/circ" and stores
.               randomized template spice circuit files there,
.               with one ".cir" file per randomized faulty circuit
.
.               the original spice file is entered as an argument
.               in the faultsim command line
.
.               the template file has a comment line "tpwtpgheretpw"
.               used as a marker for the location of the test pattern
.               to be inserted.
.       - set up statistics for simulation
.               process stats refer to lot-to-lot batch-batch process variations
.               and hence tend to be large standard deviations
.               chip stats refer to variations within a single chip
.
.               default statistics:
.
```

	Process stats			Chip Stats		
	pdf	mean	std-dev	pdf	mean	std-dev
Resistors:	gauss	1	0.1	gauss	1	0.04
Capacitors:	gauss	1	0.11	gauss	1	0.03
Inductors:	gauss	1	0.12	gauss	1	0.02

- create directory for rundata
    name it rundataxxx, where xxx is a date stamp

- generate faults and randomized files
    store in directory rundataxxx/spicefiles

    default faults:

	Faults		
	open	short	parametric
Resistors:	1e9 ohm	1	0.1
Capacitors:	2e-18 F	2	0.2
Inductors:	3e9 H	3E-18	0.3
MOSFET:	1e8 Ohm	3	0.12 (drain-source open/shorts)

- create Tpg files
    test all waveforms at 10KHz, 100KHz, 1 MHz

    possible waveforms
    cup, cdwn, cud, cuR, cdR, para, paraR, pulse,
    cudR, const, lfsr, fswp, fswpR, fswpC, fswpRC\n");

    includes lines for tapping into circuit for input nodes
        and output nodes (disabled if any argument is NULL)
        also, vbias="" disables DC bias creating pure floating
            differential input

    - - inpos,inneg are pos and neg differential input nodes
        for differential input voltage source
    - outpos,outneg are pos and neg differential output nodes
    - vbias test pattern dc bias,
        where inpos=vbias-(vin/2) through vbias+(vin/2)

```
.                       and inneg = vbias
.
.               (output file name is contained in filename class member)
.               Usage:
.
.               possible waveforms
.               cup, cdwn, cud, cuR, cdR, para, paraR, pulse,
.               cudR, const, lfsr, fswp, fswpR, fswpC, fswpRC);
.
.
.       - generate one scratch directory "procx" per parallel processes
.       - generate one ora directory for all ora results
.       - run all good circuits first
.         with parallel processes using fork
.
.       Method:
.               while loop on tpg files (test patterns)
.                 while loop on good spicefiles (circuits without test atterns)
.                   erase all process directories (procx)
.                   for loop over number of processes (parallel threads)
.                       copy/merge tpg/spice file to procx directories
.                       fork parallel processes
.                       run eldo on all procx directories
.                       create ora files (output response analaysis)
.                   end loop number processes
.                   copy ora to master ora files in directory ora
.                 end loop good spicefiles
.               end loop tpgfiles
.
.       - run all faulty circuits next
.         with parallel processes using fork
```

```
. ------------------------------------------------------------------------
.
.                                   Capacitor.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Capacitor.cc is a c++ clas for capacitor devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Capacitor.h
.
. class Capacitor
. {
.
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="C";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;   //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;     //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ---------------------------------------------------------
.
.       Check the header file for any useful debug flags
.
. CONSTRUCTORS --------------------------------------------------
. Function: Capacitor::Capacitor()
.               default constructor
.               Assigns following defaults:
.                       rawline="Cdefault"
.                       linenumber=0
.                       type="C"
.                       name="Cdefault"
.                       value=0;
```

```
.                       model="Cdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Cdefault"
.                       rawline="Cdefault"
.                       trackerr=0
.                       randomerr=0

. Function: Capacitor::Capacitor(char * xrawline, int xlinenumber)
.           constructor from raw spicefile line

. Function: Capacitor::Capacitor(char * xname, double xvalue, int xnodeplus,
.           int xnodeneg, int xlinenumber)
.           constructor from data
. Function: Capacitor::~Capacitor()
.           default destructor
. Function: Capacitor& Capacitor::operator=(const Capacitor & r)
.           overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Capacitor::loadline(char * xrawline,int xlinenumber)
.           loads capacitor with data translated from a spice-formatted line
.           Assigns following defaults:
.                   rawline="Cdefault"
.                   linenumber=0
.                   type="C"
.                   name="Cdefault"
.                   value=0;
.                   model="Cdefault"
.                   numnodes=2
.                   nodelist=0 0
.                   remainderline="Cdefault"
.                   rawline="Cdefault"
.                   trackerr=0
.                   randomerr=0
. Function: void Capacitor::writefile(ofstream * xfname )
.           writes a capacitor to the file handle xfname
.
.           file is assumed to already be opened
.           file is not closed
. Function: void Capacitor::writefile(ofstream * xfname , CircuitStats & cs )
.           writes a randomized capacitor to the file handle xfname
.           file is assumed to already be opened
.           file is not closed
. Function: void Capacitor::print()
.           prints capacitor stderr
.           Usage: a.print();
```

```
. Function: void Capacitor::setvalue(double xvalue)
.                   sets capacitor value
. Function: void Capacitor::scalevalue(double xscale)
.                   sets capacitor value to value times xscale
. Function: char * Capacitor::getname()
.                   gets capacitor name
.  END
```

```
. ------------------------------------------------------------------------
.
.                                   Circuit.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Circuit.cc is a c++ clas for a Circuit as would correspond to the
.       top-level circuit in a spice netlist file.
.       The basic structure is defined in Circuit.h
.
. class Circuit
. {
.
. private:
.
.       char * spicefilename;  //name of Original spicefile
.
.       char * spicefilename;  //Original spicefilename loaded in memory
.
.       Component * pC;                 //pointer to objects corresponding to
.                                       //   various components of circuit
.       int numcomponents;   //number of components
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Circuit::Circuit()
.               default constructor
.               Assigns following defaults:
.       char * spicefilename="Circuit Not Loaded";
.       int numcomponents=0;
.       Component * pC=NULL;
. Function: Circuit::~Circuit()
.               default destructor
. FUNCTIONS ------------------------------------------------------
. Function: Circuit::loadfile(char * xfilename)
.               loads Circuit with data translated from a spice-file
. Function: void Circuit::writefile(char * xfname)
.               writes a component to the file named xfname
.               file is opened and closed
. Function: void Circuit::writefile(char * xfname, CircuitStats & cs )
.               writes a randomized circuit to the file named xfname
.               file is opened and closed
```

```
. Function: void Circuit::print()
.                   prints Circuit stderr
.                   Usage: a.print();
. Function: int Circuit::getnumcomponents()
.                   returns numcomponents
. Function: char * Circuit::getspicefilename()
.                   returns spicefilename
. Function: Component Circuit::getcomponent(int n)
.                   returns component n
```

```
. ------------------------------------------------------------------------
.
.                                   CircuitStats.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. CircuitStats.cc is a c++ class for statistical functions
.
.       The basic structure is defined in CircuitStats.h
. class CircuitStats
. {
. private:
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: CircuitStats::CircuitStats()
.               default constructor
.               Assigns following defaults:
.                       comment="Default uniform"
.                       pdf1="uniform"
.                       pdf2="disabled"
.                       mean1=0;
.                       sigma1=1;
.                       mean2=0;
.                       sigma2=0;
. Function: CircuitStats::~CircuitStats()
.               default destructor
. Function: CircuitStats & CircuitStats::operator=(const CircuitStats & stat)
.               overloaded equal
. Function: void CircuitStats::SetCircuitStats(char * xcomment,
.               char * xrprocesspdf, double rprocessmean, double rprocesssig,
.               char * xrchippdf, double rchipmean, double rchipsig,
.               char * xcprocesspdf, double cprocessmean, double cprocesssig,
.               char * xcchippdf, double cchipmean, double cchipsig,
.               char * xlprocesspdf, double lprocessmean, double lprocesssig,
.               char * xlchippdf, double lchipmean, double lchipsig)
.                load statistics
. FUNCTIONS ------------------------------------------------------
. Function: void CircuitStats::genchip( )
.               returns a random number
. Function: double CircuitStats::scaleresistor( )
.               returns a random number
```

```
. Function: double CircuitStats::scalecapacitor( )
.                   returns a random number
. Function: double CircuitStats::scaleinductor( )
.                   returns a random number
. Function: void CircuitStats::writefile(ofstream * xfname )
.                   writes a CircuitStats object to the file handle xfname
.                   it is written as a spicefile comment
. Function: void CircuitStats::print()
.                   prints resistor stderr
.                   Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Comment.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Comment.cc is a c++ clas for Comment lines as would be found in
.       a spice netlist file.
.       The basic structure is defined in Comment.h
. class Comment
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber;                 //linenumber in original spicefile
.       char * type;            //device type, i="Comment"
.       char * name;            //="Comment"
.       double value;           //=0
.       char * model;           //="Comment"
.       int numnodes;           //=0
.       int nodelist[2];        //= 0 0
.                               //
.       char * remainderline;   //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value
.       double trackerr;        //=0
.       double randomerr;       //=0
.                               //
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Comment::Comment()
.               default constructor
.               Assigns following defaults:
. Function: Comment::Comment(char * xtext,int xlinenumber)
.                constructor from data

. Function: Comment::~Comment()
.               default destructor
. Function: Comment& operator=(const Comment& com)
.               overloaded equal
. FUNCTIONS ------------------------------------------------------
```

```
. Function: Comment::loadline(char * xrawline,int xlinenumber)
.                   loads Comment with data translated from a spice-formatted line
. Function: void Comment::writefile(ofstream * xfname )
.                   writes a Comment to the file handle xfname
                    file is assumed to already be opened
.                   file is not closed
. Function: void Comment::print()
.                   prints Comment stderr
.                   Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Component.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Component.cc is a c++ clas for Component devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Component.h
.
. class Component
. {
. private:
.       char * type;            //device type, i.e., R, L, C, V, Comment, unknown
.       int linenumber;                 //linenumber in original spicefile
.       Resistor * pR;          // pointer to object actually containing component
.       Inductor * pL;          //   only one pointer should be non-NULL
.       Capacitor * pC;
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Component::Component()
.               default constructor
.               Assigns following defaults:
.       char * type="Component Undefined";
.       int linenumber=0;
.       Resistor * pR=NULL;
.       Inductor * pL=NULL;
.       Capacitor * pC=NULL;
. Function: Component::Component(Component &)
.               copy constructor
. Function: Component::~Component()
.               default destructor
. Function: Component& Component::operator=(Component & comp)
.               overloaded equal
. FUNCTIONS ------------------------------------------------------
. Function: Component::loadline(char * xrawline,int xlinenumber)
.               loads Component with data translated from a spice-formatted line
. Function: void Component::writefile(ofstream * Xsubcktfname)
.               writes a component to the file handle Xsubcktfname

.               file is assumed to already be opened
.               file is not closed
```

```
. Function: void Component::writefile(ofstream * Xsubcktfname, CircuitStats & cs)
.                   writes a randomized component to the file handle Xsubcktfname
.
.                   file is assumed to already be opened
.                   file is not closed
. Function: void Component::print()
.                   prints Component stderr
.                   Usage: a.print();
. Function: int Component::isresistor()
.                   return 1 if true, 0 if not
. Function: int Component::iscapacitor()
.                   return 1 if true, 0 if not
. Function: int Component::isinductor()
.                   return 1 if true, 0 if not
. Function: int Component::ismosfet()
.                   return 1 if true, 0 if not
. Function: char Component::gettype()
.                   return type
. Function: char * Component::getname()
.                   return name
. Function: void Component::setvalue(double xvalue)
.                   set component value to xvalue
. Function: void Component::faultdrainopen(double xvalue)
.                   set drain to have series resistor of value xvalue
.                           typically used to open-circuit a fet
. Function: void Component::faultdrainsourceshort(double xvalue)
.                   set drain to have drain-source shunt resistor of value xvalue
.                           typically used to short-circuit a fet
. Function: void Component::scalevalue(double xscale)
.                   set component value to value times xscale
```

```
. ------------------------------------------------------------------------
.
.                                   Data.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Data.cc is a c++ clas for Data arrays
.       The basic structure is defined in Data.h
.
. class Data
. {
.
. private:
.                                               //numarray x arraysize data array
        char* comment;                          //
.       char** names;           //names of each data array
.       int numarray;           //number of arrays
.
.       double* rarray;         //array real part
.       double* iarray;         //array imaginary part
.       int arraysize;          //length of each array associated with
.                               //each name
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Data::Data()
.               default constructor
.               Assigns following defaults:
. Function: Data::Data(const Data& dat)
.               copy constructor
. Function: Data::Data(char * xcomment, int xnumarray, int arraysize)
.                constructor of particular size
. Function: Data::~Data()
.               default destructor
. Function: Data& operator=(const Data& com)
.               overloaded equal
. FUNCTIONS ------------------------------------------------------
. Function:  store(int xarraynum,
.               int xelnum,
.               double xr, double xi);  //store xr and xi into xrdata xidata
.                                       //of array number xarraynum
.                                       //at array element number xelnum
```

