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

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-accumulate (MAC) based output response analyzer (ORA) to stimulate and analyze the analog devices under test. This approach uses the digital-to-analog convertor (DAC) and the analog-to-digital convertor (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 cicuitry 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 throughly tested and functioning properly, this scan chain circuitry incures a potentially high area overhead and performance penalty. This thesis focues on using the existing BIST circuitry to test itself by utilizing a dedicated gitial loopback path. A set of test procedures is proposed and analyzed which can be used to determine a set of functional tests which provide a high effective fault coverage of the digital porition of the BIST. TO determine the effectiveness of these 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.

Abstract

Analog devices are inherently difficult to test; consequently, there are many different approaches to testing these devices. This thesis focuses on a Built-in Self-Test (BIST) approach which utilizes a Direct Digital Synthesizer (DDS) based test pattern generator (TPG) and a multiplier-accumulator (MAC) based output response analyzer to stimulate and analyze the analog devices under test. This approach uses the digital-to-analog convertor (DAC) and the analog-to-digital convertor (ADC), which typically already exist in mixed-signal circuits, to connect the digital BIST circuitry to the analog device(s) under test (DUT).

Previous work has 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 itself. Traditionally additional test circuitry such as scan chains would be added to the BIST circuitry to provide adequate fault coverage of the BIST. While ensuring that the the BIST circuitry is throughly tested and functioning properly, this circuitry incures a potentially high area overhead. 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 proposed which can be used to determine 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 procedures, an industrial sized mixed-signal BIST circuit is simulated and fault coverage results are obtained. Finally several improvements to the dedicated loopback path are proposed and simulated to analyze possible ways to improve the fault coverage of the BIST.

Acknowledgments

Put text of the acknowledgments here.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
ASL	Auburn Simulation Language
AUSIM	Auburn University SIMulator
BIST	Built-In Self-Test
CUT	Circuit Under Test
DAC	Digital to Analog Converter
DDS	Direct Digital Synthesis
DFT	Design For Testability
IC	Integrated Circuit
IM3	Third Order Inter-Modulation Point
IMP	Integer Multiple Period
IO	Input-Output
IP3	Third Order Interception Point
MAC	Multiplier Accumulator
NCO	Numerically Controlled Oscillator
NF	Noise Figure
ORA	Output Response Analyzer

SNR	Signal to Noise Ratio
SPI	Serial Peripheral Interface
SSA	Selective Spectrum Analysis
TPG	Test Pattern Generator
VCD	Value Change Dump
VHDL	VESIC Hardware Description Language

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 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 for both the digital and analog areas 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. While it is important at all of these three stages, 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 error-free operation. All of these different testing goals work to improve the device by reducing costs, improving relability, etc.

However, as chips become ever more complex it becomes more challenging and more expensive to test them[2].

As complexity increases, the traditional methods of testing these devices struggle to maintain an acceptable level of fault coverage (a concept which is explained more thoroughly in the next section). In [3] Zorian suggests the two main issues caused by increased complexity of chips as it relates to testing:

> "Progressively greater integrated circuit (IC) complexity raises several new problems and exacerbates others. One is the bandwidth gap between a chip's internal performance and the output capability of its package pins. Another is the increasing use of mixed circuitry on a chip.[3]"

In addition he also explains the consequences of increased transistor count in ICs:

> "The growth rate in 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]"

The issues he raises are very important to testing and related to very common testing paradigms, controllability and observability. To understand how these two issues play a large part in the complications of testing it is first important to understand how testing is performed and modeled on circuits.

1.2 The Basics of Testing

The basics of testing a circuit are similar at all levels of the product phase and are shown in Figure 1.1:

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

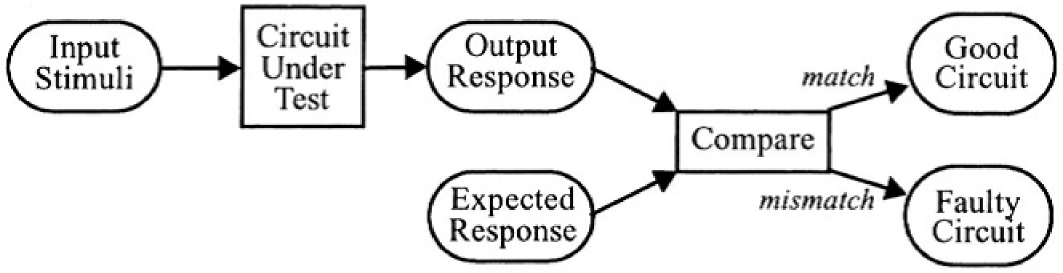


Figure 1.1: The Testing process[1]

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 involved in testing a fault. The testability of the circuit or device is directly related to the controllability and observability of the circuit[1]. This also explains Zorian's concern in [3]: the increasing amount of internal input-output IO in chips directly affects the observability and controllability of the chip. 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 is using external testing equipment. Unfortunately according to [3] external testing equipment cannot satisfactorily test todays very large IC devices for performance-related defects. As the speed of IC's continues to increase, testing IC's with external equipment will only become more difficult[3]. In addition Zorian examines the cost of external test equipment as becoming cost prohibitive and echoes the sentiments of the ITRS Roadmap from 1997 which indicates that

