Evaluating the Digital Fault Coverage for a Mixed-Signal Built-In Self Test

by

Michael Alexander Lusco

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2011

Keywords: Built-In Self Test, Mixed-Signal, Testing

Approved by

Charles Stroud, Chair, Professor of Electrical and Computer Engineering
Foster Dai, Co-Chair, Professor of Electrical and Computer Engineering
Vashani Agrawal, Professor of Electrical and Computer Engineering
Victor Nelson, Professor of Electrical and Computer Engineering

Abstract

This thesis focuses on a digital Built-in Self-Test (BIST) approach to perform specification-oriented testing of the analog portion of a mixed-signal system. The BIST utilizes a Direct Digital Synthesizer (DDS) based test pattern generator (TPG) and a multiplier-accumulator (MAC) based output response analyzer (ORA) to stimulate and analyze the analog devices under test. This approach uses the digital-to-analog converter (DAC) and the analog-to-digital converter (ADC), which typically already exist in a mixed signal circuits, to connect the digital BIST circuitry to the analog device(s) under test (DUT).

Previous work has improved and analyzed the capabilities and effectiveness of using this BIST approach to test analog circuitry; however, little work has been done to determine the fault coverage of the digital BIST circuitry iteself. Traditionally additional test circuitry such as scan chains would be added to the BIST circuitry to provide adequate fault coverage of digital circuitry. While ensuring that the digital circuitry is thoroughly tested and functioning properly, this scan chain circuitry incures a potentially high area overhead and performance penalty. This thesis focuses on using the existing BIST circuitry to test itself by utilizing a dedicated digital loopback path. A set of test procedures is developed and analyzed which can be used to provide a set of functional tests which provide a high effective fault coverage of the digital portion of the BIST. To determine the effectiveness of these test procedures, the mixed-signal BIST circuit is simulated and single stuck-at gate-level fault coverage results are determined and presented. Finally several improvements to the dedicated loopback path are proposed and simulated to analyze possible ways to improve the fault coverage of the BIST with minimal area and performance impact.

Acknowledgments

Put text of the acknowledgments here.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The testing of embedded systems is a large field with many different approaches and techniques. Some techniques work most effectively for digital systems and others for analog systems. One approach is a hybrid test and is used to test a mixed-signal system – a system with both digital and analog circuitry. Mixed systems typically require the use of multiple testing techniques from both the digital and analog domains to be fully tested. In this chapter a short introduction to testing and relevant testing techniques for digital, analog, and mixed-signal systems is given so that the reader can develop a foundation for understanding the mixed-signal testing approach studied in this thesis.

1.1 Why Test Circuits

According to Stroud[1] there are three phases of a product where testing is of critical importance: the design phase, manufacturing phase, and the system operation phase. Each phase of the product's life cycle uses testing to achieve different goals. During the design phase of the product life cycle, the goal is to focus on finding and eliminating design errors. During manufacturing the goal changes and is focused on eliminating manufacturer defects, and finally the operation phase is focused on ensuring fault-free operation. All of these different testing goals work to improve the device by reducing costs, improving relability, etc.

1.2 The Basics of Testing

The basics of testing a circuit are similar during all phases of a product's life-cycle:

- Generate a set of input stimuli or test vectors
- Apply those vectors to the DUT
- Compare the output of the DUT to the expected output for each input value
- Note any discrepancies as indication that there is an error in the device (a fault)

In reality it is often more difficult to test circuits than this basic process makes it appear. One way to ease this difficulty is by using design for testability (DFT) techniques to increase the observability and controllability of a device during the design process[1]. Stroud defines observability and controllability in [1] as the following:

> "Controllability is the ease with which we can control a fault site and observability is the ease with which we can observe a fault site.[1]"

Ultimately these properties determine the complexity of testing a circuit. As chips grow it becomes increasingly challenging to maintain an acceptable level of controllability and observability. According to [3]:

> "The growth rate in integrated circuit (IC) transistor count is far higher than the rate for IC pins and steadily reduces the accessibility of transistors from chip pins – a big problem for IC test.[3]"

Large, modern circuits can contain billions of transistors and comparatively few IO pins[3]. Without careful design considerations these properties can negatively affect the testability of a device[1].

There are many different methods to physically testing a device. Each method varies in its requirements and has its own unique set of challenges, costs, and advantages. One traditional method of testing uses automatic testing equipment. Automatic test equipment is commonly used during manufacturing to verify chips are manufactured without defects[3]. Unfortunately as the complexity and speed of IC's has increased, automatic testing equipment has struggled to maintain an acceptable test coverage of performance-related defects[3].

To test these more complex circuits requires more advanced and higher speed automatic test equipment with additional IO capabilities[3]. [3] examines the cost of high-end automatic test equipment and finds they may become cost prohibitive for complex circuits. [2] estimates that without the inclusion of alternative testing approaches "tester costs will reach up to $20 million dollars in 2010". One such alternative to more expensive testing equipment is Built-In Self-Test (BIST). BIST describes the technique of designing a device to test itself[1] and can complement or eliminate the need for automatic test equipment[3].

1.3 Built-In Self-Test

[1] defines BIST as a circuit which can test itself and determine whether it is "good" or "faulty." In essence this entails designing the circuit to perform all of the steps in section 1.2 on itself. [1] continues by outlining the basics of a BIST architecture. This simple architecture consists of several major components including a Test Pattern Generator (TPG) which generates the input stimuli necessary to test the circuit and an Output Response Analyzer (ORA) which compares the output of the circuit to the expected output for a given input value. Additionally there is circuitry which isolates the Device Under Test (DUT) during testing as well as circuitry which controls the test during execution (Test Controller)[1].

BIST has many advantages and disadvantages when compared to other techniques. [1] quantifies these advantages and disadvantages in Table 1.1. In addition [4] performs a detailed economic analysis of including BIST in circuitry. [4] concludes that:

> "As the product develops from the IC to system level and its complexity increases, so does the complexity of successfully identifying a failure's root cause. So it makes economic sense for system owners and perhaps system producers to implement BIST.[4]"

The advantages of BIST and its ability to reduce test time and cost make it an excellent choice for testing devices[3].

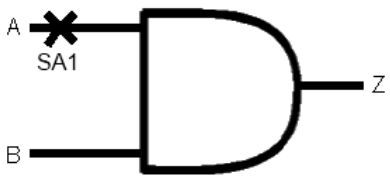
Table 1.1: Advantages and Disadvantages of BIST[1]

Advantages	**Disadvantages**
Vertical testability (wafter to system)	Area overhead
High Diagnostic resolution	Performance penalties
At-speed testing	Additional design time & effort
Reduced need for automatic test equipment	Additional risk to project
Reduced test development time & effort	
More economical burn-in testing	
Reduced manufacturing test time & cost	
Reduced time-to-market	

1.3.1 Digital Systems and Faults

Faults in a system are characterized by a fault model. A common model used for digital systems (systems functioning at discrete '1' and '0' logic values) is the gate-level stuck-at fault model[1]. According to [1], the stuck-at fault model allows for faults at the input or output of a digital logic gate. These faults can cause the input or output to be stuck-at a logic value '0' or '1' regardless of which value is applied or expected.

Figure 1.1 provides a truth table listing the faulty and fault-free output of an AND gate with its 'A' input stuck-at '1'. The 'X' notation is used to indicate the location of the fault and the 'SA1' (or 'SA0') designates rather the fault is a stuck-at '1' or stuck-at '0'[1]. The truth table given in Figure 1.1 shows how a single fault can change the behavior of a gate often having a major impact on the overall behavior of the circuit. Since digital circuits will always produce the same outputs for a given set of inputs, any difference between



Inputs (AB)	Fault Free Output (Z)	Faulty Output (Z)
0 0	0	0
0 1	0	1
1 0	0	0
1 1	1	1

Figure 1.1: AND Gate with Input A Stuck-At '1'

the expected and actual output can be exploited to determine if a circuit is functioning correctly[1]. Each clock cycle an input vector is applied and the output is compared with the expected output. If any output does not match the expected output then the chip is considered faulty and is discarded. Unfortunately the storage required to hold each input and expected output vector can be significant for large or complex chips and while feasible for costly automatic test equipment, it is often impossible due to area considerations when using a BIST approach[13].

When using BIST the input vectors are often generated by the TPG circuitry deterministically, algorithmically, or pseudo-randomly (among other methods)[1]. This keeps the size of the TPG to a minimum and removes the need for a large memory or other means of storing every input vector. Likewise it is impractical to store every expected output pattern and compare it to the actual output each clock cycle. To minimize storage requirements a signature is often used to compress the output of the circuit into a single vector. Instead of comparing each output at the end of each clock cycle, the signature is generated during the test and compared to the expected signature at the end of a test[1][13]. A signature can be generated in a number of different ways and may be as simple as a counter counting the number of 1's or 0's which occur in the output, 1's or 0's counting, or as complex as using a large Multiple Input Signature Register[1]. The most appropriate signature generation method is dependent on the requirements and output of the design and can significantly impact the effectiveness of a BIST approach. If a method is used which does not produce a suitably unique signature then faulty circuits can escape detection[1][13]. The use of an expected signature to compress the circuit output allows for a significant reduction in storage cost as in most cases only a single comparison needs to be performed to verify the circuit[13].

Returning to the example in Figure 1.1, a simple BIST can be constructed to test the AND gate. In Figure 1.2 a 2-bit counter is used as the TPG to generate all of the inputs patterns possible for a two input AND gate: "00", "01", "10", and "11". Also in the figure the output of the AND gate is connected to the *Enable* of an additional 2-bit counter to

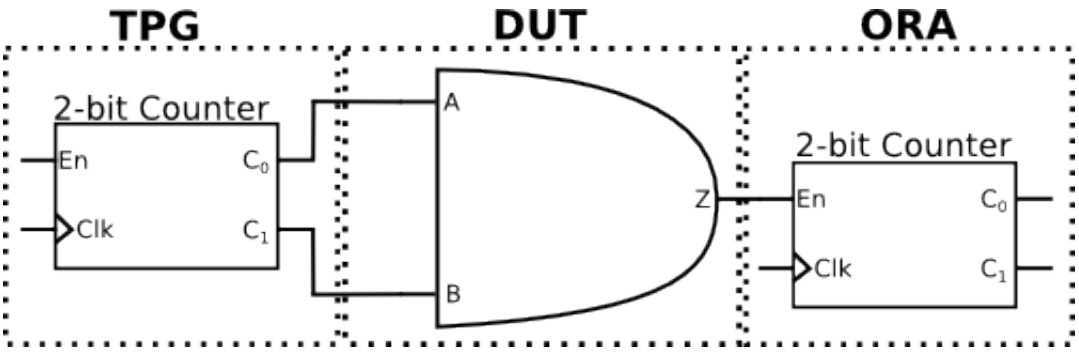


Figure 1.2: Simple BIST for an AND Gate

count each '1' and produce a signature. The fault-free signature for this circuit is "01" since a normally operating AND gate should only produce a single logic '1' when both its inputs are logic '1'. If any input is stuck-at '1' then it will produce at least one additional '1'; if any input is stuck-at '0' it will never produce a logic '1'. These two conditions will produce invalid signatures. During execution of the BIST sequence, the TPG counts from "00" to "11" and the 1's counter will increment for each '1' occurring at the output of the gate. At the end of the sequence, if the value in the 1's counter is not "01" then the gate is faulty. This example is greatly simplified and is missing required circuitry to start and stop the BIST as well as a method to isolate the inputs of the gate; however, it does demonstrate the general principal behind using BIST to test a digital circuit.