```
. Function: double readr(int xarraynum, int xelnum);
.                   read exeulnum'th element rarray value from
.                   from the xarraynum'th array
. Function: double readi(int xarraynum, int xelnum);

.                   read exeulnum'th element iarray value from
.                   from the xarraynum'th array
. Function: double readname(int xarraynum);
.                   read name of
.                    the xarraynum'th array
. Function: double storename(int xarraynum,char* xname);
.                   store name of
.                    the xarraynum'th array
. Function: Data getarray(int xnum)
.                   get
.                    the xnum'th array
. Function: void Data::rplothistoeps(char * fname,
.                   int xarraynum,
.                   double xmin, double xmax,
.                   int numbins);
.                   plot histogram of rarray xarraynum
.                   for range xmin to xmax
.                   with numbins bins
. Function: void Data::ploteps(int xarraynum,
.                           char * xfname, char * title,
.                           char * xaxlabel, char * yaxlabel )
.                   array number xnum contains the data
.                           rarray contains xaxis coordinate
.                                   iarray contains y coordnate
.                   writes an eps plotfile to the file  xfname
.                           title is placed at top of plot
.                   xaxlabel and yaxlabel are axis labels of plot
. Function: void Data::writefile(char * xfname )
.                   writes a Data to the file  xfname
. Function: void Data::print()
.                   prints Data stderr
.                   Usage: a.print();
. Function: double Data::rmean(int xarraynum)
.                   compute mean of rarray number xarraynum
. Function: double Data::imean(int xarraynum)
.                   compute mean of iarray number xarraynum
. Function: double Data::rmeansq(int xarraynum)
.                   compute mean square (second moment)
.                   of rarray number xarraynum
. Function: double Data::imeansq(int xarraynum)
.                   compute mean square (second moment)
```

```
.                   of iarray number xarraynum
. Function: double Data::rmin(int xarraynum)
.                   compute minimum of rarray number xarraynum
. Function: double Data::rmax(int xarraynum)
.                   compute maximum of rarray number xarraynum
. Function: double Data::imin(int xarraynum)
.                   compute maximum of iarray number xarraynum
. Function: double Data::imax(int xarraynum)
.                   compute maximum of iarray number xarraynum
. NON-MEMBER FUNCTIONS
**********************************************
. Function      float plot_limit_max(float)
.                   used to calculate maximum plotranges in plot_eps()
. Function      float plot_limit_min(float)
.                   used to calculate minimum plotranges in plot_eps()
```

```
. ------------------------------------------------------------------------
.
.                                   DotEnds.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. DotEnds.cc is a c++ class for .ends lines as would be found in
.       a spice netlist file.
.       The basic structure is defined in DotEnds.h
. class DotEnds
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="R";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ---------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS --------------------------------------------------
. Function: DotEnds::DotEnds()
.               default constructor
.               Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
.                       type="R"
.                       name="Rdefault"
.                       value=0;
.                       model="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Rdefault"
```

```
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
. Function: DotEnds::~DotEnds()
.               default destructor
. Function: DotEnds& DotEnds::operator=(const DotEnds & xde)
.               overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: DotEnds::loadline(char * xrawline,int xlinenumber)
.               loads resistor with data translated from a spice-formatted line
.               Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
.                       type="R"
.                       name="Rdefault"
.                       value=0;
.                       model="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Rdefault"
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
. Function: void DotEnds::writefile(ofstream * xfname )
.               writes a resistor to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void DotEnds::print()
.               prints resistor stderr
.               Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   DotSubckt.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. DotSubckt.cc is a c++ class for .ends lines as would be found in
.       a spice netlist file.
.       The basic structure is defined in DotSubckt.h
. class DotSubckt
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="R";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: DotSubckt::DotSubckt()
.               default constructor
.               Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
.                       type="R"
.                       name="Rdefault"
.                       value=0;
.                       model="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
```

```
.                       remainderline="Rdefault"
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
. Function: Resistor::~Resistor()
.               default destructor
. Function: DotSubckt& DotSubckt::operator=(const DotSubckt& sub)
.               overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Resistor::loadline(char * xrawline,int xlinenumber)
.               loads resistor with data translated from a spice-formatted line
.               Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
/.                      type="R"
.                       name="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Rdefault"
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Resistor::writefile(ofstream * xfname )
.               writes a resistor to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void Resistor::print()
.               prints resistor stderr
.               Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Esrc.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Esrc.cc is a c++ class for controlled-source devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Esrc.h
. class Esrc
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="Esrc";     //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;   //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;     //random portion of error
.                               //  R=(1+trackerr+randomerr)*value

.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Esrc::Esrc()
.               default constructor
.               Assigns following defaults:
.                       rawline="EsrcDefault"
.                       linenumber=0
.                       type="V"
.                       name="EsrcDefault"
.                       value=0;
.                       model="EsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
```

```
.                       remainderline="EsrcDefault"
.                       rawline="EsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: Esrc::Esrc(char * xrawline, int xlinenumber)
.                constructor from raw spicefile line
. Function: Esrc::Esrc(char * xname, double xvalue, int xnodeplus,
.               int xnodeneg, int xlinenumber)
.                constructor from data
. Function: Esrc::~Esrc()
.               default destructor
. Function: Esrc& Esrc::operator=(const Esrc & xe)
.               overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Esrc::loadline(char * xrawline,int xlinenumber)
.               loads vsource with data translated from a spice-formatted line
.               Assigns following defaults:
.                       rawline="EsrcDefault"
.                       linenumber=0
.                       type="V"
.                       name="EsrcDefault"
.                       value=0;
.                       model="EsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="EsrcDefault"
.                       rawline="EsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Esrc::writefile(ofstream * xfname )
.               writes a vsource to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void Esrc::print()
.               prints vsource stderr
.               Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Faultlist.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Faultlist.cc is a c++ class for a Faultlist as would correspond to the
.       top-level Faultlist in a spice netlist file.
.       The basic structure is defined in Faultlist.h
.
. class Faultlist
. {
. private:
.
.       char * spicefilename;  //name of Original spicefile
.
.       char * spicefilename;  //Original spicefilename loaded in memory
.
.       Component * pC;                 //pointer to objects corresponding to
.                                   //   various components of Faultlist
.       int numcomponents;   //number of components
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Faultlist::Faultlist()
.               default constructor
.               Assigns following defaults:
.       char * spicefilename="Faultlist Not Loaded";
.       int numcomponents=0;
.       Component * pC=NULL;
. Function: Faultlist::~Faultlist()
.               default destructor
. FUNCTIONS ------------------------------------------------------
. Function: Faultlist::genfaultlist(Circuit & xcirc,
.       double xropen,double xrshort,double xrpara,
.       double xcopen,double xcshort,double xcpara,
.       double xlopen,double xlshort,double xlpara)
.       generate faultlist (no parametric faults)
.               generates only opens and shorts
.               see genparafaultlist to generate a parametric faultlist
.
.               xcirc is circuit without faults
.
```

```
.                   xropen=resistance of open circuit resistors
.                   xrshort=resistance of shorts
.                   xrpara=parametric fault , i.e 0.5 = =/- 50%
. Function: Faultlist::genparafaultlist(Circuit & xcirc,
.         double xropen,double xrshort,double xrpara,
.         double xcopen,double xcshort,double xcpara,
.         double xlopen,double xlshort,double xlpara)
.         generate faultlist (no short/open faults)
.                   generates only parametric faults
.                   see genfaultlist to generate short/opens faultlist
.                   xcirc is circuit without faults
.                   xropen=resistance of open circuit resistors
.                   xrshort=resistance of shorts
.                   xrpara=parametric fault , i.e 0.5 = =/- 50%
. Function: void Faultlist::writefile(char * xfname)
.                   writes a component to the file named xfname
.                   file is opened and closed
. Function: void Faultlist::writefaults(char* dirname, char * xfname, Circuit & ckt )
.                   writes faulty circuits to the files named xfnamefault
.                           in directory dirname
.                   file is opened and closed
. Function: void Faultlist::writerandfaults(char * dirname, char * xfname,
.                                   Circuit & ckt,CircuitStats & cs, int xnr )
.                   writes randomized faulty circuits to the files named xfnamefault
.                           with xnr randomizations per fault
.                           in directory dirname
.                   file is opened and closed
. Function: void Faultlist::print()
.                   prints Faultlist stderr
.                   Usage: a.print();
.         char * Faultlist::getfname(int n)
.                   return faultname of fault n
.         Component Faultlist::getfcomp(int n)
.                   return faulty component for fault n
.         int Faultlist::getnumcomp(int n)
.                   return faulty component number corresponding to fault n
```

```
. ------------------------------------------------------------------------
.
.                                   Gsrc.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Gsrc.cc is a c++ class for controlled-source devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Gsrc.h
. class Gsrc
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="Gsrc";     //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Gsrc::Gsrc()
.               default constructor
.               Assigns following defaults:
.                       rawline="GsrcDefault"
.                       linenumber=0
.                       type="V"
.                       name="GsrcDefault"
.                       value=0;
.                       model="GsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
```

```
.                       remainderline="GsrcDefault"
.                       rawline="GsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: Gsrc::Gsrc(char * xrawline, int xlinenumber)
.            constructor from raw spicefile line
. Function: Gsrc::Gsrc(char * xname, double xvalue, int xnodeplus,
.           int xnodeneg, int xlinenumber)
.            constructor from data
. Function: Gsrc::~Gsrc()
.           default destructor
. Function: Gsrc& Gsrc::operator=(const Gsrc & xe)
.           overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Gsrc::loadline(char * xrawline,int xlinenumber)
.           loads vsource with data translated from a spice-formatted line
.           Assigns following defaults:
.                   rawline="GsrcDefault"
.                   linenumber=0
.                   type="V"
.                   name="GsrcDefault"
.                   value=0;
.                   model="GsrcDefault"
.                   numnodes=2
.                   nodelist=0 0
.                   remainderline="GsrcDefault"
.                   rawline="GsrcDefault"
.                   trackerr=0
.                   randomerr=0
. Function: void Gsrc::writefile(ofstream * xfname )
.           writes a vsource to the file handle xfname
.           file is assumed to already be opened
.           file is not closed
. Function: void Gsrc::print()
.           prints vsource stderr
.           Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Inductor.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Inductor.cc is a c++ clas for inductor devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Inductor.h
. class Inductor
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="L";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ---------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS --------------------------------------------------
. Function: Inductor::Inductor()
.               default constructor
.               Assigns following defaults:
.                       rawline="Ldefault"
.                       linenumber=0
.                       type="L"
.                       name="Ldefault"
.                       value=0;
.                       model="Ldefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Ldefault"
```

```
.                       rawline="Ldefault"
.                       trackerr=0
.                       randomerr=0
. Function: Inductor::Inductor(char * xrawline, int xlinenumber)
.                  constructor from raw spicefile line
. Function: Inductor::Inductor(char * xname, double xvalue, int xnodeplus,
.                  int xnodeneg, int xlinenumber)
.                  constructor from data
. Function: Inductor::~Inductor()
.                  default destructor
. Function: Inductor& Inductor::operator=(const Inductor & r)
.                  overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Inductor::loadline(char * xrawline,int xlinenumber)
.                  loads inductor with data translated from a spice-formatted line
.                  Assigns following defaults:
.                       rawline="Ldefault"
.                       linenumber=0
.                       type="L"
.                       name="Ldefault"
.                       value=0;
.                       model="Ldefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Ldefault"
.                       rawline="Ldefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Inductor::writefile(ofstream * xfname )
.                  writes a inductor to the file handle xfname
.                  file is assumed to already be opened
.                  file is not closed
. Function: void Inductor::writefile(ofstream * xfname , CircuitStats & cs )
.                  writes a randomized inductor to the file handle xfname
.                  file is assumed to already be opened
.                  file is not closed
. Function: void Inductor::print()
.                  prints inductor stderr
.                  Usage: a.print();
. Function: void Inductor::setvalue(double xvalue)
.                  sets inductor value
. Function: void Inductor::scalevalue(double xscale)
.                  sets inductor value to value times xscale
. Function: char * Inductor::getname()
.                  gets inductor name
```

```
. ------------------------------------------------------------------------
.
.                                   Isrc.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Isrc.cc is a c++ clas for source devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Isrc.h
.
. class Isrc
. {
.
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="I";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Isrc::Isrc()
.               default constructor
.               Assigns following defaults:
.                       rawline="IsrcDefault"
.                       linenumber=0
.                       type="I"
.                       name="IsrcDefault"
.                       value=0;
.                       model="IsrcDefault"
```