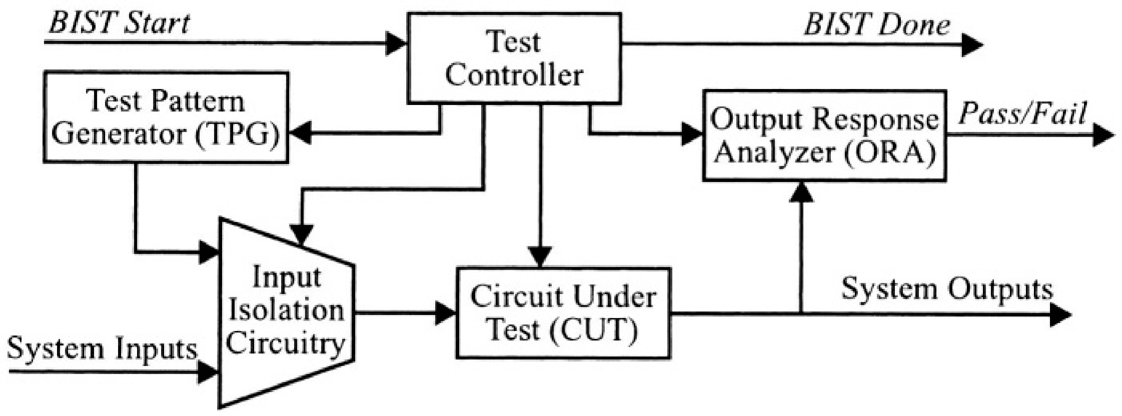


Figure 1.2: Simple BIST Architecture[1]

> "tester costs will reach up to \$20 million dollars in 2010 unless IC design incorporates more embedded test[s].[2]"

As an alternative to more expensive testing equipment, BIST describes the technique of designing a device to test itself[1].

1.3 Built-In Self-Test

Stroud 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 Figure 1.1 on itself. Stroud continues by discussing a simple BIST architecture detailed in Figure 1.2. This simple architecture consists of a Test Pattern Generator (TPG) which generates the input stimuli necessary to test the circuit. In addition there is an Output Response Analyzer or ORA which compares the output of the circuit to the expected output for a given input value. Finally there is some additional circuitry including input isolation circuitry used to isolate the CUT during testing as well as circuitry which controls the test during execution (Test Controller).

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:

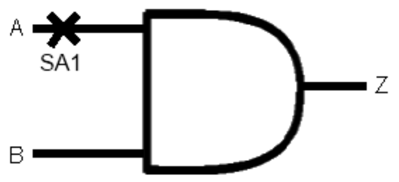
> “As the product develops from the IC to system level and its complexity increases, so does the complexity of successfully identifying a failure’s root cause. So it makes economic sense for system owners and perhaps system producers to implement BIST.[4]”

1.3.1 Digital Systems and Faults

The traditional BIST functions very well for digital devices (devices functioning at discrete ‘1’ and ‘0’ logic values). In a digital system faults (errors in the circuit) are characterized by a fault model. A common model used for digital circuits is the Gate-Level Stuck-At Fault Model. 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 what value is applied or expected. Figure 1.3 provides a truth table listing the expected output of an AND gate with its ‘A’ input stuck at ‘1’ (the notation used in Figure 1.3 is the traditional notation used to signify a fault for this model). The truth table given in Figure 1.3 demonstrates how a single fault can change the behavior of a gate often having a major impact on the overall behavior of the circuit. These differences in output

Advantages	**Disadvantages**
Vertical testability (wafter to system)	Area overhead
High Diagnostic resolution	Performance penalties
At-speed testing	Additional design time & effort
Reduced need for external 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	

Table 1.1: Advantages and Disadvantages of BIST[1]



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 1.3: AND gate stuck at 1[1]

caused by faults can be exploited to determine if a circuit is functioning correctly. Since digital systems always produce the same output for a given input, this makes simulation and detection of faults simpler and easy to implement via a Built-In Self-Test approach similar to Figure 1.2[1].

1.3.2 Analog Systems and Faults

Analog systems function differently than digital systems. Unlike a digital system which only have two discrete values, analog systems are continuous waveforms with multiple levels and voltages. In addition to the complexities of analog waveforms, analog components operate within a range of acceptable values, unlike digital components which operate at discrete levels[5]. This difference makes testing analog components for defects (defect-oriented testing) significantly more challenging. It requires complicated and time consuming simulation's such as Monte Carlo analysis to determine different component values for fault-free analog components[5]. Compounding this issue 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]."

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

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

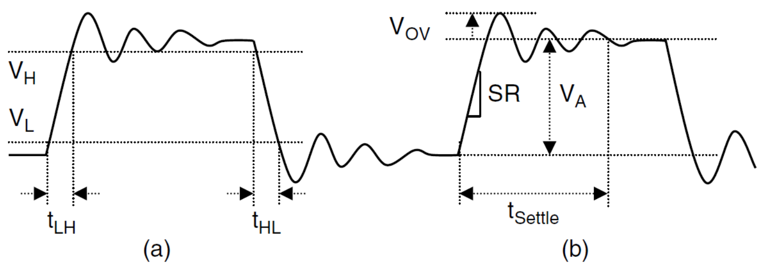


Figure 1.4: Digital and analog representations for a time domain response: (a) digital and (b) analog.[5]

Functional testing is achieved by performing a set of tests to determine if a system is operating correct as defined by its specifications. While it depends on the specifications for each device, an analog designer usually has many more important aspects that affect the output waveforms of his or her circuits than digital designers. According to [5],

> "whereas a digital designer is primarily concerned with logic one and zero, an analog designers cares about the detailed shape of the waveform and its corresponding spectrum.[5]"

Figure 1.4 shows an example of properties that might be of interest to an analog designer versus a digital designer. Common tests used when performing functional testing may include frequency response, linearity, and signal-to-noise ratio among many others[5].