1.3.2 Analog Systems and Faults

Analog systems function differently than digital systems. Unlike a digital system which only has two discrete values, analog systems are continuous waveforms with multiple levels and voltages[5]. In addition to the complexities of analog waveforms, analog components operate within a range of acceptable values[5]. This difference makes testing analog components for defects (defect-oriented testing) significantly more challenging and requires a more complex fault model. In analog components faults are classified as either parametric (soft-faults) or catastrophic (hard-faults)[11]. Parametric faults are those which affect the

performance of a specific component causing it to operate outside of its expected tolerance range, for example a resistor which has a lower than expected resistance. In constrast catastrophic faults are those which cause a component to fail, such as resistor which is no longer conductive and appears as an open circuit[11]. The simulation of these different faults requires complicated and time consuming methods such as Monte Carlo analysis to determine different component values which allow fault-free circuit operation[5]. Compounding these issues is the fact that analog components

> "function collectively to perform the overall functionality of a given circuit and, as a result cannot always be treated as a collection of individual components[5]."

and consequently are difficult to isolate for effective defect-oriented testing[14]. Furthermore any circuitry added into an analog circuit may potentially interfere and change the operating range or output of that circuit[5]. This potential interference requires that any analog testing circuitry be carefully simulated and verified to ensure it does not negatively affect the overal circuit performance. An example of a defected-oriented approach, oscillation testing is a testing approach which reconfigures the analog CUT so that it oscillates; this oscillation frequency is then measured and compared to an expected frequency. If the measured frequency falls outside the expected range, the circuit is considered faulty[12]. This method has been shown to be effective for the detection of catastrophic faults and some parametric faults; however, it can require a significant amount of planning and design effort as it may significantly impact the analog circuitry[12][5].

A simpler (and preferred[5]) method for testing analog components is via functional testing. [6] defines functional tests as:

> "... those which measure a circuit's dynamic behavior ...[6]"

Functional or specification testing is achieved by performing a set of tests to determine if a system is operating correctly as defined by its specifications. This approach is used to

test the entire analog system collectively instead of attempting to understand the implications of specific faulty components[14]. Specification testing may include the testing of important analog characteristics such as frequency response, linearity, and signal-to-noise ratio (SNR)[5]; however, the characteristics which are important to test will vary between designs. To adequately test most analog components, multiple measurements of several different characteristics must be taken as a single characteristic is usually not sufficient to ensure fault-free operation[14]. This process may require extra development time as test procedures must be developed to test each characteristic and additional time is required to perform each test[14].

1.3.3 Mixed Signal Testing

In a mixed-signal environment both digital and analog systems coexist and interact. Due to the previously discussed differences in testing analog and digital systems, the testing of the analog and digital sub-systems is generally developed separately and performed using using different test procedures and approaches[14]. Ideally a designer would like to limit any duplicate work and take advantage of a BIST approach which can be used to test both the analog and digital sub-systems.

[1] defines a BIST approach to testing mixed-signal systems shown in Figure 1.3. This BIST uses a digital BIST approach to functionally test the analog sub-system by measuring certain analog characteristics which can be used ensure that the circuit is operating within its specifications. This architecture is largely digital and thus can be integrated into the digital circuitry already in the system with minimal analog overhead. This prevents excessive interference with the analog circuitry, excluding analog multiplexers which facilitate the sending and retrieving of test values to and from the analog sub-system[1]. To test the analog circuitry, the BIST uses the existing Digital to Analog Converter (DAC) to convert digitally generated test patterns from the TPG to analog signals and the existing Analog

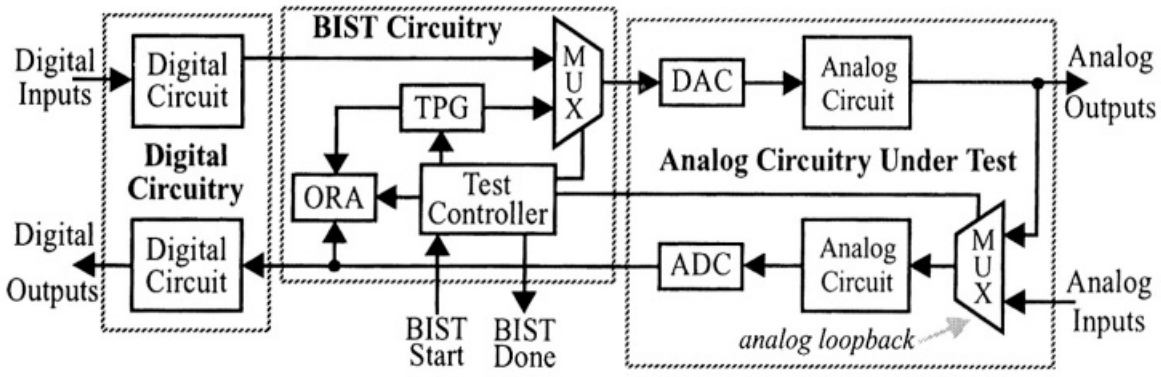


Figure 1.3: BIST Approach for Mixed-Signal Systems[1]

to Digital Converter (ADC) to convert the analog response back into the digital domain for analysis by the ORA[1].

Figure 1.3 shows a basic version of this mixed-signal BIST approach using two multiplexers. In this case one multiplexer is placed before the DAC to select between the BIST patterns and the system circuitry and a second multiplexer is positioned at the input of the analog system to select between the system level inputs and the analog outputs. These two multiplexers form a loop allowing the generated TPG pattern to be converted into an analog signal and propagate through any analog circuitry before being routed back through the analog inputs to the DAC for analysis by the ORA. While this implementation does allow testing of all analog components, it does not allow a high level of diagnostic resolution[5]. To obtain a higher diagnostic resolution, additional analog multiplexers can be added to further partition the system. Figure 1.4 shows an example of an implementation with three separate multiplexed or loopback paths to facilitate a higher level of diagnostic resolution. In Figure 1.4 the shortest loop, the short dashed path, is a digital only path (digital loopback path) which can be used to test that the digital BIST is fault-free. The next loopback path, the dashed path, connects the output of the DAC to the ADC bypassing any of the analog circuitry (from here on referred to as the bypass path). This allows the verification of the ADC and DAC separately from the analog circuit. The final path is the analog test path,

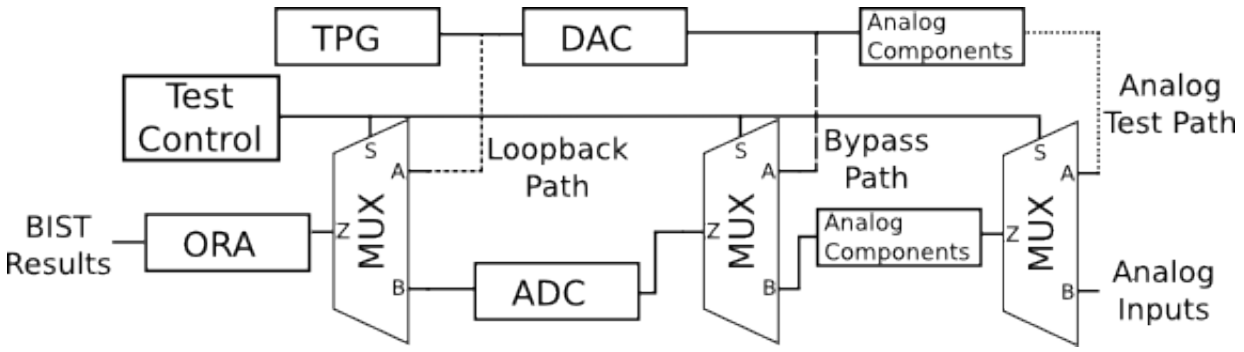


Figure 1.4: Mixed-Signal BIST with Multiple Loopback Paths

the dotted path, and is similar to the path in Figure 1.3 in that it is responsible for testing the analog circuitry. There is no limit to the number of multiplexers that can be added to the system other than the increase in area overhead[5]. Additional analog multiplexers could be added to the example in Figure 1.4 to further partition the analog circuitry and resolve faulty behavior to a specific portion of the analog DUT.

In Figure 1.4 the digital loopback path is of particular importance. [7] has shown that a digital loopback path is highly advantageous when testing the digital portion of the BIST circuitry. In a digital only environment the BIST produces exactly one output for a given set of inputs; however, without the digital loopback path the TPG outputs are not directly observable. To observe the outputs of the TPG requires conversion via the DAC and reconversion via the ADC. This will cause variations in the expected results due to the inherent variations which occur in analog signals and circuits. To account for these variations, a range of acceptable values must be considered instead of an exact signature[7].

Figure 1.5 shows a comparison of the maximum achievable digital fault coverage versus the allowed distance away from the expected good signature for different ORA designs. In the figure, three different 8-bit ORA accumulator designs are considered (ORA design is discussed in more detail in the next chapter). Figure 1.5 shows that regardless of design a (in some cases significant) reduction in the maximum achievable fault coverage is to be expected when a range of good values is considered instead of an exact signature. Thus the

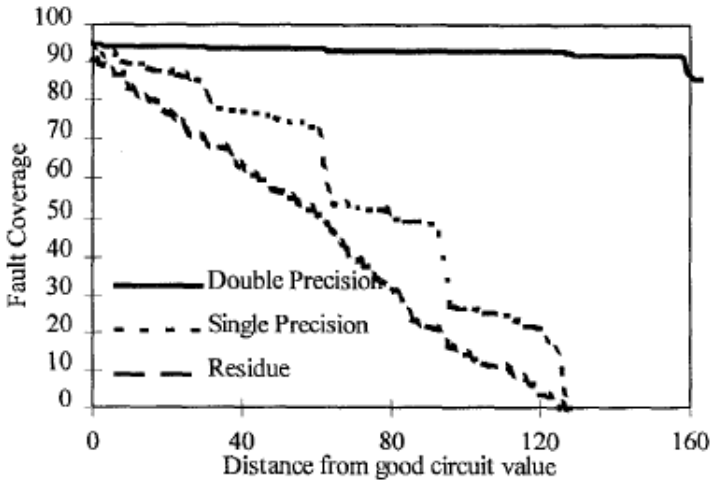


Figure 1.5: Fault Coverage v. Acceptable Value Range[7]

only way to achieve the maximum fault coverage is to not use a range of acceptable values and perform a digital only test. This requires the digital loopback path[7]. Performing a digital only test separates the digital and analog systems when testing. This allows for the usage of tools and techniques which target digital components separately from those used to generate the analog functional tests on the analog sub-system[7]. When the three loopback paths shown in Figure 1.4 are present, the test procedure should be performed first over the digital loopback path to verify the digital BIST, then using the bypass path to verify that the ADC and DAC path are functioning correctly, and finally over the entire circuit to verify the analog DUT[5][7].