```
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="IsrcDefault"
.                       rawline="IsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: Isrc::Isrc(char * xrawline, int xlinenumber)
.                  constructor from raw spicefile line
. Function: Isrc::Isrc(char * xname, double xvalue, int xnodeplus,
.                int xnodeneg, int xlinenumber)
.                  constructor from data
. Function: Isrc::~Isrc()
.                default destructor
. Function: Isrc& Isrc::operator=(const Isrc & vdd)
.                overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Isrc::loadline(char * xrawline,int xlinenumber)
.                loads vdd with data translated from a spice-formatted line
.                Assigns following defaults:
.                       rawline="IsrcDefault"
.                       linenumber=0
/.                      type="I"
.                       name="IsrcDefault"
.                       value=0;
.                       model="IsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="IsrcDefault"
.                       rawline="IsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Isrc::writefile(ofstream * xfname )
.                writes a vdd to the file handle xfname
.                file is assumed to already be opened
.                file is not closed
. Function: void Isrc::print()
.                prints vdd stderr
.                Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Mos.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Mos.cc is a c++ clas for mos devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Mos.h
. class Mos
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="M";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.       char * fault;           //fault used to print extra resistor
.       int faultflag;          //=0 if no fault
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
.
.
.
. CONSTRUCTORS ---------------------------------------------------
.
.
. Function: Mos::Mos()
.               default constructor
.               Assigns following defaults:
.                       rawline="Mdefault"
```

```
.                       linenumber=0
.                       type="M"
.                       name="Mdefault"
.                       value=0;
.                       model="Mdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Mdefault"
.                       rawline="Mdefault"
.                       trackerr=0
.                       randomerr=0
.                       rfault="Mdefault"
.                       int faultflag=0 = no fault
.
.
. Function: Mos::Mos(char * xrawline, int xlinenumber)
.                        constructor from raw spicefile line
. Function: Mos::Mos(char * xname, double xvalue, int xnodeplus,
.               int xnodeneg, int xlinenumber)
.                        constructor from data
. Function: Mos::~Mos()
.               default destructor
. Function: Mos& Mos::operator=(const Mos & mos)
.               overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Mos::loadline(char * xrawline,int xlinenumber)
.               loads mos with data translated from a spice-formatted line
.               Assigns following defaults:
.                       rawline="Mdefault"
.                       linenumber=0
/.                      type="M"
.                       name="Mdefault"
.                       value=0;
.                       model="Mdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Mdefault"
.                       rawline="Mdefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Mos::writefile(ofstream * xfname )
.               writes a mos to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void Mos::print()
.               prints mos stderr
```

```
.                   Usage: a.print();
. Function: void Mos::faultdrainopen(double xvalue)
.                   set drain to have series resistor of value xvalue
.                           typically used to open-circuit a fet
. Function: void Mos:: faultdrainsourceshort(double xvalue)
.                   set drain -source to have shunt resistor of value xvalue
.                           typically used to short-circuit a fet
. Function: char * Mos::getname()
.                   gets Mos name
```

```
. ------------------------------------------------------------------------
.
.                                   Ora.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Ora.cc is a c++ class for generating input signals
.       The basic structure is defined in Ora.h
. class Ora
. {
. private:
.       char * spicefile;       //name of spicefile
.       char * tpgfile;         //name of tpg file
.       char * chifile;         //name of chifile (spice output file)
.       double vomax;           //range of differential vout
.       double vomin;           //  assumed equal to range
.                               //  of a/d converter at output
.       double vbias;           //dc bias at input
.                               //if bias=0, assume floating inputs
.                               //else inneg=constant vbias
.                               //   with inpos=vbias+/- ampl/2
.       double ampl;            //amplitude of differential input
.                               //ranges 0-ampl if no bias
.       double vinscale;        //scale factor applied to differential
.       double voutscale;       //  in/out voltages
.                               //typically, voscale=1 and viscale is
.                               //  adjusted so vin has same volt swing
.                               //  as vout
.       double voutsum;                 //ora sum of Vout
.       double vdiffsum;        //ora sum of Vout - Vin
.       double vmagsum;                 //ora sum of |Vout - Vin|
.       double clkthresh;       //clock threshold (half-voltage)
.                               //   assumed 2.5 volts(see Tpg.cpp)
.       int errflag;            //error flag
.       char * errmsg;          //error message
.
.       int nclk;               //column number of clock data in chi file
.       int nvinplus,nvinminus; //column numbers of vin data in chi file
.       int nvoutplus,nvoutminus; //column number of vout data in chi file
.         waveforms:
.                 cup, cdwn, cud, cuR, cdR, para, paraR, pulse,
.                 cudR, const, lfsr, fswp, fswpR, fswpC, fswpRC\n");
.        clock freq  in Hz, KHz (K), MHz (M), GHz (G)
.        amplitude in Volts (integer)
.        output format: sp (SPICE) ex (EXCEL) cs (CSIM)\n");
```

```
.           repnum is number of repetitions of waveform or SR bits for freq_sweep
.           poly is an integer 0-127 giving inner coefficients of poly
. CONSTRUCTORS ----------------------------------------------------
. Function: Ora::Ora()
.                   default constructor
.                   Assigns following defaults:
. Function: Ora::~Ora()
.                   default destructor
. Function: int Ora::genora(char * spicefile,
.                   char * tpgfile,  char * chifile
.                   double xvbias, double xampl,
.                   double xvomin, double xvomax  )
.                   - generate ora from spice (.cir), tpg (.tpg),
.                     and spice output (.chi) files
.                   - the chi file is a spice output file
.                   - xvbias and xampl are dc input bias and
.                             amplitude of input voltage
.                             if bias=0, assume floating inputs,
.                              else neg innode=constant vbias
.                               with pos=vbias+/- ampl/2
.                             amplitude of differential input
.                              ranges 0-ampl if no bias
.                   Return: 1=bad ora, 0=good
.                   Usage: a.createora();
. Function: unsigned int Ora::analog2digital(double xanaval)
.                   converts analog value to digital value: digital value = analog/0.019608
.                   Usage: ;
.           This function could be use in the future
. Function: unsigned int Ora::decimal2binary_V2(unsigned int xdecval)
.                   converts decimal to binary
.                   Usage: ;
. Function: void Ora::writeorafile(char * xorafile)
.                   writes ora data/results to ora file
.                   Usage: ;
. Function: void Ora::writeoraerrfile(char * xoraerrfile)
.                   writes ora error message to ora error file
.                   Usage: ;
. Function: void Ora::print()
.                   prints Ora stderr
.                   Usage: a.print();
.
. Function: int Ora::findcolumns( )
.                   - find column locations in spice output data file
.                   -   to locate
.                     clock, vopos, voneg, vineg, vipos
.                   - set corresponding class members
```

```
.                   if bias=0, assume floating inputs,
.           Return: 1=fail, 0=succeed
.           Usage: a.createora();
```

```
. ------------------------------------------------------------------------
.
.                                   Other.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Other.cc is a c++ clas for Other lines as would be found in
.       a spice netlist file.
.       The basic structure is defined in Other.h
. class Other
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber;                 //linenumber in original spicefile
.       char * type;            //device type, i="Other"
.       char * name;            //="Other"
.       double value;           //=0
.       char * model;           //="Other"
.       int numnodes;           //=0
.       int nodelist[2];        //= 0 0
.                               //
.       char * remainderline;   //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value
.       double trackerr;        //=0
.       double randomerr;       //=0
.                               //
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Other::Other()
.               default constructor
.               Assigns following defaults:
. Function: Other::Other(char * xtext,int xlinenumber)
.
.                constructor from data
. Function: Other::~Other()
.               default destructor
. Function: Other& operator=(const Other& com)
.               overloaded equal
. FUNCTIONS ------------------------------------------------------
```

```
. Function: Other::loadline(char * xrawline,int xlinenumber)
.                   loads Other with data translated from a spice-formatted line
. Function: void Other::writefile(ofstream * xfname )
.                   writes a Other to the file handle xfname
.                   file is assumed to already be opened
.                   file is not closed
. Function: void Other::print()
.                   prints Other stderr
.                   Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Resistor.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Resistor.cc is a c++ clas for resistor devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Resistor.h
. class Resistor
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="R";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Resistor::Resistor()
.               default constructor
.               Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
.                       type="R"
.                       name="Rdefault"
.                       value=0;
.                       model="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Rdefault"
```

```
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
. Function: Resistor::Resistor(char * xrawline, int xlinenumber)
.                  constructor from raw spicefile line
. Function: Resistor::Resistor(char * xname, double xvalue, int xnodeplus,
.              int xnodeneg, int xlinenumber)
.                  constructor from data
. Function: Resistor::~Resistor()
.              default destructor
. Function: Resistor& Resistor::operator=(const Resistor & r)
.              overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Resistor::loadline(char * xrawline,int xlinenumber)
.              loads resistor with data translated from a spice-formatted line
.              Assigns following defaults:
.                       rawline="Rdefault"
.                       linenumber=0
.                       type="R"
.                       name="Rdefault"
.                       value=0;
.                       model="Rdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Rdefault"
.                       rawline="Rdefault"
.                       trackerr=0
.                       randomerr=0
.
. Function: void Resistor::writefile(ofstream * xfname )
.              writes a resistor to the file handle xfname
.              file is assumed to already be opened
.              file is not closed
. Function: void Resistor::writefile(ofstream * xfname , CircuitStats & cs )
.              writes a randomized resistor to the file handle xfname
.              file is assumed to already be opened
.              file is not closed
. Function: void Resistor::print()
.              prints resistor stderr
.              Usage: a.print();
. Function: void Resistor::setvalue(double xvalue)
.              sets resistor value
. Function: void Resistor::scalevalue(double xscale)
.              sets resistor value to value times xscale
. Function: char * Resistor::getname()
.              gets resistor name
```

```
. ------------------------------------------------------------------------
.
.                                   Statistics.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Statistics.cc is a c++ class for statistical functions
.
.       The basic structure is defined in Statistics.h
. class Statistics
. {
. private:
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Statistics::Statistics()
.               default constructor
.               Assigns following defaults:
.                       comment="Default uniform"
.                       pdf1="uniform"
.                       pdf2="disabled"
.                       mean1=0;
.                       sigma1=1;
.                       mean2=0;
.                       sigma2=0;
. Function: Statistics::Statistics(char * xcomment, char * xpdf1,
.       double xmean1, double xsigma1)
.                constructor from data
. Function: Statistics::~Statistics()
.               default destructor
. Function: Statistics & Statistics::operator=(const Statistics & stat)
.               overloaded equal
. FUNCTIONS ------------------------------------------------------
. Function: double Statistics::genrand( )
.               returns a random number
. Function: Statistics::setstats(char * xcomment, char * xpdf1,
.       double xmean1, double xsigma1)
.                set statistics
. Function: Statistics::setmean( double xmean1)
.               set mean=xmean1,
. Function: Statistics::setstatstol(char * xcomment, char * xpdf1,
.       double xmean1, double tolerance)
.                set statistics
```

```
.                   set mean=xmean1,
.                   sigma=stddev=tolerance/2.5 for gauss pdf
.                           i.e., +/- 2.5 std deviations=tolerance
.                           i.e., a 5% tolerance, tolerance=0.05, stdev=0.02
.                   for uniform pdf, sigma=tolerance*2/3.4641
. Function: void Statistics::writefile(ofstream * xfname )
.                   writes a statistics object to the file handle xfname
.                   it is written as a spicefile comment
. Function: void Statistics::print()
.                   prints resistor stderr
.                   Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Tpg.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Tpg.cc is a c++ class for generating input signals
.          The basic structure is defined in Tpg.h
. class Tpg
. {
.
. private:
.
.
.          char * clkfreq;          //clock frequency
.          char * amplitude;        //waveform amplitude
.          char * waveform;                 //waveform (square/ramp/parabola, etc)
.          char * format;           //file format of output, sp =spice
.          char * filename;                 //name of output file
.          char * poly;             //lfsr polynomial
.          char * repnum;                   //number of repetitions of waveform
.          //the following two are automatically created from above members
.          int argc;                //these items argc and argv mimmick
.          char ** argv;            //the behavior of cestroud's anatpg.c
.
.          int printdig(int xval, int xdig);//used internally only
.            waveforms:
.                    cup, cdwn, cud, cuR, cdR, para, paraR, pulse,
.                    cudR, const, lfsr, fswp, fswpR, fswpC, fswpRC\n");
.           clock freq  in Hz, KHz (K), MHz (M), GHz (G)
.           amplitude in Volts (integer)
.           output format: sp (SPICE) ex (EXCEL) cs (CSIM)\n");
.           repnum is number of repetitions of waveform or SR bits for freq_sweep
.           poly is an integer 0-127 giving inner coefficients of poly
. CONSTRUCTORS ------------------------------------------------------
. Function: Tpg::Tpg()
.                    default constructor
.                    Assigns following defaults:
. Function: void Tpg::createtpg(char * filename ,char *  waveform , char * clkfreq ,
.                              char * amplitude , char * format ,
.                              char * repnum , char * poly  )
.                    gcreates input signals
.                    Usage: a.createtpg();
. Function: Tpg::~Tpg()
.                    default destructor
```