1.3.3 Mixed Signal Testing

In a mixed-signal environment both digital and analog systems coexist and interact. Testing these systems is a challenge as it brings together the specific nuances of both types of environments together. While there are many different ways to test these systems, [1] defines a BIST approach to testing mixed-signal systems shown in Figure 1.5 This BIST system relies on the existing Digital to Analog Converter (DAC) and the existing Analog to Digital Converter (ADC) to perform functional tests on the analog circuitry of a mixed-signal system. One of the key advantages of this architecture is that it is largely digital

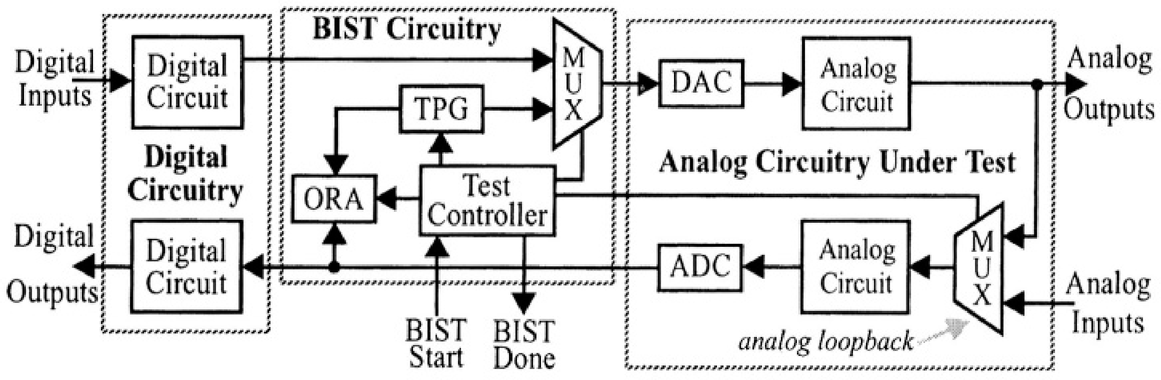


Figure 1.5: BIST approach for mixed-signal systems[1]

and thus can be integrated into the digital circuitry already in the system with minimal overhead. This prevents excess interference with the analog circuitry excluding two loop back multiplexers that facilitate the sending and retrieving of test values to and from the analog sub-system[1]. In Figure 1.5 there are only two multiplexers in the system, one to switch the inputs of the analog system to the TPG generated values and one to return the analog data back to the ORA for analysis. Additional multiplexers can be added to facilitate higher fault resolution. Figure 1.6 shows the addition of a second set of multiplexers allowing the BIST to resolve faults to either the first loop back path or the second[1]. There is no limit to the number of multiplexers that can be added to the system other than the increase in area overhead; however, our specific mixed-signal approach (which will be explained in more detail in Chapter 2) does add in several other loop back paths including a digital only path which facilitates the testing of the BIST itself and a path through the DAC and ADC which allows for the testing of both components as well as additional flexibility during system testing. The digital loop back path (the path connecting the BIST TPG directly to the ORA), is the focus of this thesis. This thesis looks to discover the practicality of using the BIST to test itself via the digital only path. It was shown in [7] that testing the BIST via the digital loop back path is an essential step to providing a sufficiently high fault coverage of mixed-signal systems. Figure 1.7 demonstrates the fault coverage achieved by

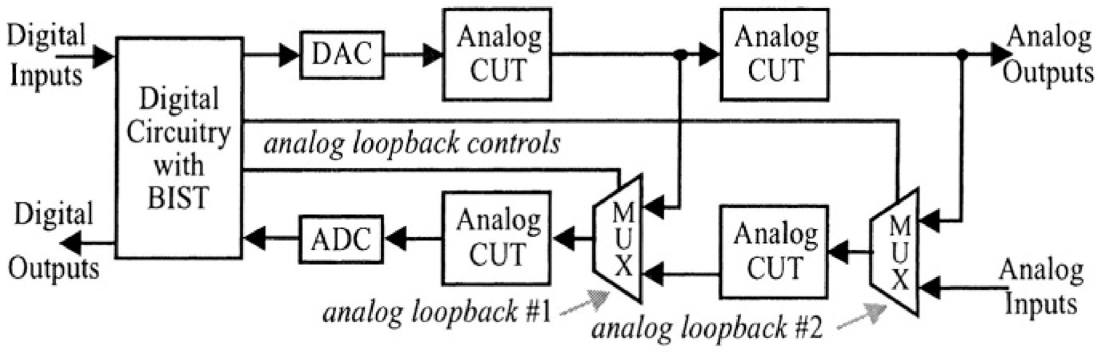


Figure 1.6: Multiple Loop Back Mixed Signal BIST[1]

[7] without using the digital loop back path to test the digital circuitry first. Due to the inherent acceptable variation in analog circuitry, digital values must be given an acceptable value range similar during testing. As this range increases, the fault coverage decreases (the double precision ORA can lessen this issue, but it too suffers some decline and also has a negative impact on area). As a result of [7]'s testing, a digital loop back path is used similar to Figure 1.8 to determine the maximum digital fault coverage achievable when using the BIST to test itself. To determine our fault coverage we use AUSIM, a fault simulator developed by Dr. Charles Stroud, to simulate the faults in our circuit and to determine the fault coverage achievable by our circuit for a given set of test vectors.

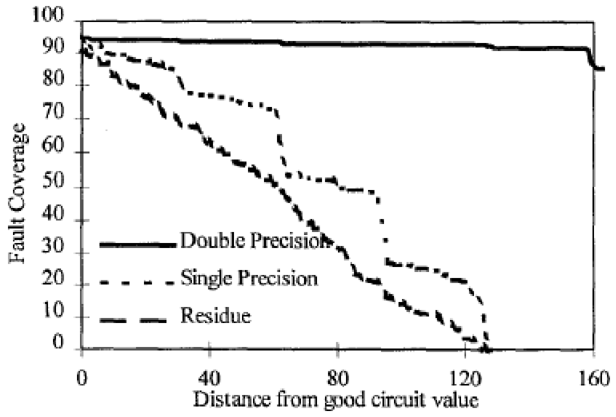


Figure 1.7: Fault Coverage v. Acceptable Value Range[7]

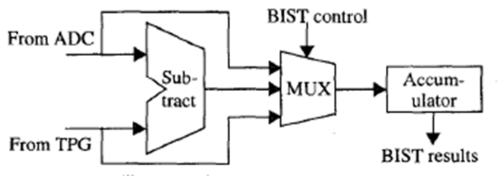


Figure 1.8: Digital Loop Back Multiplexer[1]

Using our fault simulations, we show that a digital loop back path is necessary to fully test the digital circuitry of the mixed-signal BIST and that by developing the appropriate set of a test vectors, a high fault coverage of the BIST circuitry can be achieved.