Though this technique still requires the separate testing of the analog and digital circuitry, it is an improvement over the previously discussed method since the same BIST circuitry can be used to test both the digital and analog circuitry[1]. This limits the amount of work that must be duplicated to the determination of which tests must be run and the expected values or range of values that must be considered. Furthermore the basic procedure for testing the analog and digital circuits is the same (the only differences being which loopback path is selected and the actual test being performed) which will limit the differences between test sessions.

1.4 Summary

In the previous chapter a high-level look at both digital and analog systems has been given as well as the challenges of testing these systems for faults. The concept of BIST has been presented along with its advantages and a simple BIST architecture. In addition the challenges of testing mixed-signal systems has been discussed and a basic mixed-signal BIST model has been given along with an overview of the challenges of testing the BIST without a digital loopback path. In the next chapter a more detailed explanation of fault simulation and of the specific mixed-signal BIST approach studied in this thesis is given. This approach builds upon the simple architecture discussed in the previous section and addresses the challenges discussed in Section 1.3.2. Building upon this explanation, Chapter III explores the preparation necessary for simulating the mixed-signal BIST including necessary conversions and verification. Chapter IV explores the main topic of this thesis: testing the actual mixed-signal testing circuitry to determine the level of fault coverage achievable. Methods for improving the BIST and the maximum achievable fault coverage are explored in Chapter V, before a summary and conclusions are given in Chapter VI.

Chapter 2

Background

Before discussing the fault simulations performed in this thesis, it is important to have a more thorough understanding of fault simulation techniques. In this chapter, an overview of several fault simulation techniques will be given as well as an overview of the bridging fault model which models faults between nets. Following this discussion a more detailed outline of the simulation and design process will be given. Next the fault simulator AUSIM is detailed which is used to evaluate the fault coverage of the mixed-signal BIST approach studied in the rest of this thesis. This chapter concludes with a detailed discussion of the architecture of the mixed-signal BIST and its capabilities.

2.1 Fault Simulations

Knowledge of the basic concepts of simulation is important to understanding how fault simulations are performed. The simulation process begins with a netlist. The netlist represents the circuit design to be simulated and defines the structure of the circuit at the gate level. This includes both the gates used by the circuit as well as the interconnections between those gates[19]. The generation of a netlist is discussed in more detail in Section 2.1.3. According to Section 1.2, the next step is to generate a set of test vectors to be used to stimulate the circuit during testing. Test vectors may be generated automatically using an Automatic Test Pattern Generator (ATPG)[19] or manually by the designer. When performing fault simulations, these vectors should ideally exercise as much of the internal circuitry of the design as possible so that a high fault coverage can be achieved (fault coverage is discussed towards the end of this section); consequently, test vectors generated using an ATPG can be advantageous as they typically guarantee a very high fault coverage[19]. If vectors are

created manually by the designer then it may be necessary to perform a logic simulation of the circuit to ensure the vectors are valid and to obtain the fault-free output of the circuit (when using an ATPG the fault-free output is generally recorded along with the test vector set). With both the test vector set generated and fault-free output of the circuit known, fault simulations can begin.

Fault simulation starts with the selection of a fault model which characterizes the fault behavior to be simulated. There are several different fault models used for digital circuits, but the focus of this thesis will be the gate-level stuck-at fault model which was introduced in Section 1.3.1. The stuck-at fault model has a low computation cost and accurately represents the behavior of faults seen at the gate-level of digital circuits[1]. Other models exist which characterize different fault behavior and have their own set of advantages and disadvantages. The bridging fault model, which will be introduced in Section 2.1.2, focuses on faults which occur between nets as opposed to those that occur at gate inputs and outputs and is used to accurately simulate faults that occur in circuit routing[17]. The transistor fault model targets faults which occur at the transistor level. This level of detail makes it more computationally expensive to simulate compared to the gate-level model; however, this model more accurately represents the behavior of faults which occur during the manfacturing process[15] and may detect faults which are not detectable using the stuck-at fault model. Regardless of its accuracy, it is more common to use the gate-level stuck-at model for digital fault simulations since it is less computationally expensive and is acceptably accurate at modeling the behavior of common defects in digital systems[15].

Once a fault model is chosen each fault must be simulated and the output of the circuit recorded. The output of the faulty circuit is compared to the fault-free output and if any discrepancy is found then the fault is recorded as detected. Similarly if the output of the circuit is always the same as the fault-free circuit then the fault is not detected[15]. In some cases a fault may be considered potentially detected; this special case is caused by an unknown logic value occurring in the circuit and is an artifact of simulation. An unknown

logic value will occur if a circuit element is not initialized properly[1]. In physical hardware the value must be either logic '0' or logic '1'; however, in simulation it is unknown which logic value it will initialize to which causes the detection of the fault to be uncertain in simulation[1]. The percentage of faults detected is said to be the fault coverage of the test vector set[15][1]. Fault coverage is typically calculated using Equation 2.1 where D is the number of detected faults, P is the number of potentially detected faults, X is the number of undetectable faults and T is the total number of faults simulated[1].

$$F_C = \frac{(D + .5P)}{(T - X)} \tag{2.1}$$

Undetectable faults are faults which are impossible to detect by any test vector. These faults are often caused by design issues such as reconvergent fan-out and redundant logic[1] and since they cannot be detected are typically not considered when calculating fault coverage. In the equation P is multiplied by .5 denoting that there is a 50% change of a potentially detected fault being detected. This represents the chance of an uninitialized logic value initializing to the logic value required for detection when testing is performed in physical hardware. This coefficient may be changed to represent a higher or lower chance of detecting potentially detectable faults[1].

For a large number of faults or large number of test vectors the simulation process can take a large amount of time to complete. In the worst case the time required to complete a fault simulation is shown in Equation 2.2, where T_{vec} is the time required to simulate a single vector, N_{vec} is the number of test vectors to be simulated, and N_{flts} is the number of faults to be simulated.

$$Time = T_{vec} \times N_{vec} \times N_{flts} \tag{2.2}$$

There are a couple of methods which can decrease the amount of time taken to simulate a list of faults. One common method is fault dropping. When using fault dropping a fault is only simulated until a discrepancy between the output of the circuit and the fault-free output is

found. At that point the fault is recorded as detected, simulation of the fault is halted, and a new fault is simulated[1]. The time required to perform fault simulation when using fault dropping is shown in Equation 2.3 where N_{flts} is the number of faults to be simulated, T_{vec} is the time required to simulated a single vector, and N_{vec_i} is the number of vectors simulated for the i^{th} fault.

$$Time = \sum_{i=0}^{N_{flts}} T_{vec} \times N_{vec_i} \tag{2.3}$$

In the case where the i^{th} simulated fault is detected early in the simulation, N_{vec_i} will be small shortening the simulation time for the fault; in contrast, if the ith fault is detected toward the end of the simulation or not detected at all then N_{vec_i} will be approximately N_{vec} causing little to no savings in simulation time for the fault[1].

Further speed up can be obtained by performing parallel fault simulation. Equations 2.2 and 2.3 show the time required to perform serial fault simulations where a single fault is simulated at a time. To decrease the time required for simulation, it is beneficial to simulate multiple faults concurrently[1]. This is a common approach to decreasing simulation time and at least in the case of gate-level stuck-at fault simulation, does not require any additional considerations by the user[18]. Commonly 32 faults[2] are simulated in parallel though different simulators may have options to simulate more or less faults[18]. Equation 2.4 defines the worst case time required to perform parallel fault simulation and Equation 2.5 defines the time required to perform parallel fault simulation with fault dropping assuming 32 faults are simulated in parallel (in this case N_{vec} is the number of vectors required to detect all faults in a parallel group).

$$Time = T_{vec} \times N_{vec} \times \frac{N_{flts}}{32} \tag{2.4}$$

[1]This does not mean perform the simulation of multiple faults in the same circuit simultaneously; but instead, it means perform multiple simulations in parallel each with a single fault[18]

[2]An integer is 32-bits on a 32-bit machine; consequently, simulating 32 faults allows for the use of the integer data type in the simulator and makes parallel fault simulation easier to implement[18]

$$Time = \sum_{i=0}^{\frac{N_{flts}}{32}} T_{vec} \times N_{vec_i} \tag{2.5}$$

Using both fault dropping and parallel fault simulation can greatly decrease the required simulation time for a large circuit. Due to this benefit both of these methods are used in the fault simulations performed in this thesis. The next section discusses further optimizations which can be performed to reduce the number of faults simulated and consequently the time required to perform simulation.

2.1.1 Collapsed vs Uncollapsed Faults

When performing fault simulations certain optimizations can be made to improve the efficency of the simulation. One such optimization is fault collapsing. With the gate-level single stuck-at fault model, each gate input and output can be stuck-at logic '0' or logic '1'. For elementary logic gates this leads to many faults which produce identical faulty behavior; these faults can be said to be equivalent[15]. During simulation these equivalent faults can be collapsed, requiring only a single fault out of the group of equivalent faults to be simulated[15]. Depending on the circuit this can substantially reduce the number of faults to be simulated while still accurately representing the faults which can occur in the circuit[15]. Figure 2.1 shows an example of fault collapsing. The NOR gate in the figure has six uncollapsed faults; however, when either input is stuck-at '1' it is equivalent to the output being stuck-at '0' and vice versa (since a logic '1' is the controlling input for a NOR gate) which results in four collapsed faults. These equivalent faults are shown in bold in the figure. [1] states that the number of collapsed faults for any elementary logic gate with greater than one input is $K + 2$ where K is the number of inputs to the gate. This is certainly apparent with the NOR gate in Figure 2.1, where $K + 2 = 4$. Additionally when the output of a gate is connected to exactly one input of another gate, a fault occurring at the output of the source gate is indistinguishable from the same fault occurring at the input of the connected gate[1]. These faults can be structurally collapsed together which leads to a large chain of