```
. Function: void Tpg::mergefiles(char * xtpgfile,
.                           char * xcirfile, char * xmergedfile)
.                    merges tpg  file and circuit file into output file
.                           .end statement is placed at end of final output file
.                   Usage:
. Function: void Tpg::Tpg::writefile()
.                    writes tpg to file
.                   (output file name is contained in filename class member)
.                   Usage:
. Function: void Tpg::writefulltpg(
.                   char * xinposnode, char * xinnegnode,
.                   char * xoutposnode, char * xoutnegnode,
.                   char * xvbias  )
.                   writes tpg to file contained in "filename" member
.                   includes lines for tapping into circuit for input nodes
.                           and output nodes (disabled if any argument is NULL)
.                           also, vbias="" disables DC bias of input and you
.                           get a true floating differential input
.                      - xinposnode, xinnegnode: pos and neg input nodes
.                           for differential input voltage source
.                      - xoutposnode, xoutnegnode: pos and neg output nodes
.                      - vxbias: dc bias value where posnode input ranges from
.                           xvbias+(vin/2) to xvbias-(vin/2) and
.                           innegnode is set to vbias
.                   (output file name is contained in filename class member)
.                   Usage:
. Function: int TPG::printdig(int val, int dig)
.                   writes the line components for csim file format
.                   Usage: a.writetpg();
. Function: void Tpg::getterminput()
.                   prompts user for terminal input
.                   Usage:
. Function: void Tpg::print()
.                   prints Tpg stderr
.                   Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Vsrc.cpp
.
.                                   Rev 1.0
.
. ------------------------------------------------------------------------
. Vsrc.cc is a c++ clas for source devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Vsrc.h
. class Vsrc
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="V";        //device type, i.e., R, L, C, V, M
.       char * name;            //instance name, i.e., R1, R2, etc
.       double value;           //resistance value
.       char * model=" ";       //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for R, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  R=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
. FLAGS ----------------------------------------------------------
.       Check the header file for any useful debug flags
. CONSTRUCTORS ---------------------------------------------------
. Function: Vsrc::Vsrc()
.               default constructor
.               Assigns following defaults:
.                       rawline="VsrcDefault"
.                       linenumber=0
.                       type="V"
.                       name="VsrcDefault"
.                       value=0;
.                       model="VsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="VsrcDefault"
```

```
.                       rawline="VsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: Vsrc::Vsrc(char * xrawline, int xlinenumber)
.                  constructor from raw spicefile line
. Function: Vsrc::Vsrc(char * xname, double xvalue, int xnodeplus,
.                  int xnodeneg, int xlinenumber)
.                  constructor from data
. Function: Vsrc::~Vsrc()
.                  default destructor
. Function: Vsrc& Vsrc::operator=(const Vsrc & vdd)
.                  overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Vsrc::loadline(char * xrawline,int xlinenumber)
.                  loads vdd with data translated from a spice-formatted line
.                  Assigns following defaults:
.                       rawline="VsrcDefault"
.                       linenumber=0
/.                      type="V"
.                       name="VsrcDefault"
.                       value=0;
.                       model="VsrcDefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="VsrcDefault"
.                       rawline="VsrcDefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Vsrc::writefile(ofstream * xfname )
.                  writes a vdd to the file handle xfname
.
.                  file is assumed to already be opened
.                  file is not closed
. Function: void Vsrc::print()
.                  prints vdd stderr
.                  Usage: a.print();
```

```
. ------------------------------------------------------------------------
.
.                                   Xsubckt.cpp
.
.                                   Xsubcktev 1.0
.
. ------------------------------------------------------------------------
. Xsubckt.cc is a c++ clas for Xsubckt devices as would be found in
.       a spice netlist file.
.       The basic structure is defined in Xsubckt.h
. class Xsubckt
. {
. private:
.       char * rawline;         //raw spicefile line as read in from file
.       int linenumber=0;       //linenumber in original spicefile
.       char * type="Xsubckt";          //device type, i.e., Xsubckt, L, C, V, M
.       char * name;            //instance name, i.e., Xsubckt1, Xsubckt2, etc
.       //double value;         //resistance value
.       //char * model=" ";     //optional device model name
.       int numnodes=2;                 //number of nodes/pins the device has
.       int nodelist[2];        //ordered list of node numbers for device
.                               //  for Xsubckt, nodelist is +node, -node
.       char * remainderline;  //remainder of raw spice-file line contents
.                               //  as contained in rawline,
.                               //  after stripping off name, model,
.                               //  nodelist and value (first line only)
.       double trackerr=0;      //tracking portion of error factor
.       double randomerr=0;  //random portion of error
.                               //  Xsubckt=(1+trackerr+randomerr)*value
.  A wide class of operators is provided, and generally memory is
.       allocated and deallocated automatically.
.
.
. FLAGS ----------------------------------------------------------
.
.       Check the header file for any useful debug flags
. CONSTXsubcktUCTOXsubcktS ---------------------------------------------------
. Function: Xsubckt::Xsubckt()
.               default constructor
.               Assigns following defaults:
.                       rawline="Xsubcktdefault"
.                       linenumber=0
.                       type="Xsubckt"
.                       name="Xsubcktdefault"
.                       //value=0;
.                       //model="Xsubcktdefault"
```