1.4 Thesis Statement

In the next chapter a more detailed explanation about fault simulation and of our specific mixed-signal testing approach is given. Building upon this explanation, Chapter III explores the main topic of this thesis: testing the actual mixed-signal testing circuitry to determine the level of fault coverage achievable. The results of this testing and an examination of the effectiveness of these tests are explained in Chapter IV, before a summary and conclusions are given in Chapter V.

Chapter 2

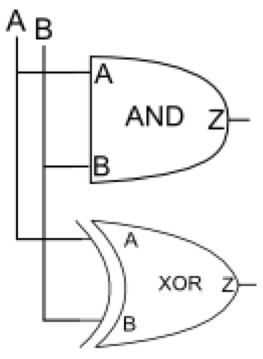
Background

2.1 Fault Simulations

Before discussing the tests performed on our BIST approach, it is important to have a thorough understanding of how fault simulations are performed. There are many different simulators and methods of performing fault simulations on a circuit. For our tests we choose the AUSIM fault simulator. Developed by Stroud, this simulator can perform logic simulations, as well as both stuck-at fault simulations and bridging fault simulations (faults which occur between wires) [8]. This section will discuss how to set-up and use this simulator as well as several custom tools used in conjunction with AUSIM to perform our testing.

2.1.1 Auburn Simulation Language

Before a circuit can be simulated it must be converted to 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. An example circuit is given in Figure 2.1. Figure 2.1 A is a half-adder which takes in two inputs and outputs a sum and carry bit, the corresponding ASL description is given in Figure 2.1 B. As can be seen from the figure the circuit declaration is started by the *ckt* keyword. Each keyword in ASL is followed by a ':' character. Following the *ckt* keyword is the name of the circuit in this case "HF". Following that, the primary inputs and outputs are declared using



(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 2.1: A Half-Adder circuit

the *in* and *out* keywords. Each statement in ASL is terminated using a ';' character. 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 circuits in ASL follow this simple syntax. The and, xor, and dff gates are examples of gates which are built into AUSIM. All of the basic logic gates as well as the D flip-flop and two input multiplexer are provided automatically. Table 2.1 shows each gate and its inputs and outputs.

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

Table 2.1: Built-in AUSIM Gates[9]

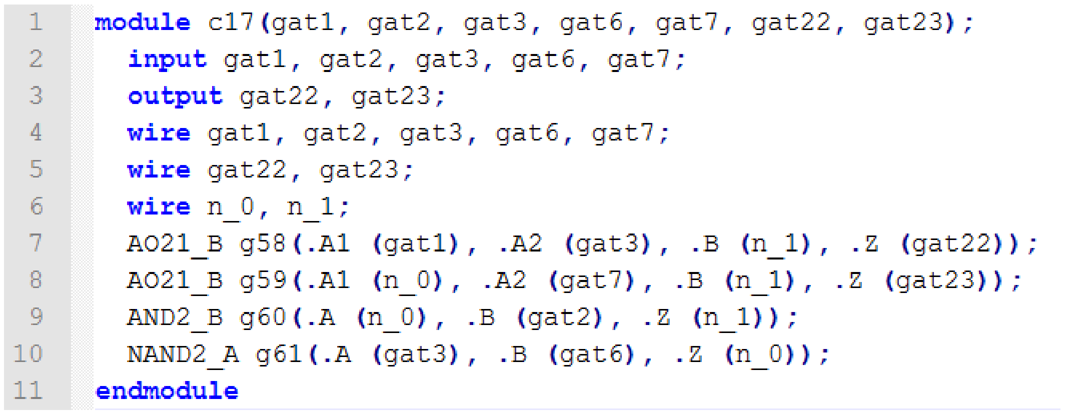
It is important to understand that custom gates can be implemented and used in a similar syntax. To do this one must use the *subckt* command. While an ASL file can only have a single circuit declaration, it can have any number of sub-circuit declarations[9]. Each sub-circuit has its own set of top-level 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.

2.1.2 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.

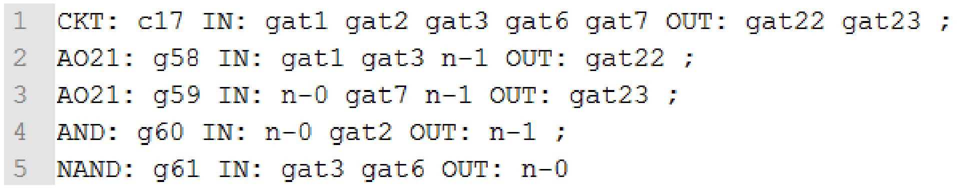
As a simple example the ISCAS '85 C17 benchmark circuit (Figure 2.2) was used to demonstrate the conversion process and the differences between a Verilog net-list and an ASL net-list. In Figure 2.2 A the behavioral model of the C17 circuit has been synthesized into a post-layout net-list in Verilog. 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 2.2 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 2.2 B, line 2[1] the same gate is instantiated in ASL. ASL uses a syntax that declares the inputs and outputs

[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



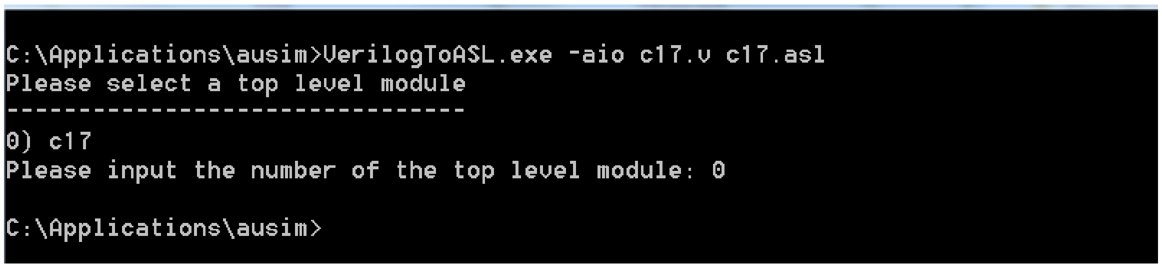
```
CKT: c17 IN: gat1 gat2 gat3 gat6 gat7 OUT: gat22 gat23 ;
AO21: g58 IN: gat1 gat3 n-1 OUT: gat22 ;
AO21: g59 IN: n-0 gat7 n-1 OUT: gat23 ;
AND: g60 IN: n-0 gat2 OUT: n-1 ;
NAND: g61 IN: gat3 gat6 OUT: n-0
```