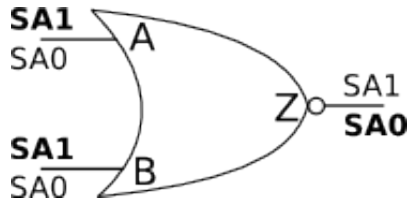


Figure 2.1: NOR Gate with Six Uncollapsed and Four Collapsed Faults

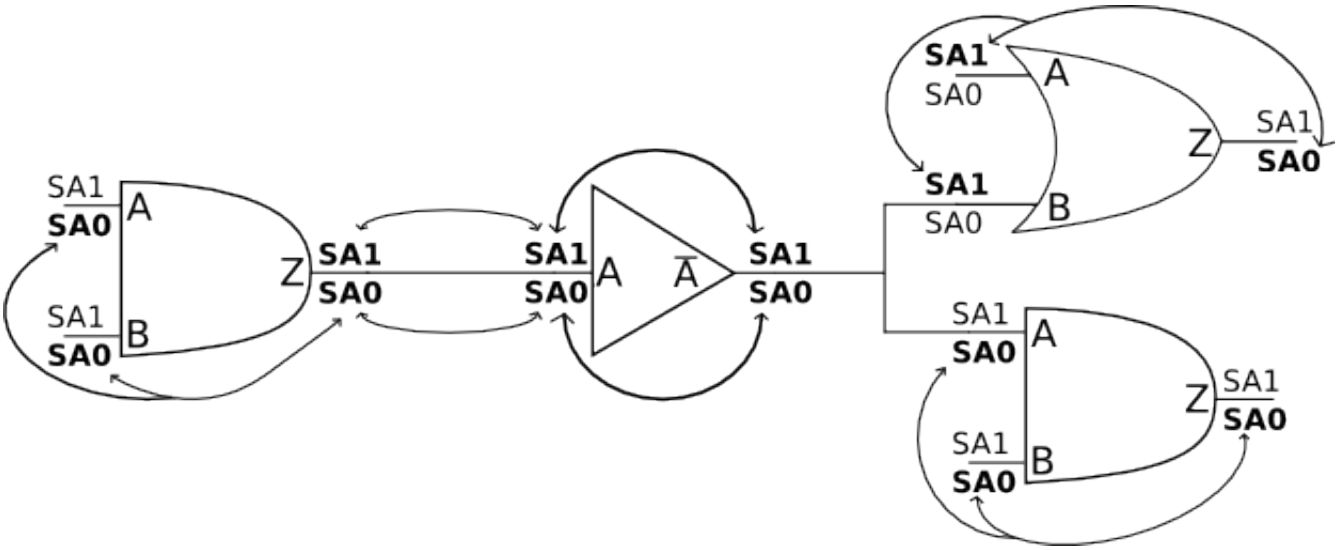


Figure 2.2: Structural Fault Collapsing Performed on a Group of Logic Gates

faults being collapsed such as the example in Figure 2.2[1]. In the figure groups of equivalent faults are shown with a line drawn between them. Unfortunately due to the limitation of the single stuck-at fault model, which does not allow multiple faults to appear in the circuit simultaneously (as opposed to a multiple fault model which allows this), the fan-out stem in Figure 2.2 cannot be collapsed[15]. Since the output of the inverter fans-out to the input of two different gates, collapsing these faults would make a fault appear to be on both the input of the AND gate and the OR gate simultaneously which violates the single stuck-at fault model and is not allowed[15]. Counting the faults in the figure, there are a total of 22 uncollapsed faults and a total of 12 collapsed faults, a significant reduction.

In the single stuck-at fault model the number of uncollapsed faults in a circuit are $2 \times G_{io}$ where G_{io} is the number of gate inputs and outputs in the circuit[1]. In contrast the number

of collapsed faults can be determined by Equation 2.6 where P_o is the number of primary outputs, F_o is the number of fan-out stems, G_i is the number of gate inputs, and N_i is the number of inverters in the circuit[1].

$$F = 2(P_o + F_o) + G_i - N_i \tag{2.6}$$

As an example, the BIST model evaluated in this thesis has a total of 83386 uncollapsed faults. It has 30 primary outputs, 4436 fan out stems, 27861 gate inputs, and 2740 inverters. By using Equation 2.6 the number of collapsed faults in the BIST circuitry is 34053. This results in a 59% reduction in the number of faults to be simulated.

[1] discusses the advantages and disadvantages of using the collapsed or uncollapsed fault set for simulation purposes. According to [1] it is obvious that the simulation time can be greatly reduced by using the collapsed fault list. This is very advantageous as more simulations can be performed in the same amount of time. However, [1] does say that the uncollapsed fault list more accurately represents the possible defects which occur during manufacturing. Due to this difference, the fault coverage obtained with the collapsed fault list will be different than the coverage according to the uncollapsed fault list by a few percent[1]. Through the accuracy of the uncollapsed fault list is preferable, the collapsed fault list is more often used due to its decreased simulation time[1].

2.1.2 The Bridging Fault Model

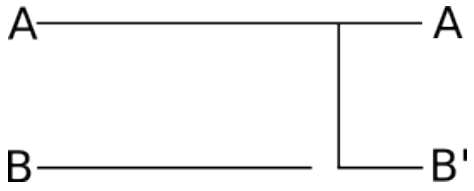
Though the focus of this thesis is gate-level stuck-at fault simulations, additional models may be of interest for future work; specifically bridging fault simulations are commonly performed to assess the fault coverage of faults which occur between nets. There are a number of different fault models which describe the behavior of bridging faults, faults where two nets are shorted together due to a manufacturing defect, including the Wired OR/AND, Dominant, and Dominant OR/AND fault model[17]. In this section we will focus on the

dominant bridging fault model as it is the most commonly used[1]. For many years it was a commonly held belief that a high stuck-at fault coverage provided a high bridging fault coverage; however, more recent work has shown that this is not always the case and that it is important to perform these simulations separately to ensure that an acceptably high bridging fault coverage is achieved[16]. Unfortunately bridging fault simulation can be significantly more time consuming than stuck-at fault coverage due to the large number of fault sites that must be considered.

$$F_B = (N^2 - N)/2 \tag{2.7}$$

Equation 2.7 shows the calculation to determine the number of possible bridging fault sites in a circuit, where N is the number of nets in the circuit[1]. Like the previous section, the BIST model can be used to give a sense of scale. The BIST studied in this thesis has 15221 nets, by applying Equation 2.7 the number of bridging fault sites is 115,831,810! Depending on the bridging fault model up to four faults (when using the DOM AND/OR model[17]) are possible at each fault site, so this means that the worst case number of faults to be simulated is almost 500 million! Similar to the gate-level model, bridging faults can also be structurally collapsed which can substaintly lower the number of faults to be simulated (using the same circuit the number of collapsed faults was approximiately 30K when using the more common dominant model).

As discussed in the previous paragraph, there are different models to define the types of bridging faults which can occur. Since it is the most common, the dominant bridging fault model will be discussed. According to the dominant bridging fault model, two different faults can occur at a bridging fault site: net A can dominate net B or net B can dominate net A. Whichever net is dominated will be affected by the other net's current logic value; this occurs due to the dominate net having a stronger drive transistor[17]. Figure 2.3 shows an example of a dominant bridging fault where net A dominates net B, along with a truth table containing the fault-free and faulty behavior of the circuit. When net B has the same logic value as net A its output is unaffected; however, when the logic value of net B is different



(A)

Net A	Net B	Faulty' Net B'
0	0	0
0	1	0
1	0	1
1	1	1

(B)

Figure 2.3: Bridging Fault where Net A Dominates Net B

than that of net A, net B's logic value will be changed to that of net A. The first two columns of the truth table in Figure 2.3 represent the fault-free behavior of net A and net B and the remaining column B' represents the faulty output of net B when net A dominates it. In the second row when net A is a '0' and net B should be a '1', B' is instead a '0' since it is dominated by net A. Likewise in the third row when net A is a '1', net B should be a '0' but instead is a '1' due to the influence from net A.

Bridging faults are important to address during the design of a circuit. Unfortunately they can be computationally expensive and can be difficult to test and observe[17]. Bridging faults were not simulated in this thesis though it is a target of future work.

2.1.3 Design Process

The design of a digital circuit typically starts with a behavioral description of a circuit. This is generally done in a high-level design language such as Verilog or VHSIC Hardware Design Language (VHDL) and usually will not include any implementation details of the circuit[19]. This code is then fed into a synthesis computer-aided design (CAD) tool which will interpret the behavioral description of the design, combine it with parameters such as the implementation technology, timing information, and/or area constraints, and ultimately produce a gate-level netlist of the design[19]. Following this step a place-and-route tool takes each gate and its interconnections and decides where on the silicon chip (or in the FPGA or other programmable device) to place the gate so that any timing and area constraints can

be achieved. Once completed the design is said to be "post-layout" indicating it is ready for fabrication[19].

Logic simulation is important during each step of this design process, so that the circuit behavior can be verified before fabrication[19]. During the behavioral modeling phase, design verification is performed to ensure the description of the circuit is correct and that no design errors have been made. Simulation is also performed after synthesis has taken place to verify that the circuit's behavior is still correct after it has been implemented at the gate-level and to investigate any potential timing problems with the circuit[19]. Following post-synthesis simulation, post-layout simulation is also performed. Post-layout represents the final version of the circuit to be fabricated. Simulation can be performed using the circuit's final timing information and can incorporate many different performance characteristics. Post-layout is the final opportunity to verify a circuit is functioning correctly before fabrication[19].

In constrast to logic simulation, fault simulation is not typically performed at any of the high-level steps in the design process[1]. Ideally the post-layout netlist should be used for all fault simulations, since the layout can have a significant impact on the faults in a circuit[15]. This is especially true of bridging fault simulations as routing is not finalized until the layout stage and thus cannot be accurately performed earlier in the design process[15]; however, other fault models may be simulated earlier in the design verification process[1]. In this thesis all fault simulations of the BIST are performed using the post-layout design. As discussed in the next chapter, this does present a challenge since the post-layout design produced by the CAD tools is a Verilog netlist which is not a format used by the fault simulator. As such a tool was written to convert a Verilog netlist to the ASL netlist format recognized by our simulator AUSIM (ASL and the fault simulator AUSIM will be discussed in Section 2.2).

2.1.4 Value Change Dump File

The Value Change Dump or VCD format is defined in section 15 of the IEEE 1364-1995 standard for Verilog[10]. Originally created as a light-weight format to dump simulation results for post-processing[10], the VCD format is an industry standard format and is typically used to store logic simulation results in as small of a footprint as possible[10]. In addition as an IEEE standard it is supported in a number of free and commercial viewers which can easily visualize and navigate the underlying simulation data.

The VCD format compresses simulation results into a smaller footprint by only storing the changes between vectors instead of every vector[10]. While this can make the output of the file less readable it makes it very easy to store large amounts of simulation data and for a viewer to parse and display the file. An example of a VCD file is shown in Table 2.1.4[3]. On the right in Table 2.1.4 is a text description explaining each section of the VCD file. In addition to the file size reduction, the key benefit of conversion to the VCD format is the ability to visualize the output of a logic simulation in one of a number of commercial viewer applications. For the example in Figure 2.4 GTKWave was used to visualize the VCD file in Table 2.1.4. GTKWave is a free, open source VCD file viewer for Linux available on-line at `http://gtkwave.sourceforge.net`. In GTKWave each IO is displayed as a waveform. The displayed waveforms can be panned, zoomed and combined. This functionality alone makes it much easier to verify the output of a complex logic simulation. Conversion to the VCD file format and the GTKWave viewer are used in this thesis to simplify the verification of test vectors used in fault simulations.