```
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Xsubcktdefault"
.                       rawline="Xsubcktdefault"
.                       trackerr=0
.                       randomerr=0
. Function: Xsubckt::Xsubckt(char * xrawline, int xlinenumber)
.                constructor from raw spicefile line
. Function: Xsubckt::Xsubckt(char * xname,  int xnodeplus,
.               int xnodeneg, int xlinenumber)
.                constructor from data
. Function: Xsubckt::~Xsubckt()
.               default destructor
. Function: Xsubckt& Xsubckt::operator=(const Xsubckt & r)
.               overloaded equal
. FUNCTIONS ----------------------------------------------------
. Function: Xsubckt::loadline(char * xrawline,int xlinenumber)
.               loads Xsubckt with data translated from a spice-formatted line
.               Assigns following defaults:
.                       rawline="Xsubcktdefault"
.                       linenumber=0
.                       type="Xsubckt"
.                       name="Xsubcktdefault"
.                       value=0;
.                       model="Xsubcktdefault"
.                       numnodes=2
.                       nodelist=0 0
.                       remainderline="Xsubcktdefault"
.                       rawline="Xsubcktdefault"
.                       trackerr=0
.                       randomerr=0
. Function: void Xsubckt::writefile(ofstream * xfname )
.               writes a Xsubckt to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void Xsubckt::writefile(ofstream * xfname , CircuitStats & cs )
.               writes a randomized Xsubckt to the file handle xfname
.               file is assumed to already be opened