(B) ASL Net-list

Figure 2.2: C17 Benchmark Circuit Net-list

to a gate by position and not by name. For the translation to be successful, the inputs and 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[14]. As a quick reference Figure 2.3 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



```
C:\Applications\ausim>VerilogToASL.exe -aio c17.v c17.asl
Please select a top level module
--------------------------------
0) c17
Please input the number of the top level module: 0

C:\Applications\ausim>
```

Figure 2.3: Using the Verilog to ASL tool to convert a circuit to ASL

implemented due to time constraints. It is important to note that while the ASL generated 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 A.

2.1.3 Commanding the Simulator

AUSIM can preform many different operations pertaining to the circuit. For our purposes we will focus on the steps leading upto and including logic and stuck-at fault simulation. The AUSIM simulator is executed by providing a command file. This command file is a simple file which is a sequential list of commands to be executed by the simulator. Each command and its required parameters are given on each new-line. Before exploring the commands available, it is important to understand how AUSIM locates and uses pertinent files during execution. Each important file has a unique file extension which designates which type of file it is. This can include files such as *.asl* for ASL files or *.vec* for Vector files (explained in the next section) among many others. In the usual case all of these files would be named the same prefix and only differ in their respective file extensions. This allows the use of the *default* command to specify the filename prefix for AUSIM to use when attempting to locate or creating new files. AUSIM does provide the ability to override the filename for individual files if needed[8]. Table 2.3 provides a list of common AUSIM commands and their respective descriptions. Each command file is provided to AUSIM during execution as a command-line argument. As a simple example, a command file for a circuit called counter which performed a logic simulation could look like the following: This file would tell AUSIM

```
default counter
proc
audit
simul8
```

Table 2.2: Example AUSIM command file

to look for files prefixed with counter (such as counter.asl, counter.vec etc.) and to process the circuit description, audit the circuit for area and performance, and finally perform a logic simulation using the circuit and a vector file. The *simul8* command is one of two modes of execution that are important for this thesis. The *simul8* command stands for "simulate" which performs a logic simulation using the circuit (the other mode is fault simulation which will be discussed in Section 2.1.8). Logic simulation applies a given set of inputs to the circuit for a given number of cycles and records the output. The inputs that are given are vector files and are discussed in the next section.

2.1.4 Test Vector Files

Once a circuit has been created in the ASL format the simulator needs to be given the appropriate input vectors to properly stimulate the circuit's inputs. These vectors are used for both logic and fault simulation (discussed in Section 2.1.8). Understanding the vector files and how to create them is essential to using AUSIM and properly testing circuits.

Command	Description
default prefix	Uses the prefix and the default suffix for each file
proc	Processes an asl file
audit	Performs area and performance analysis of circuit
simul8	Performs logic simulation
fltgen	Generates a fault list for the circuit
pftsim	Performs parallel fault simulation

Table 2.3: Common AUSIM Commands[8]

```
# Vector File ;
00
01
10
11
```

Table 2.4: Example Vector File For Half-Adder

Fortunately the vector file format is very simple. Generally a vector file should end in the *.vec* file extension and have the filename prefix that is set in the AUSIM command file[8]. In the vector file each line should include a test vector. Each test vector should be a binary sequence corresponding in length to the number of primary inputs in the ASL circuit. The number of test vectors should be equal to the length of the test the user wants to apply to the circuit. Each test vector that is applied will correspond to exactly one vector recording the outputs of the circuit after the circuit has settled (output is discussed in the next section). An example vector file for the previously mentioned half-adder (Figure 2.1) is shown in Table 2.4. This vector file applies four test vectors to the circuit.

This is great for simple circuits and short tests; however, for larger, complex circuits or long complex tests this can become difficult to use. AUSIM offers no way to generate large amounts of vectors (there is a random vector tool[8] but this is not always appropriate). To remedy this a tool was created to generate vector files from a simple command language.

2.1.5 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#[14] 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

```
GENERATE 4 2

FRAME 1
BIT 0 1

FRAME 2
BIT 0 0
BIT 1 1

FRAME 3
BIT 0 1
```

Table 2.5: Command to generate the half-adder vector sequence

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 B). An example generation command file, which would create the vector file shown in Table 2.4, is given in Table 2.5 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 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 B 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.

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	

Table 2.6: Vecgen Example

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 2.6. 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 2.6 is much more powerful than the previous example in Table 2.5 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.

2.1.6 Output Files

During logic simulation the output of the circuit is recorded into an out file. Like the vector file format, AUSIM's out file format is very straight forward. Each out file consists of a header created by AUSIM which labels each column of the file with the corresponding IO name. Following the header is the output of the logic simulation. Each output line consists of the test vector which was applied and the resulting output recorded from the circuit. An

```
#   AUSIM (2.7) Simulation Results ;
#   AB   SC ;
    00   00
    01   10
    10   10
    11   01
```

Table 2.7: Example Half-Adder Out File

example out file for the half-adder example from Figure 2.1 is shown in Figure 2.7. As shown in the Figure, the top consists of the previously mentioned header labeling each column with its IO name (if an IO name is more than a single letter it is written vertically with one letter on each line). Following the header is the four vectors from Table 2.4 and the resulting output from logic simulation. Like the vector file this format works well for small circuits and simple simulations; however, it can be a nightmare when verifying larger circuits. This is especially true of circuits with a large number of outputs or which use serial communication interfaces such as a Serial Peripheral Interface (SPI). For these more complicated interfaces a format which can be visualized greatly eases the difficulty involved in spotting easy to miss mistakes (such as an incorrect or invalid test vector) or invalid output. For these more complicated circuits (such as our BIST) a tool has been written to convert an AUSIM out file into a Value Change Dump or VCD.