2.2 AUSIM Fault Simulator

For the fault simulations performed in this thesis the Auburn University SIMulator or AUSIM is used. Developed by Dr. Charles Stroud[8][9], AUSIM can perform both logic

[3]It is important to note that while all IO in this example are a single bit, values in a VCD file can be any arbitrary width and in that case operate as a bit-vector

Table 2.1: Example VCD File

VCD File	Description
$date	The date section defines the day
Tuesday, December 07, 2010	which this file was created
$end	
$version Generated For ASL2VCD 1.0	A comment detailing the
Alex Lusco	version of the application creating this file
$end	
$comment	
VCD File	An additional comment section detailing
$end	information about the file
$timescale 10ns **$end**	Defines the time scale used in this file
$scope module logic **$end**	The simulation scope, particular to the simulation
$var wire 1 ! A **$end**	This section aliases each input with a single ASCII
$var wire 1 ” B **$end**	character. For long IO names this shrinks them to a
$var wire 1 # S **$end**	single character allowing for the ability to condense
$var wire 1 $ C **$end**	the dump for large simulations greatly
$upscope $end	
$enddefinitions $end	End of header
$dumpvars	Sets the initial value for each input
0!	Each IO must be set to an initial value here
0”	The !,”,#, & $ characters represent the
0#	nets aliased in the $var section
0$	
$end	
#1	The #N construct denotes a time step to apply the
1”	following value changes at, in this case at 10ns
1#	Only nets whose value changed will appear in the time step
#2	These changes occurred at 20ns
1!	
0”	
#3	These changes occurred at 30ns
1”	
0#	
1$	
#4	The end of the simulation, no changes occurred

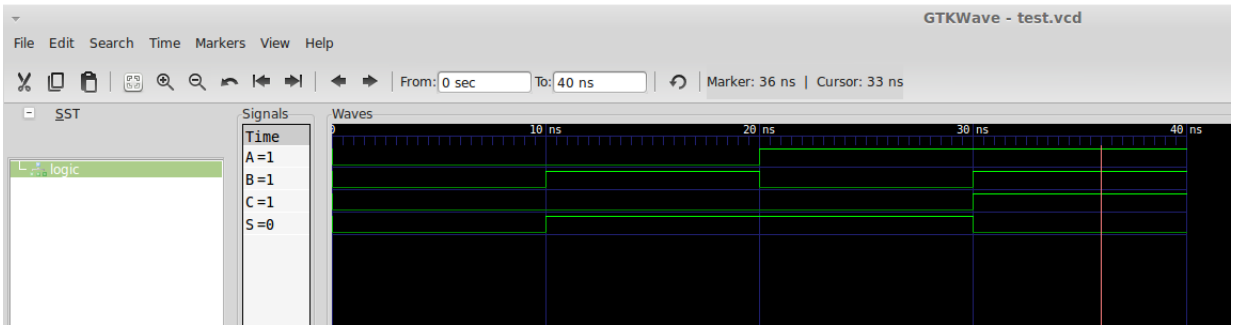
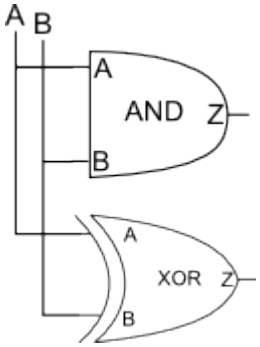


Figure 2.4: VCD File Visualized in the GTKWave Viewer

and fault simulations. In addition it supports multiple fault models including the gate-level stuck-at and multiple bridging fault models[8]. It supports all of the previously discussed methods of improving simulation time including fault collapsing and parallel fault simulation of both bridging and stuck-at faults[8]. A detailed description of the operation and usage of AUSIM will not be given in this thesis; however, more information can be found in [8][9]; instead, the focus of this section is an overview of the Auburn Simulation Language or ASL which is the netlist format used by AUSIM.

2.2.1 ASL Netlist Format

Before a circuit can be simulated by AUSIM it must be in a format that AUSIM can understand. The Auburn Simulation Language (ASL) is the circuit description used by AUSIM[9]. ASL is used to provide a textual representation of a circuit at the gate-level to the simulator. This gate-level net-list is used to describe each connection and gate used in a circuit design and allows the simulator to build a representation of the circuit under test. ASL begins with a top-level circuit declaration. This declaration defines the name of the circuit and uses the *in* and *out* keywords to define the inputs and outputs to the circuit[9]. An example circuit is given in Figure 2.5. Figure 2.5 A is a half-adder which takes in two inputs and outputs a sum (S) and carry bit (C), the corresponding ASL description is given in Figure 2.5 B. As can be seen from the figure the circuit declaration is started by the *ckt* keyword[9]. Each keyword in ASL is followed by a ':' character[9]. Following the *ckt*



(A)

Half-Adder Circuit

1. # Half-Adder ;
2. ckt: HF in: A B out: S C ;
3. xor: X1 in: A B out: S ;
4. and: A1 in: A B out: C ;

(B)

Figure 2.5: A Half-Adder Circuit

keyword is the name of the circuit in this case "HF". Following that, the primary inputs and outputs are declared using the *in* and *out* keywords. Each statement in ASL is terminated using a ';' character[9]. After the circuit statement the gate-level description of the circuit is given. The format of each gate is similar to the format of the circuit statement:

GATE: Name **IN**: In1 In2... InN **OUT**: Out1 Out2... OutN ;[9]

All gate declarations in ASL follow this simple syntax. All of the elementary logic gates (AND, OR, XOR, etc..) as well as the data flip-flop (DFF) and two input multiplexer are built-in to AUSIM and available to all circuits[9]. Table 2.2 shows each built-in gate and its inputs and outputs.

Table 2.2: Built-In AUSIM Gates[9]

AUSIM Keywords	
Gate	Example
AND	AND: a1 in: i1.. iN out: Z
OR	OR: o1 in: i1.. iN out: Z
NAND	NAND: na1 in: i1.. iN out: Z
NOR	NOR: no1 in: i1.. iN out: Z
NOT	NOT: n1 in: A out: nA
XOR	XOR: x1: in: i1... iN out: Z
MUX2	MUX2: m1 in: i1 i2 s1 out: Z
DFF	DFF: d1 in: CLK D out: Q

It is important to understand that custom gates can be implemented hierarchically and used in a similar syntax. To do this one must use the *subckt* command. While an ASL file can only have a single circuit declaration, it can have any number of sub-circuit declarations[9]. Each sub-circuit has its own set of top-level inputs and outputs and its own gate-level net-list. Once defined the name of the sub-circuit can be used as a gate and be instantiated elsewhere in the circuit description. These sub-circuit definitions can be used to define the behavior of more complex CMOS standard cell gates such as the OAI222 etc. For the BIST discussed in this thesis, an entire library of sub-circuits was created to support simulation. This library defines many of the complex CMOS logic gates and several technology specific standard cells. This library is discussed in more detail in Chapter 3.

2.3 BIST Architecture

The BIST architecture tested in this thesis is a mixed-signal BIST approach which uses Selective Spectrum Analysis (SSA) to test analog circuitry. SSA has many advantages when used for the measurement of analog spectrum. SSA benefits from being a largely digital approach and integrates well into a BIST environment[26]. In addition due to SSA measuring only a single frequency point at a time, the hardware overhead is much lower when compared to a more common approach such as FFT-BIST[27]. This same advantage can affect the test time required to analyze an analog spectrum; however, it can prove very advantageous when only a few frequencies are of interest in the analog spectrum[26]. In addition [24] discusses a number of optimizations in detail which when performed (some of which will be discussed in Section 2.3.2) can greatly reduce the time required to perform a measurement.

A block diagram of the basic architecture of the studied BIST SSA approach can be seen in Figure 2.6. This BIST architecture is capable of measuring a number of different analog characteristics including frequency response and linearity (also known as third-order interception point or IP3)[22]. In addition by sweeping through a frequency spectrum both

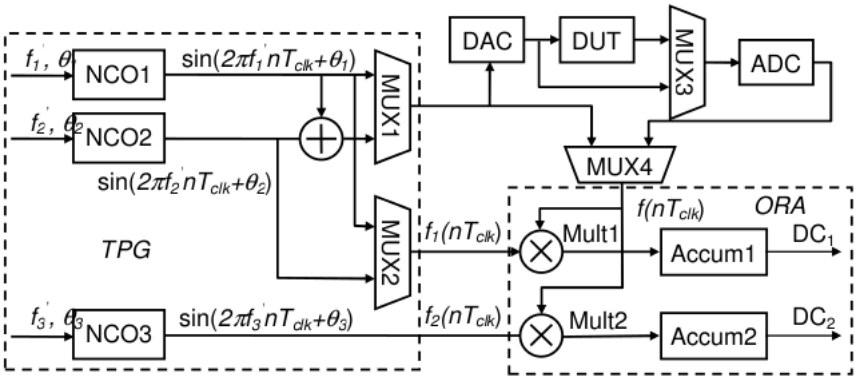


Figure 2.6: General BIST Architecture[20]

Signal-to-Noise Ratio (SNR) and Noise Figure[21] and be measured. The architecture in Figure 2.6 consists of a Direct Digital Synthesis (DDS) based TPG, Multiplier-Accumulator (MAC) based ORA, and test controller (not shown). Also not shown in the figure is the on-chip calculation circuit which will be discussed in detail in Section 2.3.3. The following sections will discuss each of the major BIST components in detail and how each test is performed so that the reader can developer a thorough understanding of the capabilities and uses of this BIST architecture.

2.3.1 DDS Based TPG

DDS is a popular technique for generating analog frequencies on chip. Compared to other techniques of frequency generation it offers the ability to produce very precisely controlled frequencies which can be rapidly manipulated and changed using digital inputs. In addition with adequate design considerations, DDS designs can have extreamly fine frequency and phase resolution[28]. The BIST TPG is DDS based consisting of three numerically controlled oscillators (NCOs) which utilize the existing DAC in the mixed-signal system to generate the analog waveforms for testing[20]. Each NCO utilizes a phase accumulator and sin/cos lookup table (LUT) to produce a frequency based on a supplied frequency word f. The supplied frequency word is accumulated each clock cycle by the phase accumulator. The

value of the phase accumulator is truncated and used as to lookup address to find the the appropriate sin or cos value. This value is then converted by the DAC into an analog system. This basic steps for this process are shown in Figure **??**. The resulting frequency generated by the NCO is determined by Equation 2.8, where f_{gen} is the generated frequency, f is the supplied frequency word, f_{clk} is the system clock frequency, and n is the number of bits int he frequency word. The maximum resolution of the DDS is directly related to the number of bits used for the frequency words, n.

$$f_{gen} = \frac{f \times f_{clk}}{2^n} \tag{2.8}$$

As shown in Figure 2.6, there are three NCOs. The first two NCOs are used to generated frequencies which stimulate the analog DUT. There are two possible configurations for these two NCOs when testing the analog DUT. In case one, only a single tone needs to be generated for testing. This case is used when performing tests such as frequency response (tests will be explained in more detail in Section 2.3.3. When only a single tone is necessary, both NCOs are supplied the same frequency word and produce the same frequency. This frequency is then selected and passed into the DUT. In the second case, two tones are generated. This case is used when performing tests such as a linearity test. In this case NCO_1 is assigned the first frequency word and NCO_2 is assigned the second. The output of these two NCOs is added together so that the two sin waves are super-imposed before being converted by the DAC[24]. The third NCO NCO_3 is used by the ORA. NCO_3 is slightly different than both NCO_1 and NCO_2 in that it produces both the sin and cos output for a given frequency word. These two outputs are used by the ORA to measure the in-phase and out-of-phase components of a frequency of interest (the ORA operation is explained in detail in the next section)[24]. These three NCOs allow for up to two separate frequencies to be generated simulatenously as well as any frequency of interest to be measured by the ORA. When

coupled with a test-controller, this architecture is very powerful and as will be discussed shortly can be used to measure several significant analog characteristics.