.               file is not closed
. Function: void Xsubckt::print()
.               prints Xsubckt stderr
.               Usage: a.print();
```

APPENDIX F: S_{16OUT} ORA METRIC FAULT METRIC DATA

The histograms of Figs. F.1 to F.12 are histograms showing individual faults and fault free circuits, for the ORA metric $S_{16out,}$ for simulations of the circuit of Fig. 3.3. The histograms of F.1 to F.12 comprise the composite histogram of Fig. 3.14. The histograms of Figs. F.13 to F.24 are histograms showing individual faults and fault free circuits, for the ORA metric $S_{16mag,}$ for simulations of the circuit of Fig. 3.3. The histograms of F.13 to F.24 comprise the composite histogram of Fig. 3.16.

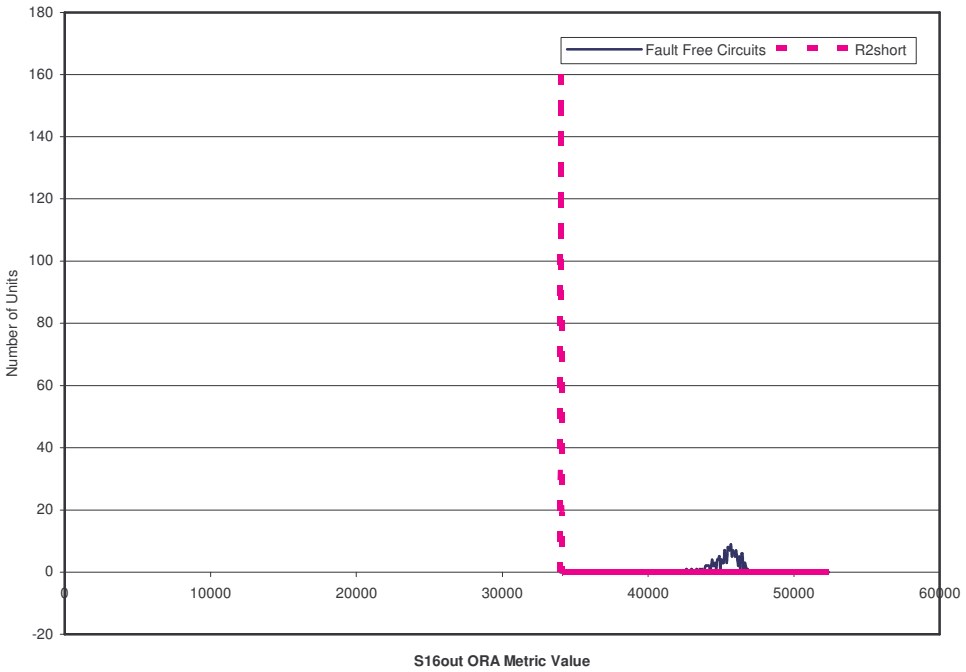


Figure F.1: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R2short(dotted) and fault-free(solid) circuits.

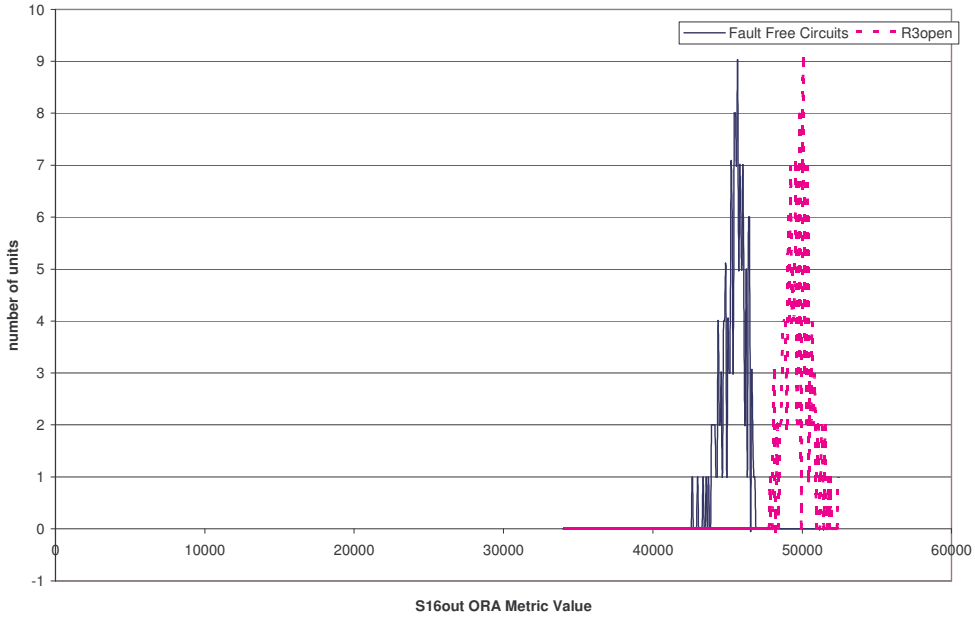


Figure F.2: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R3open(dotted) and fault-free(solid) circuits.

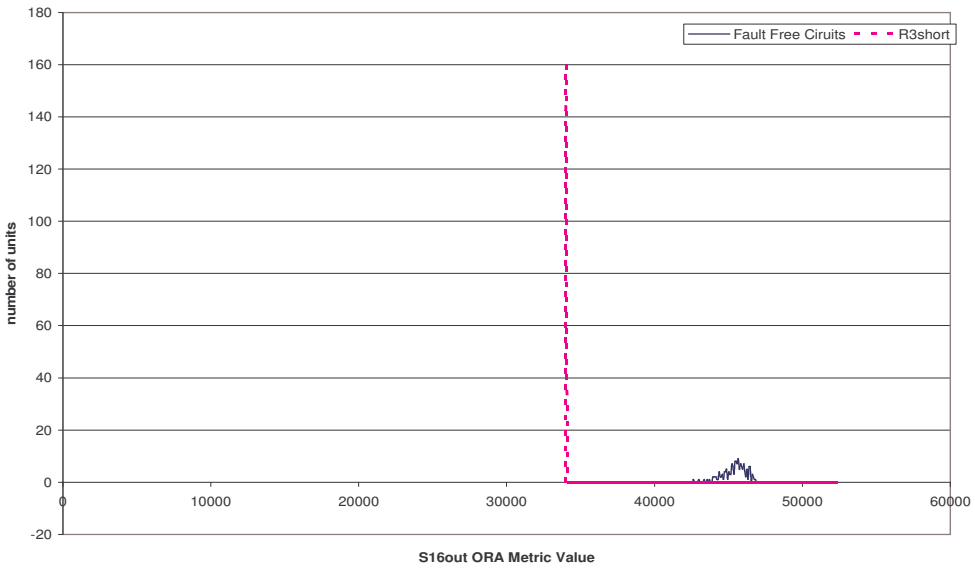


Figure F.3: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R3short(dotted) and fault-free(solid) circuits.

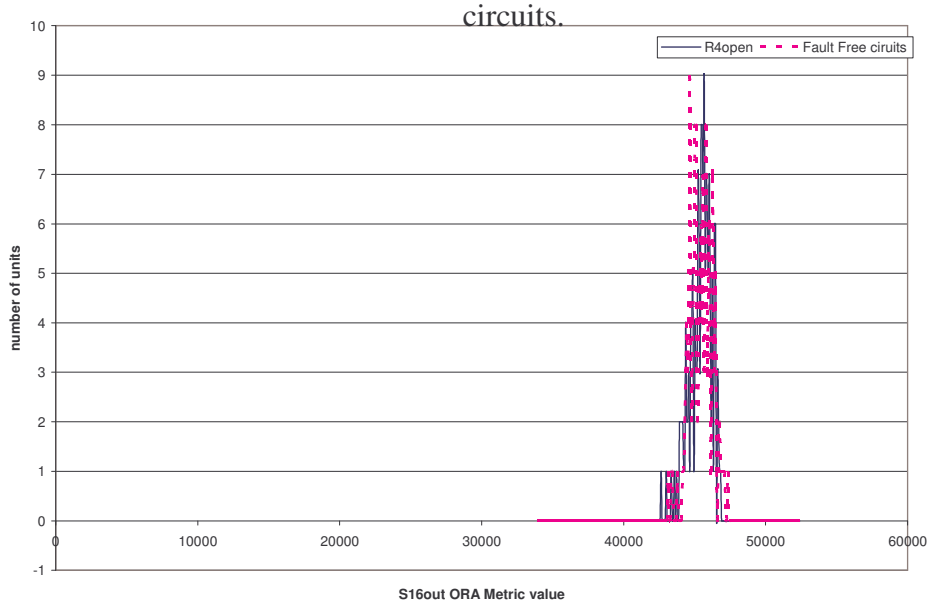


Figure F.4Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R4open(dotted) and fault-free(solid) circuits.

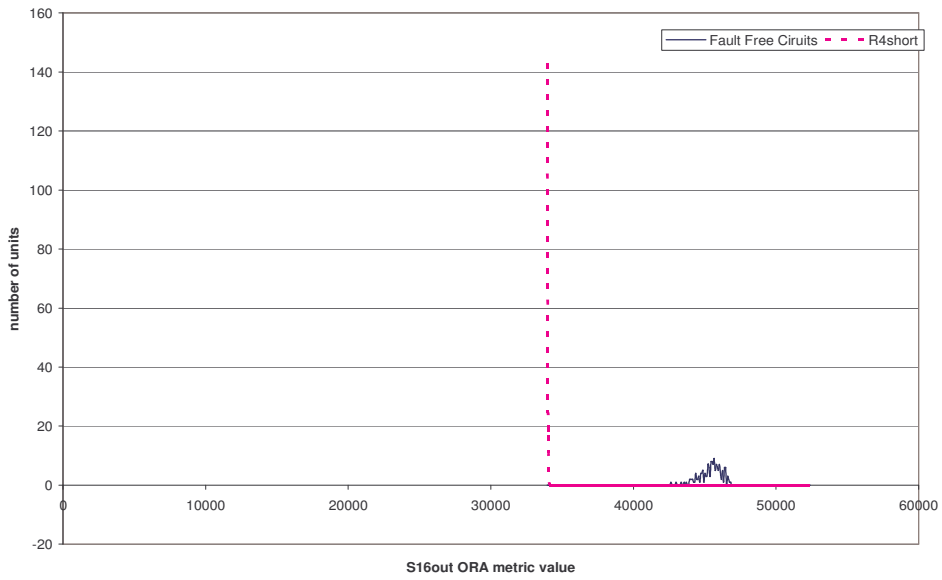


Figure F.5: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R4short(dotted) and fault-free(solid) circuits.

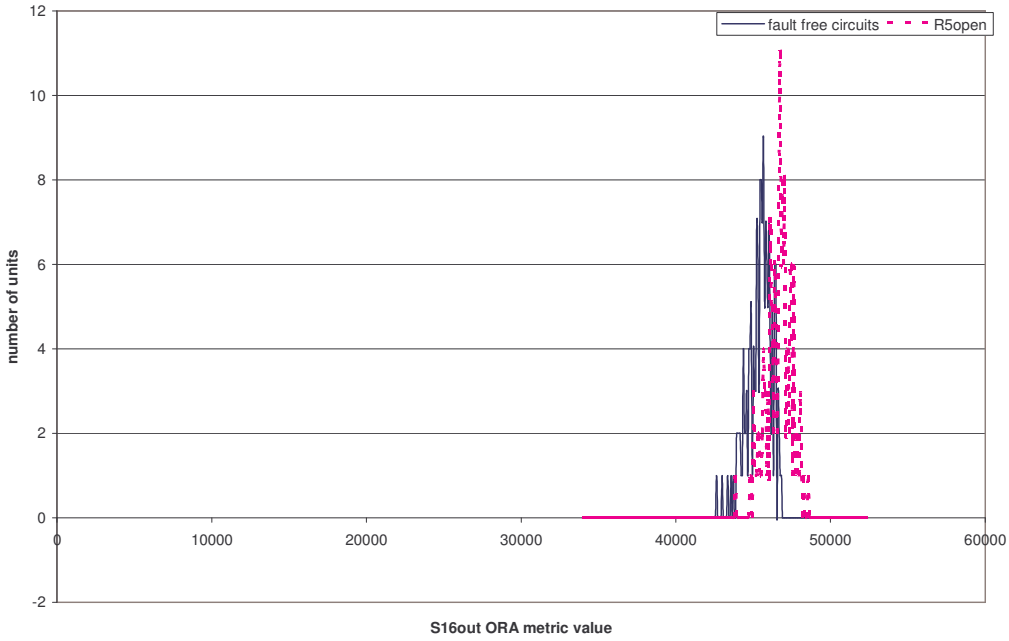


Figure F.6: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R5open(dotted) and fault-free(solid) circuits.