2.1.7 Value Change Dump Format

The Value Change Dump 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 a standard format and is tailored towards storing digital simulation results in as small as a footprint as possible. 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.

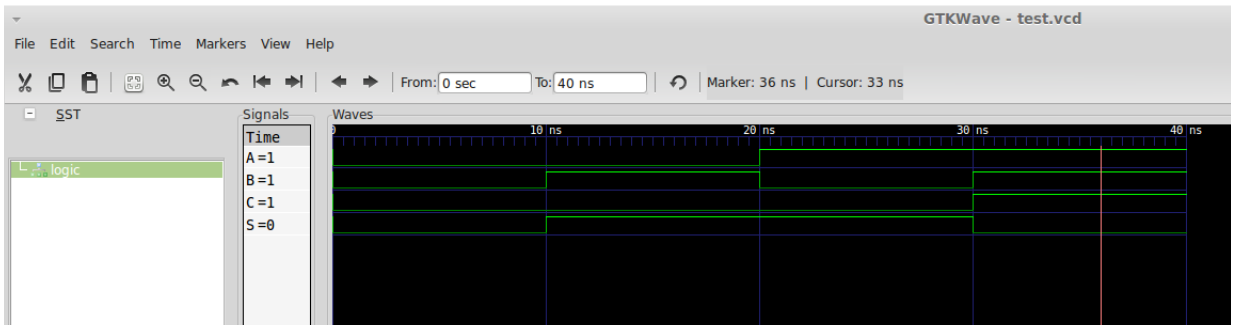


Figure 2.4: The VCD file visualized in the GTKWave Viewer

At its heart the VCD file stores simulation data in the same manner that the Vecgen markup defines test vectors. In contrast to the AUSIM out file format, the VCD format only stores changes in values[10]. While this makes the file less human readable, it makes it very easy to store large amounts of simulation data and for a viewer to parse and display the file. Table 2.8[2] displays the output file converted into VCD format. This conversion was performed using the Out2VCD program developed by the author. The program is very straight forward and uses the following syntax:

OUT2VCD [-g groups.txt][3] input.out output.vcd

In the example in Table 2.8, the VCD file is much larger than the AUSIM out file format; however, for long simulations with many inputs and outputs the VCD format is much more compact. Regardless of the file length, the key benefit to conversion to the VCD format is the ability to visualize AUSIM's output in one of a number of commercial viewer applications. For the example in Figure 2.4 GTKWave was used. 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 and zoomed.

[2]It is important to note 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

[3]The groups file is a list of IO prefixes which should be grouped together to form a bit vector. ie. if the file contained the prefix "color" then IO such as color1 color2 and color3 would be combined into color[2:0] in the VCD file

VCD File	Description
$date	The date section defines the day
Tuesday, December 07, 2010	which this file was generated
$end	
$version Generated For ASL2VCD 1.0	A comment detailing the
Alex Lusco	version of the generator for this file
$end	
$comment	
VCD File	An additional comment section detailing
$end	information about the file
$timescale 10ns **$end**	Defines the time scale for each value frame
$scope module logic **$end**	The Logic Module
$var wire 1 ! A **$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
$dumpvars	Sets the initial value for each input
0!	Each line IO must be set here
0"	
0#	
0$	
$end	
#1	The #N construct denotes a time step to apply the
1"	following value changes at In this case 10ns
1#	simulation time
#2	This time its at 20ns
1!	
0"	
#3	30ns
1"	
0#	
1$	
#4	40ns

Table 2.8: VCD File for Half-Adder Vectors

This functionality alone makes it much easier to verify the output of a complex simulation. This tool proved invaluable for verification of test vectors when testing the BIST.

2.1.8 Fault Simulation

Fault Simulations using AUSIM are achieved by adding the "pftsim" command to the AUSIM command file. When performing fault simulations AUSIM reads a list of faults from a *.flt* file which contains nets to be faulted[8]. The time required to complete the fault simulation is a direct function of both how many faults are in the fault list and the time required for a logic simulation. The "pftsim" command performs a parallel fault simulation which means that as many as thirty-two faults can be simulated at once. This means the lower-bound of fault simulation time is shown in Equation 2.1. This is important and is discussed in more detail in Chapter 3 where simulation time becomes a major issue for the large BIST circuit.

$$\frac{NumFaults}{32} \times LogicSimulationTime \approx FaultSimulationTime \tag{2.1}$$

When Fault Simulation is run, it faults each fault specified in the fault file and compares the output of the circuit against the previously performed logic simulation. If at any point there is a discrepancy between the good circuit output (from the logic simulation) and the simulated circuit with the fault that fault is put in the "detected list". If after completing the individual fault's simulation, no discrepancy is found then the fault is placed in the "undetected list". There is also a third possibility which is an artifact of fault simulation. If a fault causes an uninitialized output (an output that could be either a logic 0 or a logic 1) then it is said to be potentially detected since its detection will ultimately depend on which logic value that output is set too. The total percentage of faults detected is called the fault coverage and is determined by Equation 2.2 (assuming there is a 50% change of detecting potentially detected faults), where D is the number of detected faults, P is the number of potentially detected faults, and U is the number of undetected faults[1].

$$\frac{D + (.5 \times P)}{U} \tag{2.2}$$

If any of the undetected faults are determined to be undetectable, which is to say that there is no possible test vector or sequence of test vectors which can detect the fault, then it can be removed from the denominator which will increase the fault coverage percent, as shown in Equation 2.3 (where X is the number of undetectable faults). This proved important for the BIST as there were a number of faults where the undetectability was caused by the lack of a hard-coded logic value in the simulator (for example where Vdd was tied to a gate input by the synthesis tool).

$$\frac{D + (.5 \times P)}{U - X} \tag{2.3}$$

A number of scripts were developed to ease the handling of multiple simulations and the combining of multiple simulation outputs; however, discussion of these scripts is reserved for a Chapter 4 where they become important for parsing the BIST simulation Results.