2.3.2 ORA Multiplier-Accumulators

After generating the the analog frequencies required to stimulate the DUT the output of the DUT is analyzed by the BIST ORA. The ORA consists of two multiplier-accumulators, DC_1 and DC_2[**?**]. By multiplying the response of the DUT by the cos wave output of NCO_3, DC_1 will accumulate the out-of-phase component of the DUT response at the frequency of interest, ω. Likewise by multiplying the response of the DUT by the sin wave output of NCO_3, DC_2 will accumulate the in-phase component of the DUT response at ω. The accumulator values of both DC_1 and DC_2 can be described in Equations 2.9 and 2.10. In these equations nT_{clk} represents the sampled output response of the DUT[27].

$$DC_1 = \sum_n f(nT_{clk}) * cos(\omega nT_{clk}) \tag{2.9}$$

$$DC_2 = \sum_n f(nT_{clk}) * sin(\omega nT_{clk}) \tag{2.10}$$

It has been shown in [24][27] and others that these calculations are similar to an FFT. Unlike the FFT the entire frequency domain is not computed simultaenously; instead, only a single frequency is measured for each accumulation[27]. When more than one frequency ω is of interest, each must be obtained through successive accumulations (each accumulation which measures a single frequency will from here on be referred to as a single measurement). The DC_1 and DC_2 results for a measurement can be used to obtain both the amplitude and phase of the signal at the measured frequency ω. The amplitude calculation is shown in Equation 2.11 and the phase calculation is shown in Equation 2.12[24].

$$A(\omega) = \sqrt{DC_1^2 + DC_2^2} \tag{2.11}$$

$$\Delta_\phi(\omega) = \tan^{-1} \frac{DC_2}{DC_1} \tag{2.12}$$

To obtain these measurements a hardware method must be used to calculate arc-tangent as used in Equation 2.12 and the square and square-root functions as used in Equation 2.11. These calculations are performed using the on-chip calculation circuitry and will be discussed in the next section.

Before discussing the calculation circuitry, it is important to understand how the amount of accumulation time required for a measurement is determined. The accumulation time is exceptionally important to achieving an accurate measurement due to potential AC calculation errors which are accumulated[24]. A large body of work was put into determining both the appropriate length of time to accumulate and a method for determining the length of time efficiently in hardware. [24] and [20] discuss in detail the complexity of stopping accumulation at the correct moment to reduce the error. As it is only pertinent to this thesis due to its relation to fault simulation time, it will only be briefly discussed here. More details can be found in [24].

As mentioned, there is the potential for errors to occurr in an SSA approach due to AC errors accumulating in the ORA[24]. One method of minimizing or eliminating these AC errors requires that the BIST accumulate for a very long period of time (referred to as free-run accumulation). Unfortunately the amount of time required for accumulation would be prohibitively high[20]. [24] discusses a method of reducing the required accumulation time while still retaining the accuracy of free-run accumulation. This method requires that accumulation occur for an exact number of clock cycles, determined by the integer multiple period (IMP) of the frequency being measured. Though there are many complications to determining the IMP in hardware, it will be discussed at a high-level so that the reader can understand how measurement time is affected.

As explained in [20], if accumulation is stopped at an IMP the AC error will be greatly reduced resulting in a very accurate measurement. The length of the test will subsequently be

determined by the effective number of bits used in the DDS (M_{eff}). Equation 2.13 shows how the effective number of bits is calculated, where M_{full} is the width of the phase accumulator and m is the bit position of the least significant '1' in the frequency word used[24]. The resulting number of accumulation clock cycles is determined by 2^{M_eff}.

$$M_{eff} = M_{full} - m \tag{2.13}$$

As an example, given a four-bit wide phase accumulator and the frequency word 1010 (or decimal 10), M_{eff} would be equal to $4-1$ or 3. This makes the number of clock cycles before a good IMP occures as 2^3 or 8. For large phase accumulators the number of clock cycles to accumulate can vary greatly depending on the frequency word chosen. In the studied BIST, 16-bit phase accumulators are used. In this case the difference between the shortest accumulation time of 2 clock cycles and than the longest accumulation time of 65535 clock cycles is much greater. Consequently frequency words should be carefully choosen so as to minimize test time[24].

The previous example is represents a single-tone test where only a single tone is of interest. For a multi-tone test (one where either two-tones are generated or where the measured tone is not the same as the generated tone) the IMP must be the common IMP of all frequency words[24]. In this case logically OR'ing all frequency words together then calculating the resulting M_eff number of bits will result in the number of clock cycles required for accumulation[24][20].

In hardware a similar method is used which directly calculates the number of clock cycles required for accumulation. After first OR'ing all frequency words together, an or-chain is used to determine the number of clock cycles to accumulate. Figure 2.7 shows this or-chain. As shown in the figure, if a logic '1' is in the LSB of any of the frequency words, it will result in the maximum number of clock cycles for the given phase accumulator width. Using the resulting clock cycle count, the BIST uses a clock cycle counter to determine when to

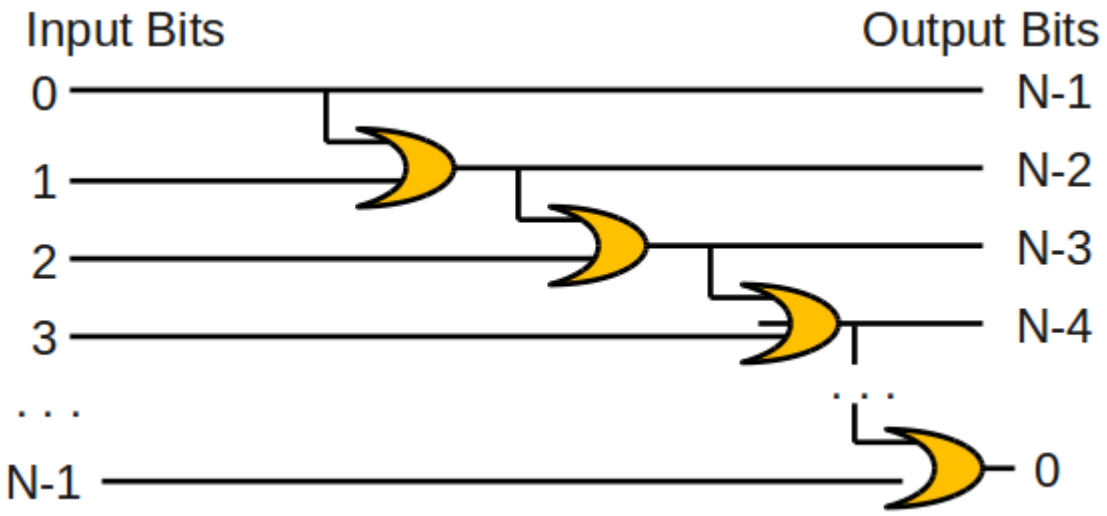


Figure 2.7: The OR Chain which calculates the number of clock cycles from the logical OR of all frequency words

stop accumulation on an IMP and minimize the error. Once accumulation has completed the calculation step is triggered and the results are handed off to the calculation circuit.

2.3.3 On-Chip Calculation Circuit

The ORA portion of the BIST accumulates the in-phase and out-of-phase components at the frequency of interest. The resulting DC_1 and DC_2 values store the in-phase and out-of-phase components of the frequency of interest. Using these values more advanced measurements of special analog characteristics such as linearity, SNR, and NF can be performed. To do this, DC_1 and DC_2 must be transformed mathmatically into the relative magnitude and phase using Equations 2.11 and 2.12[25]. Traditionally this can be performed off-chip using a mathmatical tool such as MATLAB; however, the BIST architecture discussed in this thesis uses an on-chip calculation circuit to perform both of these calculations and to perform multi-measurement tests on-chip[25]. First the calculation cordic, which converts the DC_1 and DC_2 values to the magnitude and phase values, will be discussed.

[25] discusses the original implementation of the calculation circuitry for the BIST approach. Shown in Figure 2.8 the original calculation circuit uses a custom CORDIC developed by [25] to perform the operations from Equations 2.11 and 2.12 and determine the magnitude and phase from the DC_1 and DC_2 values. The CORDIC algorithm, explained in depth in [25], uses successive approximation to perform the magnitude and phase calculations for the supplied DC_1 and DC_2 values. The CORDIC algorithm is a well known approach to calculating a number of linear, circular, and hyperbolic functions using inexpensive digital operations such as shift, add, and substraction[25]. Since it requires successive approximation, the calculation cordic requires several clock cycles to complete. At the end of the approximation the output of the CORDIC is the raw magnitude and phase values which can be read directly and used by the remaining calculation circuitry to perform high-order tests[25].

The remaining circuitry in the BIST is used for performing tests including measurements of the DUT linearity, spur search, SNR, and noise figure (NF). Before discussing each test in detail, it is important to understand that the output of the calculation will be a logrithmic result. Decibels (dB) are a common unit used to represent signal strength. To simplify the evaluation of test output, the calculation circuitry converts the output to decibels as described in Equation 2.14[25]. This operation is performed by the Logrithmic Translation Unit (LTU) shown at the output in Figure 2.8.

$$dB = 20 * log_{10}(INPUT) \tag{2.14}$$

Each test discussed will be represented in decibels by the calculation circuit.

With the help of the test controller the BIST can perform multi-measurement tests and consequently measure several different important analog characteristics. The first such test measures the linearity of the analog DUT. The linearity of a system is usually measured by analysis of the third-order inter-modulation point (IP3) using a two-tone test[29]. When