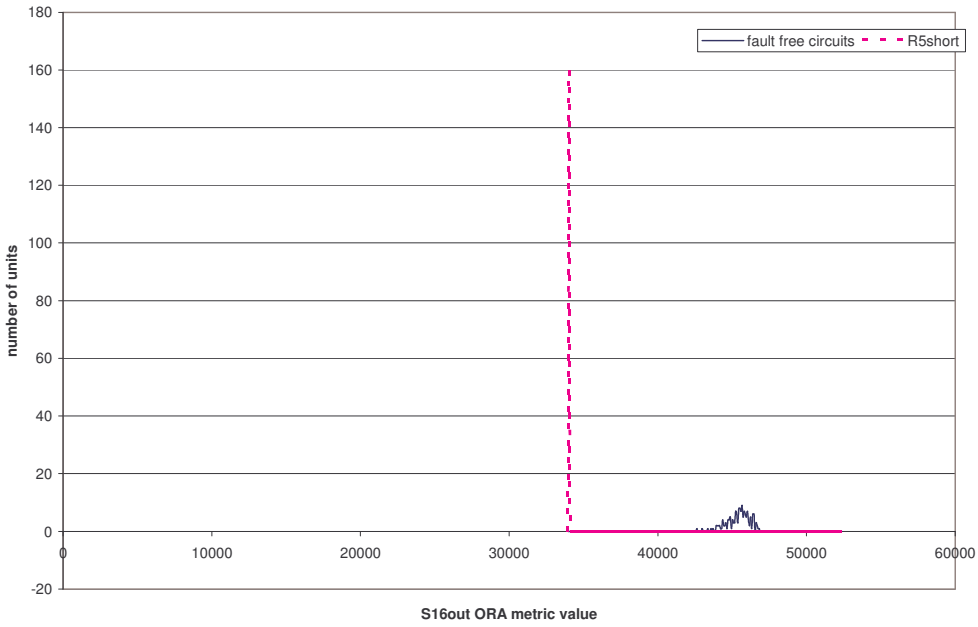


Figure F.7: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R5short(dotted) and fault-free(solid) circuits.

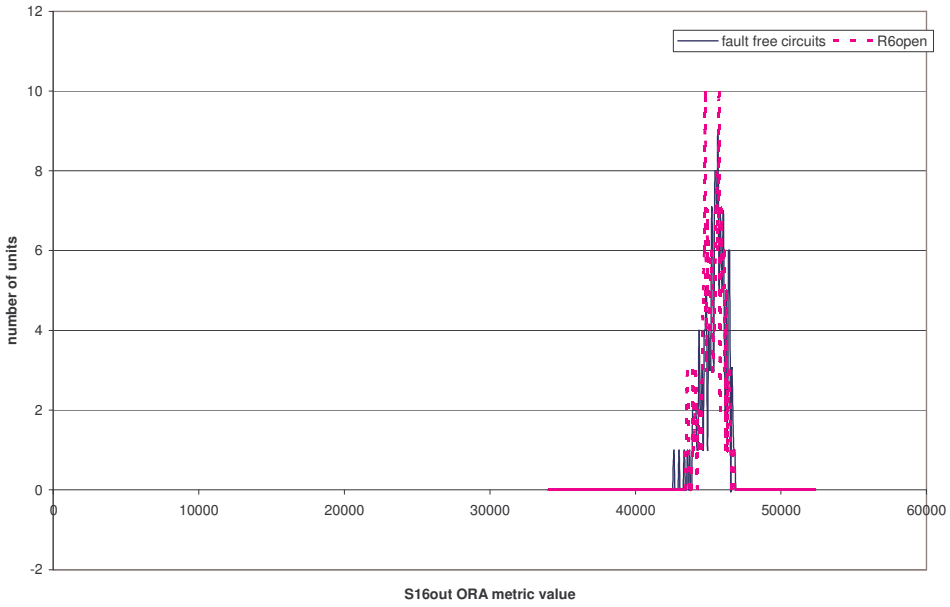


Figure F.8: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R6open(dotted) and fault-free(solid) circuits.

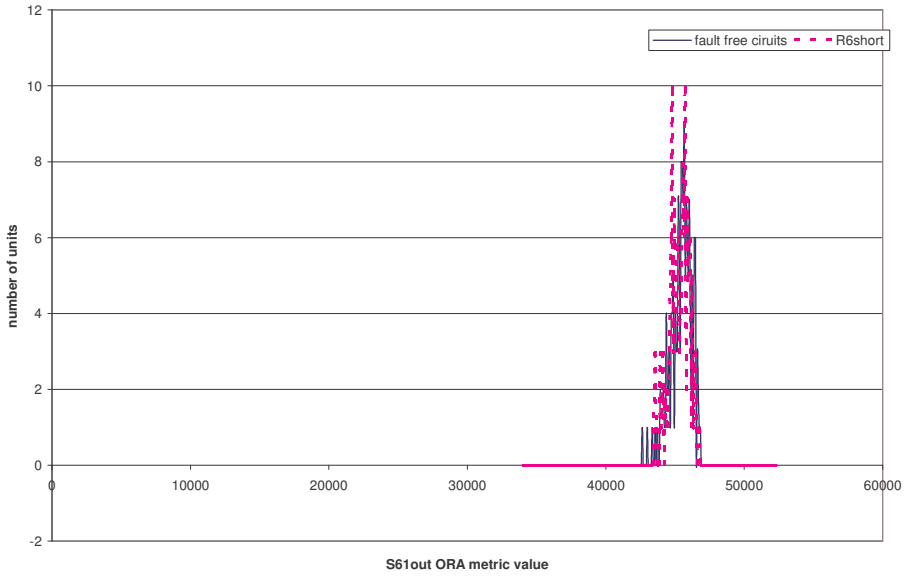


Figure F.9: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R6short(dotted) and fault-free(solid) circuits.

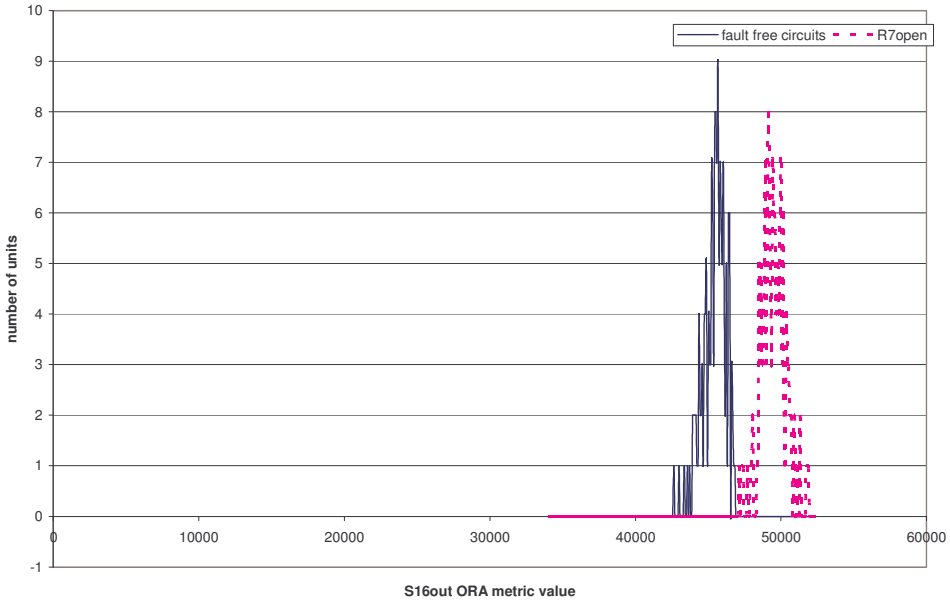


Figure F.10: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R7open(dotted) and fault-free(solid) circuits.

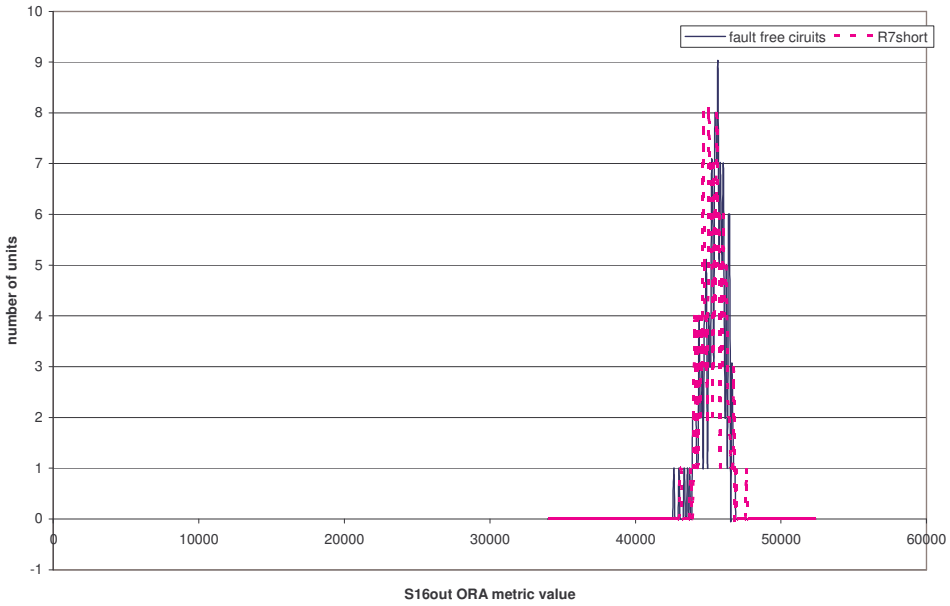


Figure F.11: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R7short(dotted) and fault-free(solid) circuits.

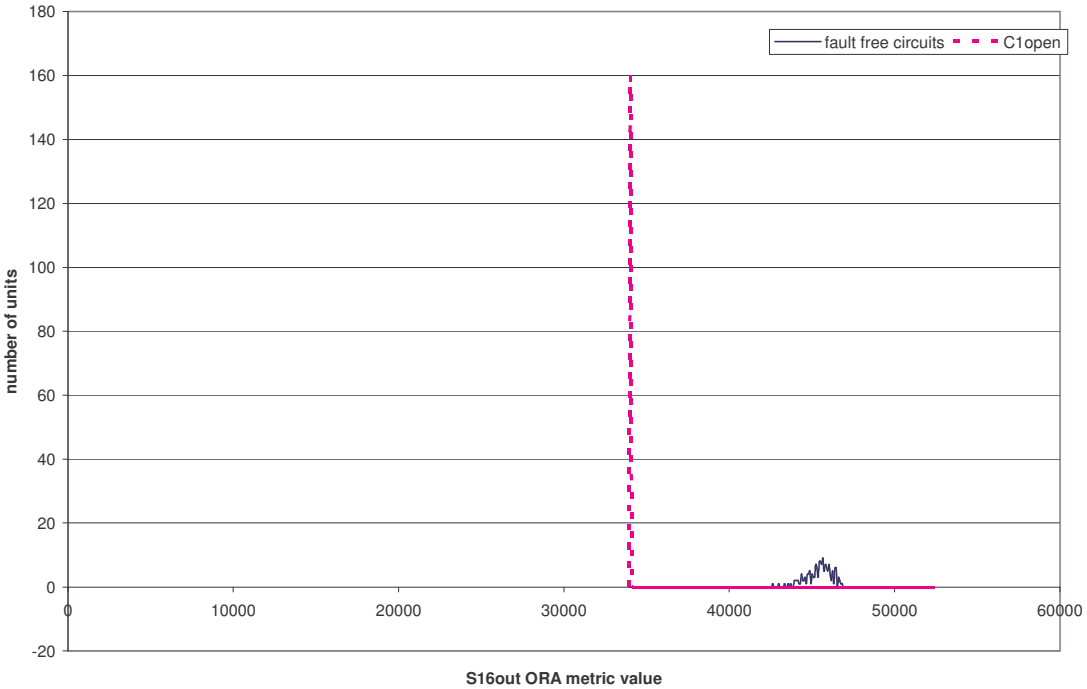


Figure F.12: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are C1open(dotted) and fault-free(solid) circuits.

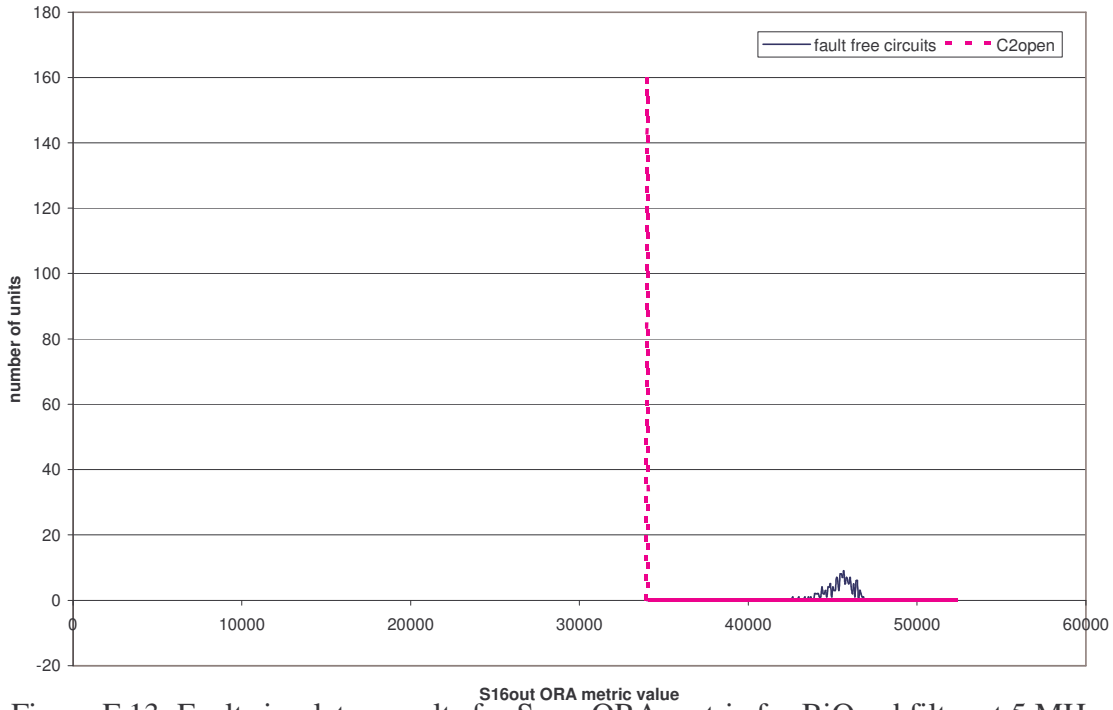


Figure F.13: Fault simulator results for S_{16out} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are C2open(dotted) and fault-free(solid) circuits.

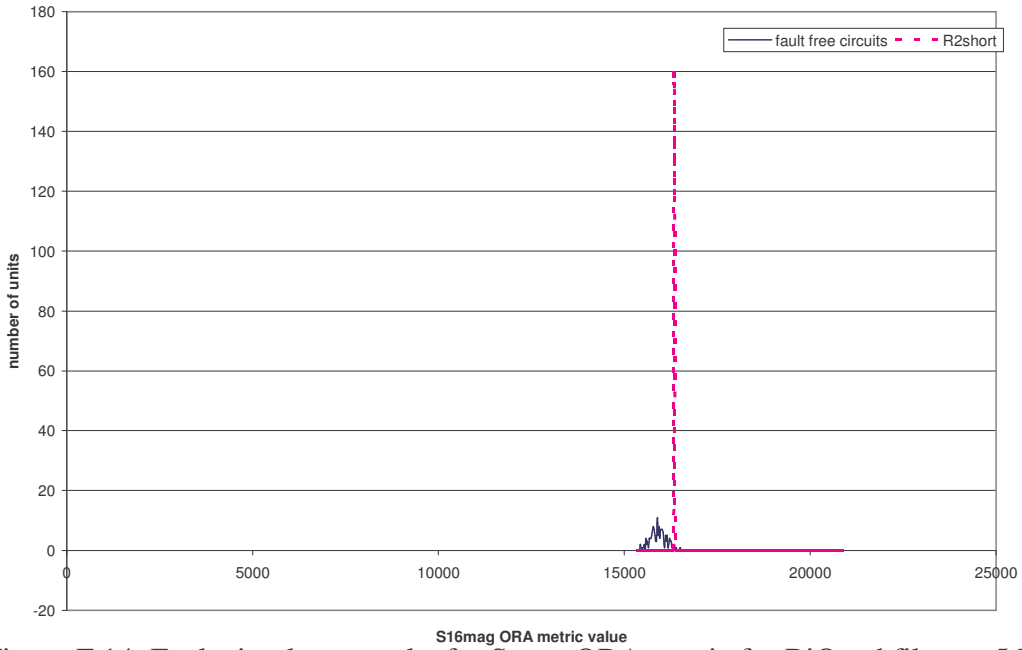


Figure F.14: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R2short(dotted) and fault-free(solid) circuits.

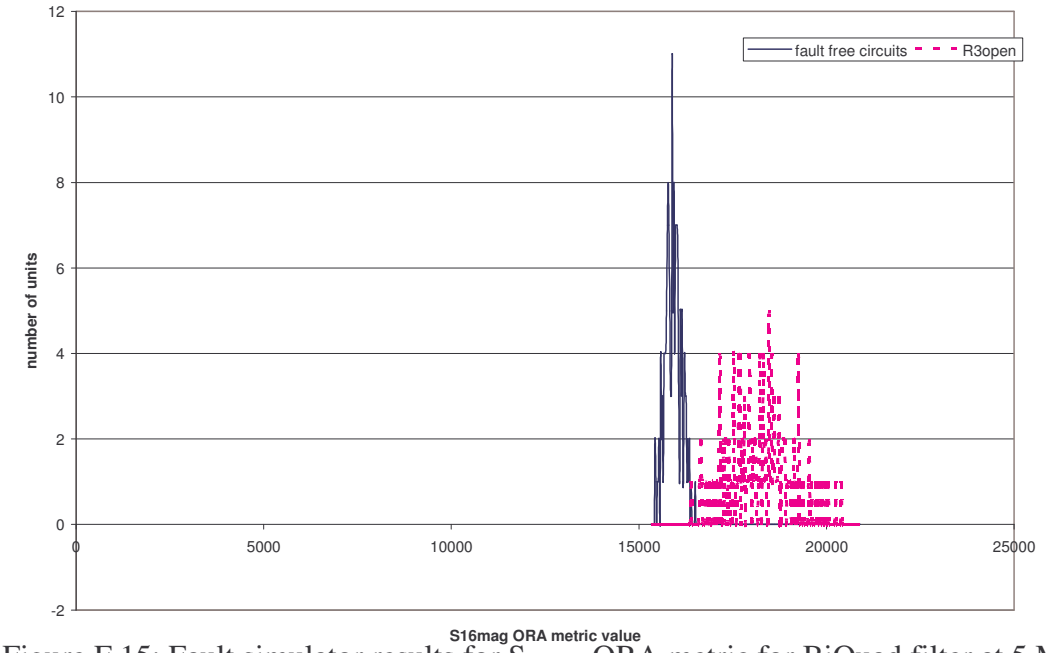


Figure F.15: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R3open(dotted) and fault-free(solid) circuits.

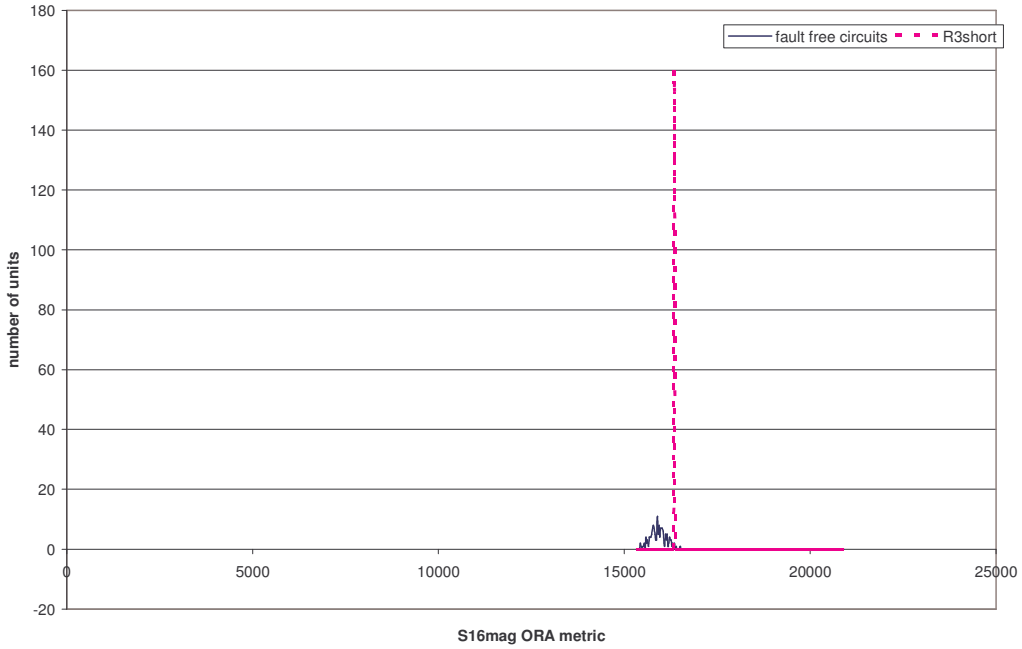


Figure F.16: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R3short(dotted) and fault-free(solid) circuits.

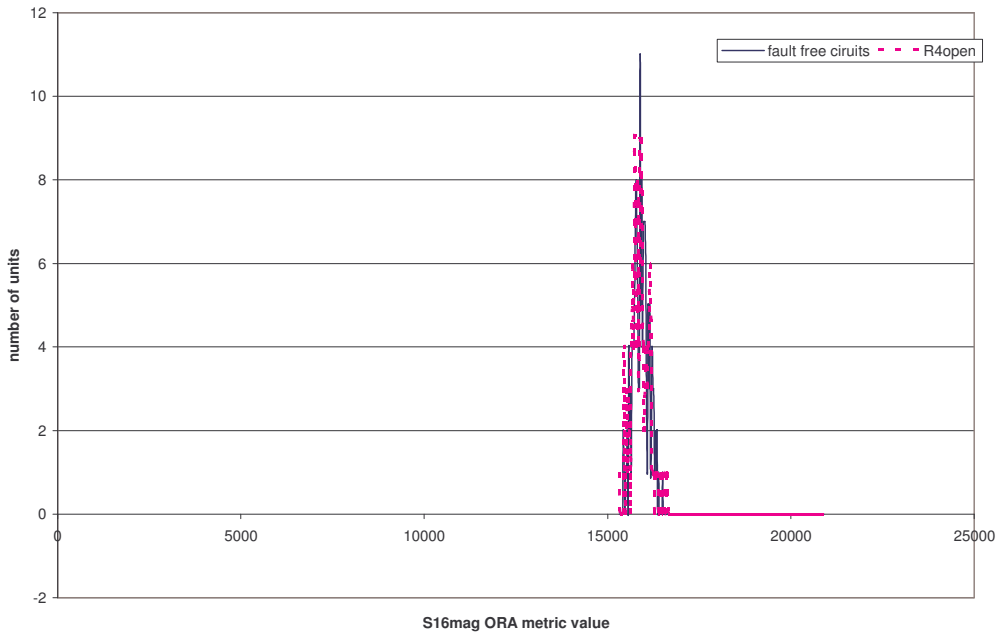