2.2 BIST Architecture

Our BIST architecture is a mixed-signal BIST approach which uses Selective Spectrum Analysis (SSA) to test analog circuitry. A block diagram of the basic architecture can be seen in Figure 2.5. This architecture is capable of measuring a number of different analog characteristics including frequency response and third-order interception point (IP3)[13]. In addition by sweeping through a frequency spectrum both Signal to Noise Ratio (SNR) and Noise Figure[12] and be measured. As seen in Figure 2.5 the BIST consists of a Direct Digital Synthesis (DDS) based TPG, Multiplier Accumulator (MAC) based ORA, and test controller (not shown). The TPG consists of three numerically controlled oscillators (NCOs) and utilizes the existing DAC in the mixed-signal system to generate the analog waveforms

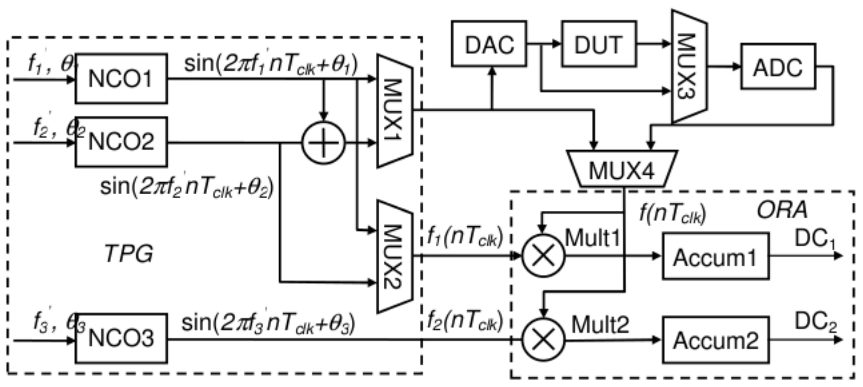


Figure 2.5: General BIST Architecture[11]

for testing[11]. Each NCO has a phase accumulator which is used to generate the output frequency[13]. By accumulating the phase word based on a given frequency word f each NCO will generate a frequency according to Equation 2.4[11]. The generated tones serve two purposes: one is to stimulate the analog CUT, and the second is to provide an in-phase and out-of-phase reference tone for the ORA[11]

$$f = \frac{f \times f_{clk}}{2^n} \tag{2.4}$$

The ORA architecture consists of two MACs. Each MAC receives the output from the CUT as well either the in-phase or out-of-phase signal from an NCO. Using this method the ORA is able to measure the amplitude and phase response at a given frequency[11]. Each test the BIST performs relies on this method of measurement. The tests and how they are run will be discussed in the next section.

In addition to the TPG and MAC-based ORA, there are a number of multiplexers in Figure 2.5. These multiplexers define different paths for sending and receiving signal to or from the CUT. As referenced in Section 1.3.3 these multiplexed loopback paths can be essential to achieving a high fault coverage when testing. In the architecture shown there are three loopback paths in the system. The first path is a path which passes a signal out through MUX1, through the DAC, through the analog CUT and returns it back through the ADC and into MUX4. This is the outermost path which stimulates the Analog CUT for testing. A second path exists similar to the first; however, instead of the output of the analog CUT, MUX3 instead selects the bypass path around the CUT. This bypass path be used to calibrate the BIST for any noise introduced by the DAC and ADC pair. The final path is the digital loopback path. In this path MUX4 selects the output from MUX3 directly. This path bypasses all analog circuitry including the ADC and DAC. As stated in Section 1.3.3, this is the path that is most important for our testing of the BIST. The digital loopback path does not introduce any analog error into the signal so when performing a "loopback test" the test will always result in the same accumulated value, no range of acceptable values is required[7].

2.2.1 Performing Tests

In a single measurement (where a measurement is defined as a single accumulation at a particular frequency) the BIST can measure the magnitude and phase of the CUT output at the measured frequency. To calculate these values Equation 2.5 is applied to the two resulting accumulation values one in-phase and one out-of-phase[15]. In the equation DC_1 and DC_2 are the output of the two in-phase and out-of-phase accumulations.

$$\begin{aligned} A(f) &= \sqrt{DC_1^2 + DC_2^2}, \\ \Delta\phi(f) &= -\tan^{-1}\tfrac{DC_2}{DC_1}. \end{aligned} \tag{2.5}$$

To perform more complex tests such as frequency response and IP3 (also called linearity) these measurements are performed repeatedly at different frequencies of interest and the results of each tests are used to calculate the result. As an example during a linearity measurement the CUT is driven by a two-tone signal which consists of two fundamental tones f_1 and f_2. Due to the CUT's nonlinearity, its output will consist of not only f_1 and f_2 but also the third order inter-modulation points (IM3)[11]. Figure 2.6 demonstrates what the input and resulting spectrum will look like[11]. The linearity (ΔP) of the CUT can be