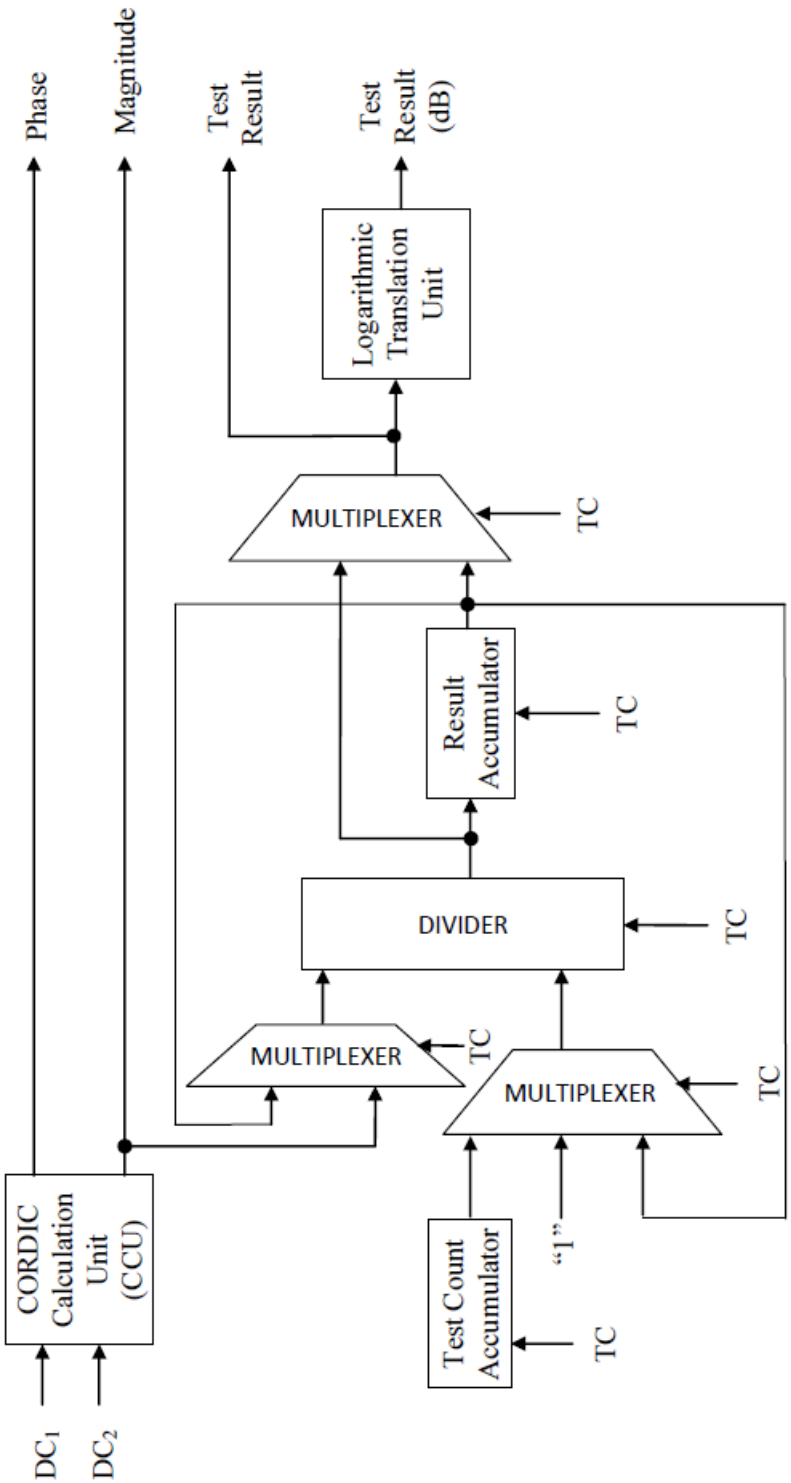


Figure 2.8: Calculation Circuit presented in [25]

two tones f_1 and f_2 are generated and used to stimulate the DUT, the resulting output frequency will consist of not only f_1 and f_2 but also the IP3 terms caused by the non-linearity of the DUT. The output spectrum is shown in Figure **??**. The linearity of a system is the difference in magnitude between the fundamental tone and the IP3 tone; this is often referred to as ΔP[29]. When performed in the BIST this requires two measurements. In both measurements, two frequencies are generated by the DDS f_1 and f_2. In the first test the measured frequency is set to f_2 and the ORA will accumulate the magnitude and phase of the fundamental frequency f_2. This result is stored in the calculation circuit while a second measurement is performed. The second measurement measures the IP3 tone at $2 * f_2 - f_1$[20][24]. The resulting magnitude is divided by the magnitude of the previous calculation and the result is the ΔP or linearity of the system.

The spur search test searches a given spectrum for the largest magnitude frequency. Due to TODO: why do spurs appear again other than crappy circuit?, it is beneficial to measure spurious tones in the output of the DUT. To perform a spur search a fundamental frequency is generated by the DDS, and a starting frequency f_s is used as the first frequency of interest. To begin the test the frequency of interest f is set to f_s and the magnitude measured. This magnitude is stored in the calculation circuitry while f is incremented by a predetermined amount f_{inc}. A new measurement will be made at the new frequency of interest f and the magnitude compared to the previously recorded magnitude. If the new magnitude is greater than the stored magnitude, the existing stored values are replaced with the new frequency word and magnitude. This process will repeat for the number of samples specified. In a special case, if $Samples = 0$ only a single frequency's magnitude is measured. This allows for the frequency response of the DUT to be measured at the given frequency. This test can be repeated accross the spectrum so that the frequency response of the DUT can be measured[29].

Signal-to-Noise ratio (SNR) and noise figure (NF) are important characteristics of an analog system. SNR measures the power of the signal of interest in comparison to the power

of the noise in the system over a bandwidth of interest. NF measures the amount of noise added to the system by the DUT by measuring the SNR at the input to the DUT SNR_{in} and comparing it to the the SNR at the output of the DUT SNR_{out}. The formulas for these calculations are shown in Equations 2.15 and 2.16 respectively[30].

$$SNR = \frac{Signal\ Power}{Average\ Noise\ Power} \tag{2.15}$$

$$NF = \frac{SNR_{in}}{SNR_{out}} \tag{2.16}$$

The BIST can directly perform these tests and measure both of these characteristics. When performing a SNR measurement the BIST first measures the magnitude of the fundamental signal and stores it. The next measurement begins at the starting frequency of the bandwidth of interest, f and measures the magnitude. This measurement is repeated for a predetermined number of samples and the result of each measurement is accumulated in the calculation circuit. For each repetition the measured frequency f is incremented by a constant f_{inc} specified by the user. After the specified number of samples have been taken, the accumulated noise power is divided by the number of samples to obtain an average noise power as shown in Equation 2.17, where f_s is the starting frequency, $Samples$ is the number of noise samples to take, f_{inc} is the constant frequency value by which f is incremented.

$$Average\ Noise\ Power = \frac{\sum_{f=f_s}^{f_s+Samples*f_{inc}} |f|}{Samples} \tag{2.17}$$

After the average noise power is calculated the original signal magnitude is divided by the noise power to obtain the SNR ratio shown in Equation 2.15. The calculation for NF is very similar. When performing a NF test the SNR_{in} is calculated by bypassing the DUT (using the previously mentioned bypass path) and then divided by the result of a second SNR test SNR_{out} calculated using the standard loopback path. The result is the NF ratio of the DUT.

While the general prinicpals and methods employed by the calculation circuit described in [25] are still used, improvements to this circuit have been made to the circuitry used by the BIST studied in this thesis. Like the previous calculation circuit it uses the same calculation CORDIC based on the design by [25] to convert the DC_1 and DC_2 values to their respective magnitude and phase representations. Unlike the previous circuit, the LTU has been moved to the front of the circuit so that all operations are performed in the logrithmic domain. The divider can then be replaced by a substractor as division is performed via substraction in the log domain. The new circuit allows for faster operation and a large reduction in area and power (approximiately 33%).

2.3.4 Summary

2.4 Thesis Statement

While these descriptions of the BIST architecture are not a comprehensive operating manual, they are provided to give a background of the circuit so that a basic understanding can be developed by the reader. The focus of the next chapters will shift to the actual fault simulations using the techniques discussed in Section 2.1 and away from the underlying architecture except where it relates to potential improvements in fault coverage and the coverage that is achieved. The goal of the techniques and methods discussed next as well as the goal of this thesis is to prove the effectiveness and necessity of the digital loopback path for achieving a high-fault coverage when testing a mixed-signal built-in self-test approach by using the studied BIST approach both as a benchmark and for context.

2.4.1 Interfacing with the BIST

Finally given an understanding of the calculation circuit, it is important to be familiar with how our BIST approach is controlled and observed (after all Section 1.1 clearly shows that controllability and observability directly influence the testability of a circuit). Excluding some run flags, the majority of the BIST is controlled via a SPI interface. When writing to

Bits	Name	Description
0-14	Frequency Word 1	First frequency of interest[a]
15-29	Frequency Word 2	Second frequency of interest
30-44	Frequency Word 3	Third frequency of interest
45-59	Frequency Word 4	Fourth frequency of interest
60-69	Samples	Number of samples to take[b]
70-85	Test Control Word	Determines the test to run and how to run it
86	Run Force	Makes the BIST run the test after it is loaded
87	Disable DDS Disable	Forces the DDS to reset at the start of a test

[a]The frequency words are used in different ways depending on the test being run
[b]Used for SNR and Spur Search to indicate the number of measurements to take

Table 2.3: SPI Write Command

the SPI interface there is a lower and upper 64-bit word that should be written with the data required for the test being run. Of the total 128 bits written to the SPI, only 88 of them are actually used by the BIST. Table 2.3 shows an overview of the SPI word. In addition the test control word is broken down in Table 2.4.

To read data out of the BIST the same SPI is used. To read data, a write must first occur with the read flag set and must include two address bits. The address bits are decoded by the test controller and correspond to four different 64-bit words which can be read out of the BIST each containing different calculation data related to the test. The values retrievable are shown in Table 2.5. In the context of fault simulation it is less important to focus on the values read and instead focus on how many values can be read in relation to the observability of the system. In the Chapter 3 and Chapter 4 this will be discussed in more detail as it relates to the tests run and potential improvements to the fault coverage by increasing controllability and observability.

In addition to the SPI IO, there are a few other important outputs used in simulation. First there is the DDS output which is the generated tone for the test being executed, there is also a 10-bit test result output which is updated to the final log value calculated when the test has completed. Finally, there is a done flag used to denote that the BIST has finished performing the test requested and that the results have been calculated.

Bits	Values	Description
0	Loopback Path	Allows bypassing the DAC-ADC pair to test the BIST only
1	Preset	Allows selection of preset tests*
2	Half-IMP	Causes the BIST to wait only half of an IMP length shortening the test time but sacrificing some accuracy
3	Noise Floor	Used to select the summation of the noise instead of the SNR result during and SNR test
4	Bypass Path	Allows bypassing the DUT to test the DAC-ADC path*
6:5	Test Mode	Select the test to be run
10:7	Attenuation	Attenuates the DAC output*
12:11	Read	An initial value to be loaded into the register when reading
13	Disable DDS	Disables the DDS producing no output
14	Disable ORA	Disables the ORA, putting it in reset mode*
15	Run Enable	Enables an external run signal*
		*Unused during fault simulation

Table 2.4: Test Control Word

Table 2.5: Read Address Values

Address	Values
00	DC_1
01	DC_2
10	Magnitude and Phase
11	Spur FW, Log Result, Noise Floor Value

3.0.2 Converting to ASL

In many cases behavioral models are written in a high-level hardware description language such as VESIC Hardware Description Language (VHDL) or Verilog. These languages allow a behavioral model to be developed of a circuit at a much higher level than ASL's gate-level description. This means that for large, complex circuits (such as our BIST approach) design in VHDL or Verilog is preferred. To aid in the simulation of larger circuits a tool was developed to convert a Verilog net-list to an ASL net-list. This allows a user to write a high-level behavioral description of the circuit, synthesis it down to a Verilog net-list using one of many different CAD tools, then convert it to ASL for fault simulation using the VerilogParser tool.