Figure F.17: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R4open(dotted) and fault-free(solid) circuits.

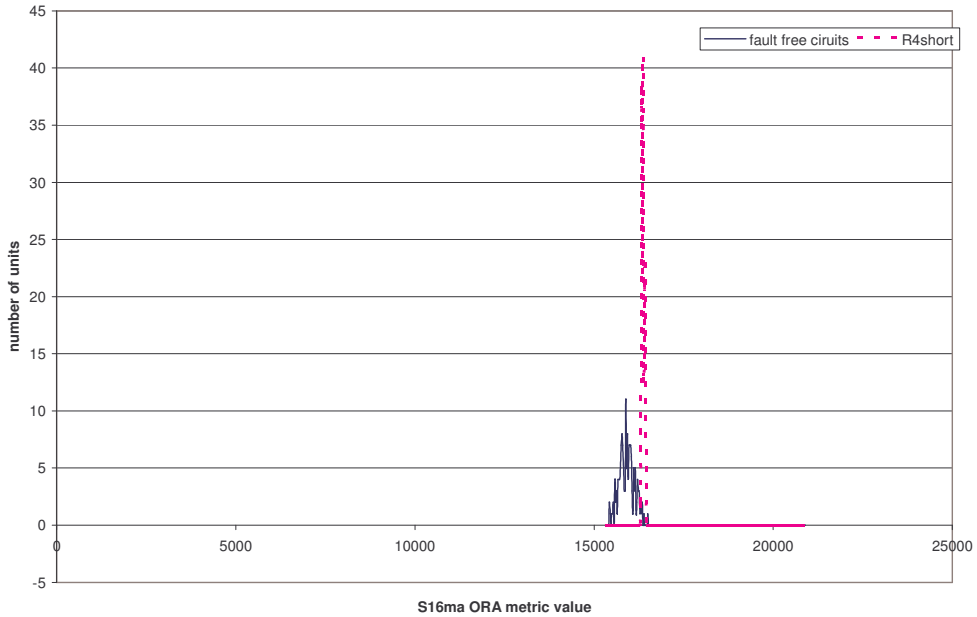


Figure F.18: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R4short(dotted) and fault-free(solid) circuits.

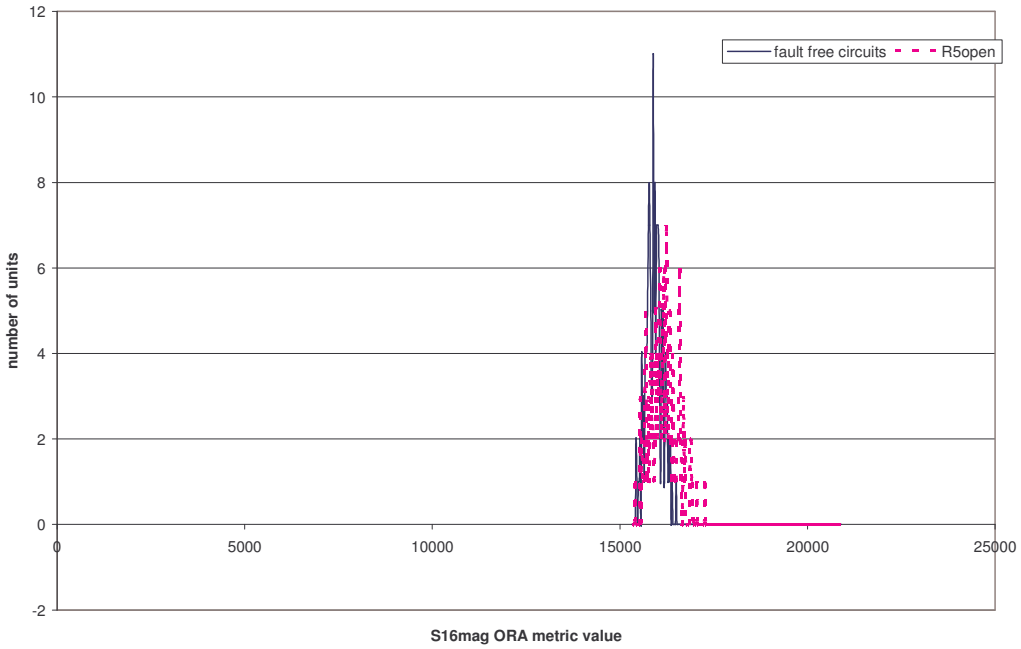


Figure F.19: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R5open(dotted) and fault-free(solid) circuits.

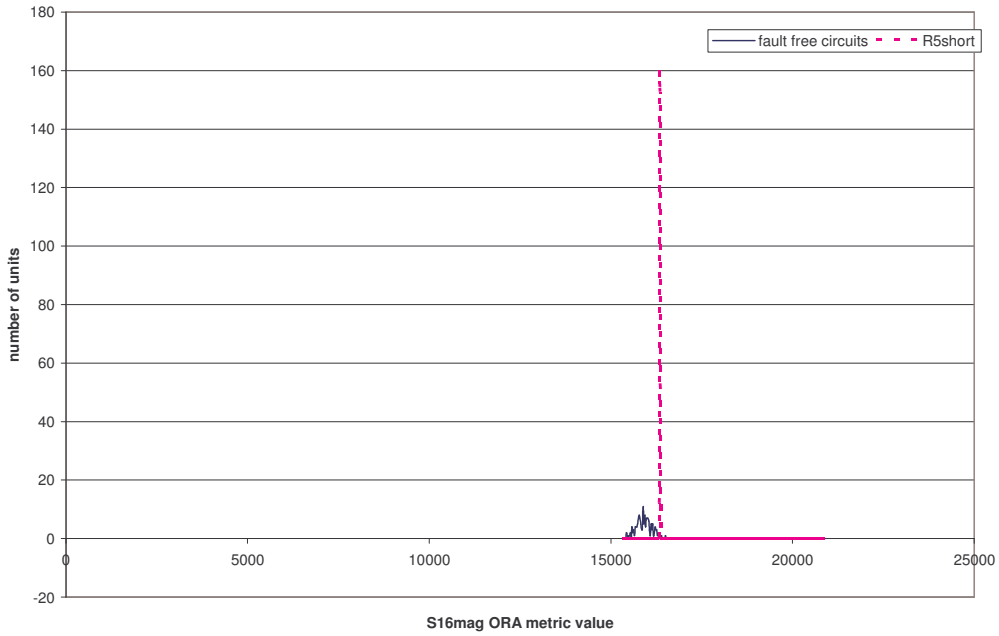


Figure F.20: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R5short(dotted) and fault-free(solid) circuits.

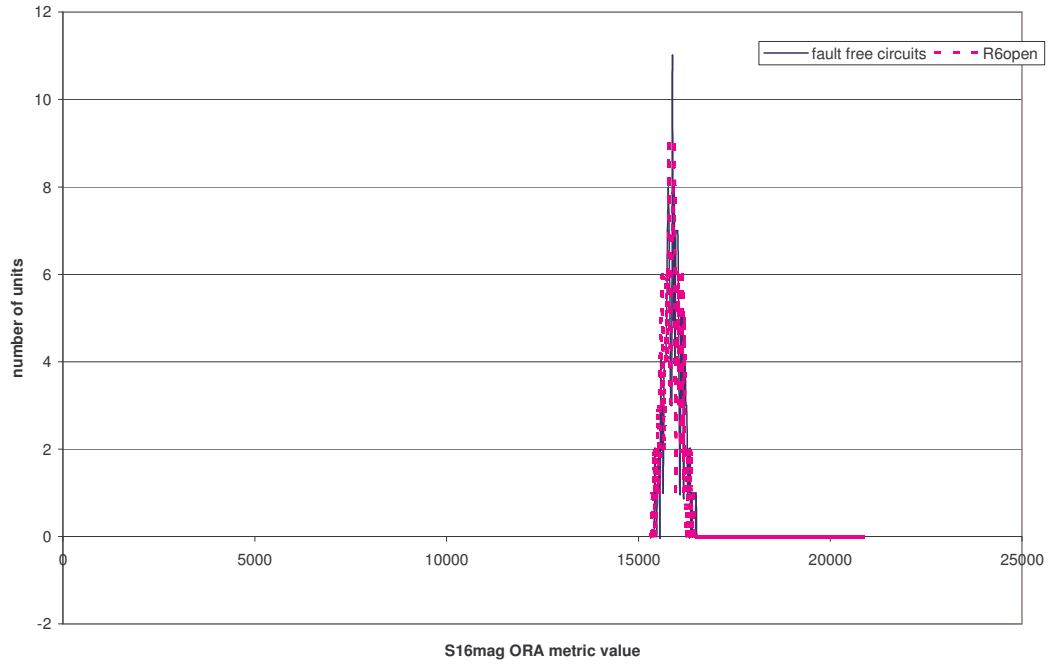


Figure F.21: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R6open(dotted) and fault-free(solid) circuits.

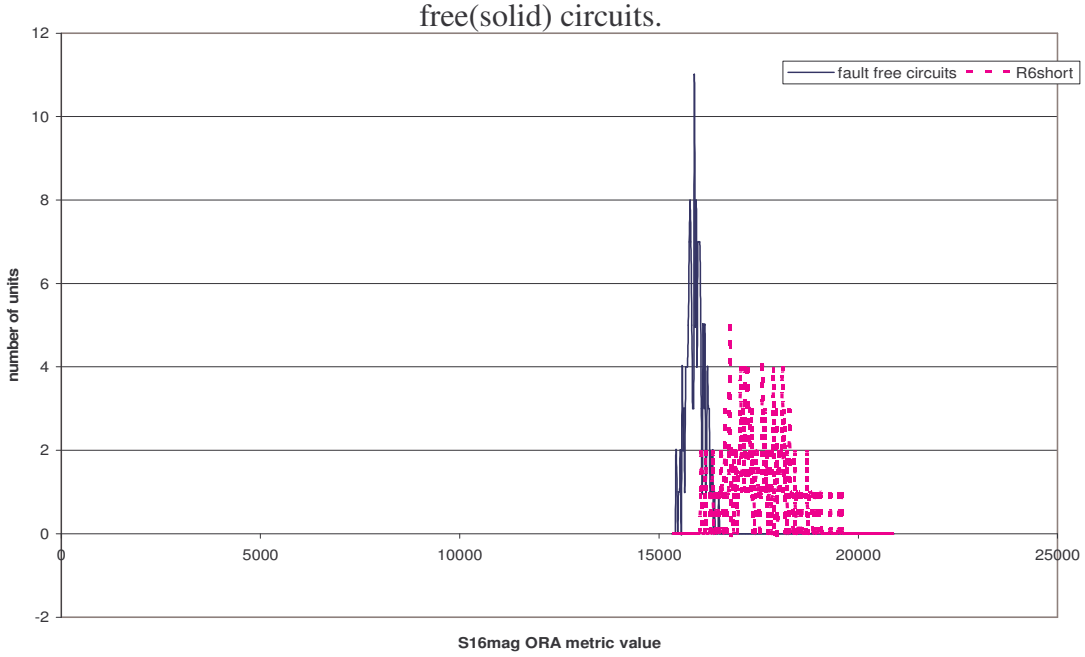


Figure F.22: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R6short(dotted) and fault-free(solid) circuits.

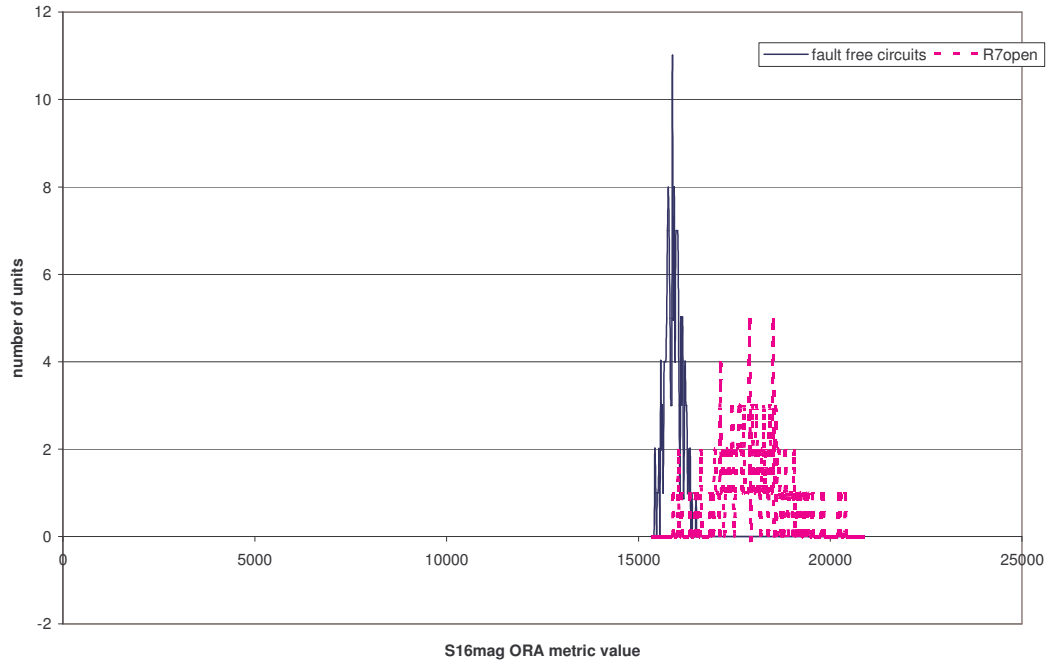


Figure F.23: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R7open(dotted) and fault-free(solid) circuits.

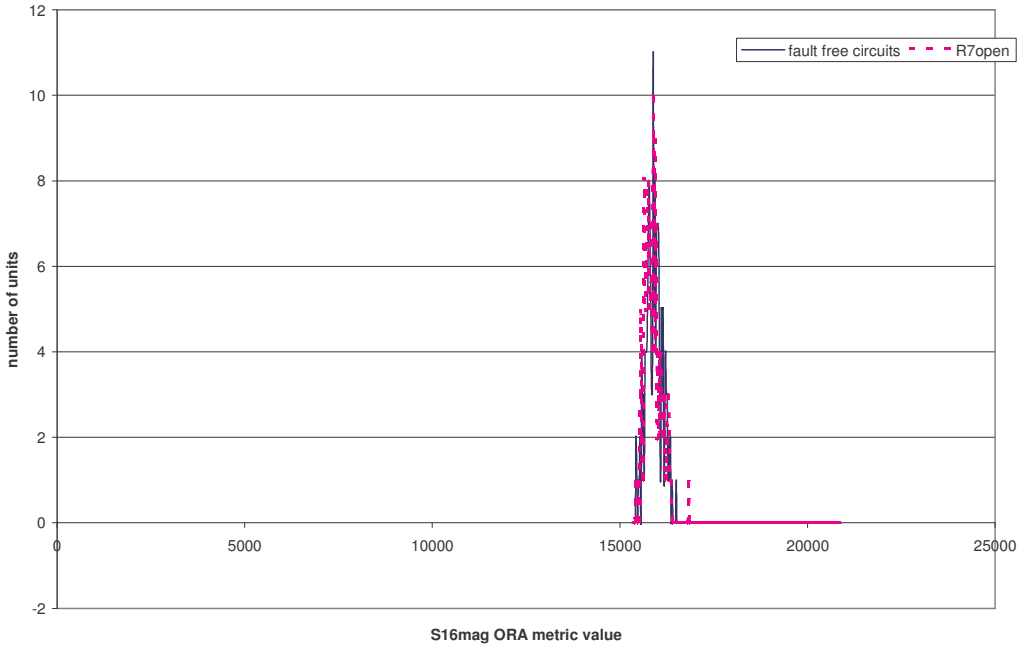


Figure F.24: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are R7open(dotted) and fault-free(solid) circuits.

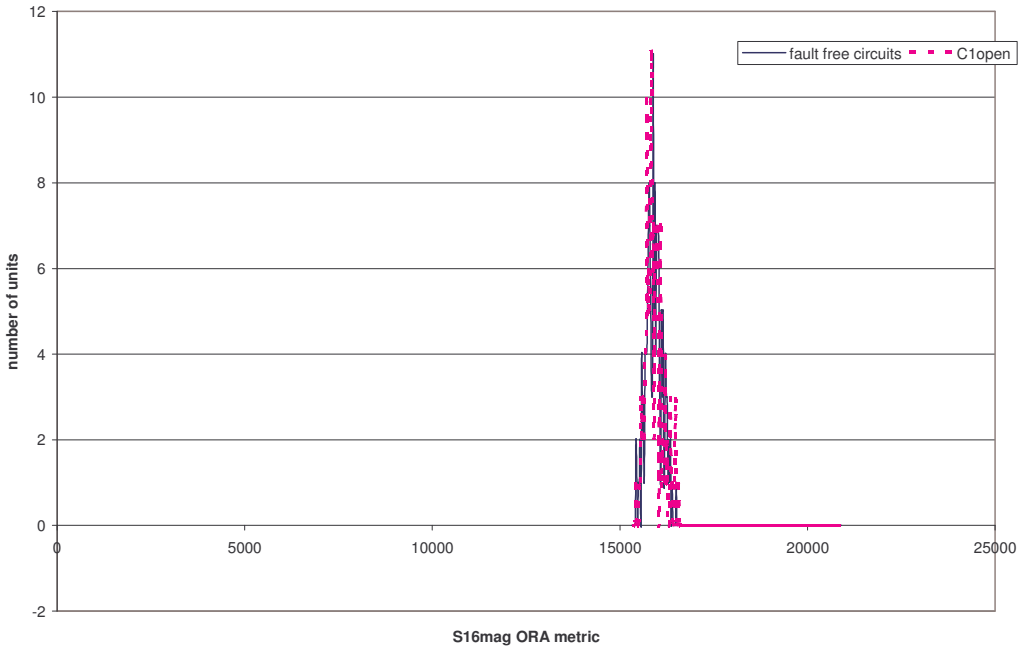


Figure F.24: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are C1open(dotted) and fault-free(solid) circuits.

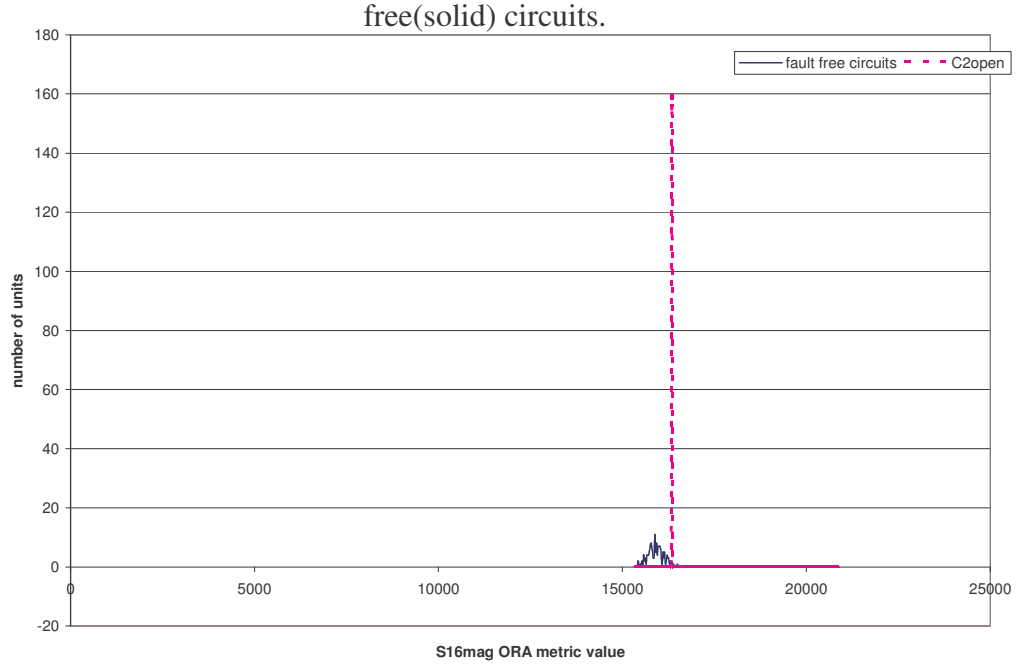


Figure F.24: Fault simulator results for S_{16mag} ORA metric for BiQuad filter at 5 MHz clock frequency (19.5 kHz effective frequency), Cup waveform, 5 V amplitude, 2.5 V offset, and 0-5V output range. The histograms shown are C2open(dotted) and fault-free(solid) circuits.