As a simple example the ISCAS '85 C17 benchmark circuit (Figure 3.1) was used to demonstrate the conversion process and the differences between a Verilog net-list and an ASL net-list. In Figure 3.1 A the behavioral model of the C17 circuit has been synthesized into a post-layout net-list in Verilog. In B the Verilog model has been converted to ASL. The most significant challenge converting from Verilog to ASL is the difference in the treatment of inputs and outputs of gates and circuits. In 3.1 A, line 7 the instantiation of the AO21 gate is seen. The gate's inputs and outputs are not designated separately. In addition the net names are attached to their respective gate IO names. In contrast, in 3.1 B, line 2[1] the same gate is instantiated in ASL. ASL uses a syntax that declares the inputs and outputs to a gate by position and not by name. For the translation to be successful, the inputs and

[1]The AO21 gate is a custom sub-circuit previously declared

```
module c17(gat1, gat2, gat3, gat6, gat7, gat22, gat23);
  input gat1, gat2, gat3, gat6, gat7;
  output gat22, gat23;
  wire gat1, gat2, gat3, gat6, gat7;
  wire gat22, gat23;
  wire n_0, n_1;
  AO21_B g58(.A1 (gat1), .A2 (gat3), .B (n_1), .Z (gat22));
  AO21_B g59(.A1 (n_0), .A2 (gat7), .B (n_1), .Z (gat23));
  AND2_B g60(.A (n_0), .B (gat2), .Z (n_1));
  NAND2_A g61(.A (gat3), .B (gat6), .Z (n_0));
endmodule
```

(A) Verilog Net-list

```
CKT: c17 IN: gat1 gat2 gat3 gat6 gat7 OUT: gat22 gat23 ;
AO21: g58 IN: gat1 gat3 n-1 OUT: gat22 ;
AO21: g59 IN: n-0 gat7 n-1 OUT: gat23 ;
AND: g60 IN: n-0 gat2 OUT: n-1 ;
NAND: g61 IN: gat3 gat6 OUT: n-0
```

(B) ASL Net-list

Figure 3.1: C17 Benchmark Circuit Net-list

outputs to the gates must be ordered correctly. Various techniques including lookup tables must be used to build the ASL correctly.

The VerilogParser tool performs the translation to ASL. It is written in Microsoft .NET C# 3.5. The language was chosen due to the author's familiarity with the language as well as the extensive standard library included in .NET which simplified the program considerably[23]. As a quick reference Figure 3.2 demonstrates how to use the tool on a Verilog net-list. The tool works by scanning the Verilog file for module definitions and parses the modules into an intermediate representation of the circuit. It then prompts the user to choose which module should be used as the top-level circuit before outputting the resulting ASL. Some improvements can be made to the converter such as automatic top-level module detection as well as the ability to convert VHDL net-lists; however, these features were not implemented due to time constraints. It is important to note that while the ASL generated

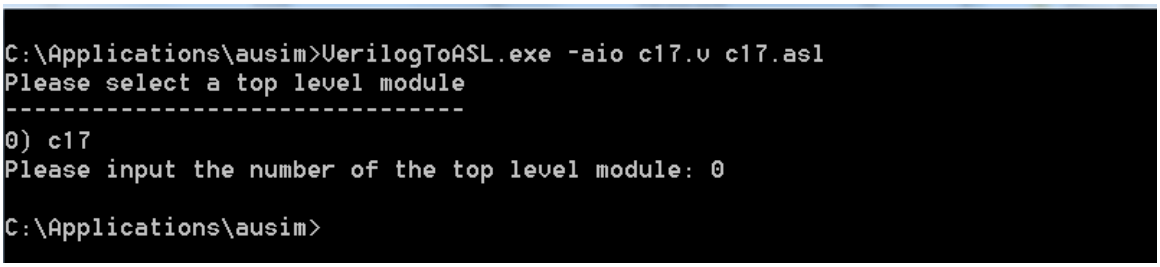


Figure 3.2: Using the Verilog to ASL tool to convert a circuit to ASL

will always be syntactically correct, the tool does not check for validity of the circuit. It is assumed any necessary sub-circuits that are used in the net-list have already been created and verified in ASL. For a more detailed analysis of the Verilog to ASL parser please refer to the Appendix **??**.

3.0.3 Generating Test Vectors using Vecgen

Vecgen is a program written to generate AUSIM vector files. It uses a straight forward command language to build a vector file of any length. It is written in Microsoft's .NET C#[23] language. Vecgen addresses the issue of large circuits by using a concept called frames.

The first line of a Vecgen file must be the GENERATE command which tells the generator how many frames(vectors) to generate and the number of primary inputs of the circuit. If all the user provides is the GENERATE command then Vecgen will create a vector file with the specified number of vectors, filled with all '0' characters as wide as the number of primary inputs in the circuit. This is the idea behind Vecgen, the vector each time is the same as the vector before it unless changed by the user in a frame. There are a number of commands which are available to manipulate the output using Vecgen (the full list is available in Appendix **??**). An example generation command file, which would create the vector file shown in Table **??**, is given in Table 3.0.3 showing the Vecgen format: In the case of the half-adder example, the generator is much more verbose and would not be preferred; however, for this example one can see how it would help in more complex generation. In

```
GENERATE 4 2

FRAME 1
BIT 0 1

FRAME 2
BIT 0 0
BIT 1 1

FRAME 3
BIT 0 1
```

Table 3.1: Command to generate the half-adder vector sequence

our example FRAME 0 is not modified since the first vector generated is always all logic 0's. FRAME 1 changes bit 0 to a logic 1 so that the resulting vector is now *01*. FRAME 2 changes bit 0 back to a logic 0 and changes bit 1 to a logic 1, resulting in *10*. Finally FRAME 3 updates bit 0 back to a logic 1 to obtain the vector *11*. Only at the times the output vector is changing does a FRAME need to be declared: if we were to generate ten frames instead of the four specified in the example, then the remaining six vectors would be automatically filled in and be the same *11* vector set by FRAME 3.

This is a very simple example, there are many more commands as shown in Appendix **??** that allow for much more powerful generation. Some commands such as RANGE affect multiple bits at once, others such as SERIALIZE or CLOCK work over multiple frames. One interesting command is the DECLARE command. The DECLARE command creates a reusable group of commands which can be called from any frame. This is especially useful if a certain sequence is required repeatedly during a test. An example sequence and its output are shown in Table 3.2. The sequence creates a clock on its MSB, then calls a function which counts up then counts back down. It then disables the clock and calls the counter function again. The generation in Table 3.2 is much more powerful than the previous example in Table 3.0.3 and demonstrates some of the power that the Vecgen program provides. All fault simulation tests for the BIST are designed in the Vecgen markup language for simplicity.

Table 3.2: Vecgen Example

Commands	1-15	16-24
GENERATE 24 3	100	001
	000	000
DECLARE counter	101	000
FRAME 0	001	001
COUNT 0 2 U	110	001
FRAME 8	010	010
COUNT 0 2 D	111	010
ENDDECLARE	011	011
	100	011
FRAME 0	000	
CLOCK 2	111	
CALL counter	011	
FRAME 16	110	
BIT 2 0	010	
CALL counter	101	

Chapter 4

Simulation Results

Chapter 5

Conclusions and Summary

Bibliography

[1] C. Stroud, *A Designer's Guide to Built-In Self-Test*. Kluwer Academic Publishers, 2002.

[2] ITRS 2009 (I am not sure how to cite this I need to ask you about it)

[3] Y. Zorian, "Testing the Monster Chip," *IEEE Spectrum*, vol. 37, no. 7, pp. 54-60, 1999.

[4] L. Ungar and T. Ambler, "Economics of Built-In Self-Test," Proc. *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 70-79, 2001.

[5] L. Wang, C. Stroud, N. Touba, Eds., *System On Chip Test Architectures*. Elsevier, 2008.

[6] L. Milor and V. Visvanathan, "Detection of Catastrophic Faults in Analog Integrated Circuits," Proc. *IEEE Transactions on Computered-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 2, pp. 114-130, 1989.

[7] C. Stroud, P. Karunaratna, and E. Bardley, "Digital Components for Built-In Self-Test of Analog Circuits," Proc. *IEEE 10th ASIC Conference and Exhibit*, pp. 47-51, 1997.

[8] C. Stroud, "AUSIM: Auburn University SIMulator - version 2.0," Dept. of Electrical & Computer Engineering, Auburn University, July 7, 2003.

[9] C. Stroud, "ASL: Auburn Simulation Language," Dept. of Electrical & Computer Engineering, Auburn University, July 7, 2003.

[10] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language," IEEE Std 1364-1995, 1996.

[11] L. Milor, V. Visvanathan, "Detection of Catastrophic Faults in Analog Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.8, no.2, pp.114-130, 1989.

[12] K. Arabi and B. Kaminska, "Oscillation-Test Strategy for Analog and Mixed-Signal Integrated Circuits," Proc. IEEE VLSI Test Symposium, 1996, pp. 476-482.

[13] V. Yarmolik, *Fault Diagnosis of Digital Circuits*. John Wiley Sons, 1990.

[14] B. Vinnakota, *Analog and Mixed-Signal Test*. Prentice Hall, 1998.

[15] M. Sachdev, *Defect Oriented Testing for CMOS Analog and Digital Circuits*. Kluwer Academic Pub, 1998.

[16] C. Stroud, J.Emmert, J. Bailey, D. Nickolic and K Chhor, "Bridging Fault Extraction from Physical Design Data for Manufacturing Test Development", Proc. *IEEE International Test Conference*, 2000.

[17] J.M. Emmert, C. Stroud, J.R. Bailey, "A New Bridging Fault Model For Accurate Fault Behavior," Proc. *IEEE Automatic Test Conference*, pp 481-485, 2000.

[18] D.G. Saab, I.N. Hajj, J.T. Rahmeh, "Parallel-Concurrent Fault Simulation," Proc. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp.298-301, 1989.

[19] C. Last, *Advanced Digital Design The Verilog HDL*. Prentice-Hall, 2004.

[20] J. Qin, C. Stroud, F. Dai, "Test Time of Multiplier/Accumulator Based Output Response Analyzer in Built-In Analog Functional Testing," Proc. *41st Southeastern Symposium on System Theory*, pp.363-367, 15-17 2009.

[21] J. Qin, C. Stroud, F. Dai, "Noise Figure Measurement Using Mixed-Signal BIST," Proc. *IEEE International Symposium on Circuits and Systems*, pp.2180-2183, 27-30 2007.

[22] F. Dai, C. Stroud, and D. Yang, "Automatic Linearity and Frequency Response Tests with Built-in Pattern Generator and Analyzer," *IEEE Transactions on VLSI Systems*, vol. 14, no. 6, pp. 561-572, 2006.

[23] Microsoft. The C# Language. Visual C# Developer Center. [Online] `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`.

[24] J. Qin, "Selective Spectrum Analysis (SSA) and Numerically Controlled Oscillator (NCO) in Mixed-Signal Built-In Self-Test," Doctoral Dissertation, Auburn University, 2010.

[25] G. Starr, "Built-in Self-Test for the Analysis of Mixed-Signal Systems," Master's Thesis, Auburn University, 2010.

[26] J. Qin, J. Cali, B. Dutton, G. Starr, F. Dai, C. Stroud, "Selective Spectrum Analysis for Analog Measurements," *IEEE Transactions on Industrial Electronics*, no.99, pp.1, 2011.

[27] J. Qin, C. Stroud, and F. Dai, "Phase Delay Measurement and Calibration in Built-In Analog Functional Testing, Proc. *IEEE Southeastern Symposium on System Theory*, pp. 145149, 2007.

[28] S. Qi, "Analog Circuit Testing using Built-In Direct-Digital Synthesis," Master's Thesis, Auburn University, 2005.

[29] F. Dai, C. Stroud, Y. Dayu, "Automatic linearity and frequency response tests with built-in pattern generator and analyzer," Proc. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* vol.14, no.6, pp.561-572, 2006.

[30] J. Qin, C. Stroud, F. Dai, "Noise Figure Measurement Using Mixed-Signal BIST," Proc. *IEEE International Symposium on Circuits and Systems*, pp.2180-2183, 2007.