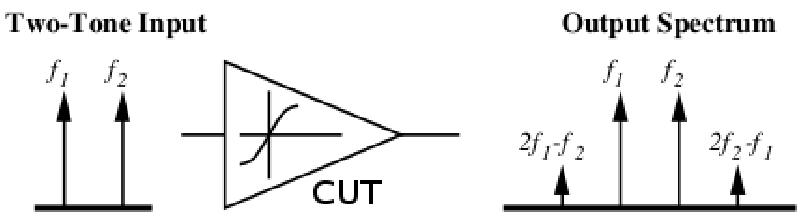


Figure 2.6: Two-tone test for IP3 measurement[11]

determined by the difference in dB between the fundamental frequency (either f_1 or f_2) and the IM3 tone[13]. To calculate this the BIST must take two measurements at either the two lower frequencies f_1 and $2 * f_1 - f_2$ or the two upper frequencies f_2 and $2 * f_2 - f_1$[11]. The result is the difference in the two measured magnitudes. The linearity test is just one example of how a measurement is made using this BIST architecture, in the next section more details will be discussed about how calculations are made on-chip and how the test controller is used to control and execute different BIST sequences. Before discussing the rest of the BIST architecture it is important to understand how the amount of time required for a measurement is determined. The accumulation is exceptionally important to achieving an accurate measurement due to potential AC calculation errors[15]. 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. [15] and [11] discuss in detail the complexity of stopping accumulation at the correct moment to reduce the error. As it is only pertinent to this thesis as it relates to fault simulation time, it will only be briefly discussed here as in relation to test-time.

According to [15], there is a potential for calculations errors in an SSA approach. In an effort to minimize or the errors the BIST approach must either accumulate for a long period of time (referred to as free-run accumulation) or the accumulation time must be stopped at an integer multiple period (IMP) of the frequency being measured. As explained in [15] a good IMP occurs when using a M_{full} width phase accumulator at $2^{M_{eff}}$ accumulation cycles where M_{eff} is calculated in Equation 2.6, when m is the bit position of the least significant

‘1’ in a frequency word[15].

$$M_{eff} = M_{full} - m \tag{2.6}$$

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 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 our BIST architecture with 16-bit phase accumulators, the difference between the shortest accumulation time of 2 clock cycles and than the longest accumulation time of 65535 clock cycles (which occurs when using any odd frequency word) is much greater. Another level of complexity occurs when doing a test which requires more than a single tone. For a multi-tone test the IMP must be the common IMP of all frequency words. In practice a simple method is used to determine the common IMP and subsequently the maximum number of clock cycles to accumulate. By logically OR’ing the frequency words together, our BIST approach uses an OR chain to directly determine the number of clock cycles to accumulate. Figure 2.7 shows the or-chain which calculates the number of clock cycles. The figure easily shows that a logic ‘1’ in the LSB or 0th bit of the input produces a ‘1’ on all the output bits (which corresponds to the maximum number of clock cycles for a given phase accumulator size). Using this clock cycle count, the BIST uses a clock cycle counter to determine when to stop accumulation on an IMP and minimize the error. Once accumulation has completed the calculation step is triggered and the results are handed off the calculation circuit.

2.2.2 Calculation Circuitry

As discussed previously when the BIST is used to measure a frequency, the result is the in-phase and out-of-phase values measured at the given frequency (DC_1 and DC_2). While these values contain the magnitude and phase information about the measured frequency, they are in a format which is not intuitive and requires post-processing to determine the

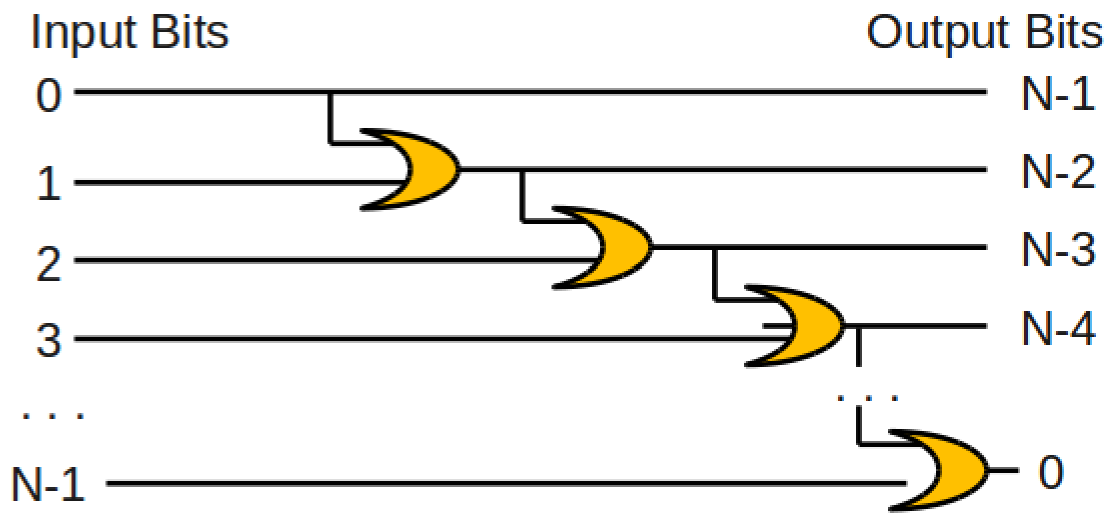


Figure 2.7: The OR Chain which calculates the number of clock cycles from the logical OR of all frequency words

result. To calculate the magnitude and phase the previously discussed equations from Equation 2.5 can be used. While this provides a simple step to converting the DC_1 and DC_2 values to a more intuitive format, it requires manual post-processing to be completed. The circuitry that will be discussed brings the processing of the DC_1 and DC_2 on-chip allowing the magnitude and phase to be determined and output directly. In addition, with the help of a sophisticated test controller the BIST is capable of taking more complex measurements (such as IM3 and SNR) and produce the output of those higher-order measurements directly.

[16] discusses the original implementation of the calculation circuitry for our BIST approach. Shown in Figure 2.8 the original calculation circuit uses a custom CORDIC developed by [16] to perform the operations from Equation 2.5 and determine the magnitude and phase from the DC_1 and DC_2 values. Also in this circuit is some additional circuitry to perform many of the higher level tests available to the BIST such as the previously discussed IM3 or linearity test. With the help of the test controller the calculation circuit presented by [16] can be used for IM3, spur search[4], signal-to-noise ratio, and noise figure

[4]A spur search consists of a measurement from a starting frequency to an ending frequency which returns the frequency and magnitude of the most powerful frequency measured in the given range

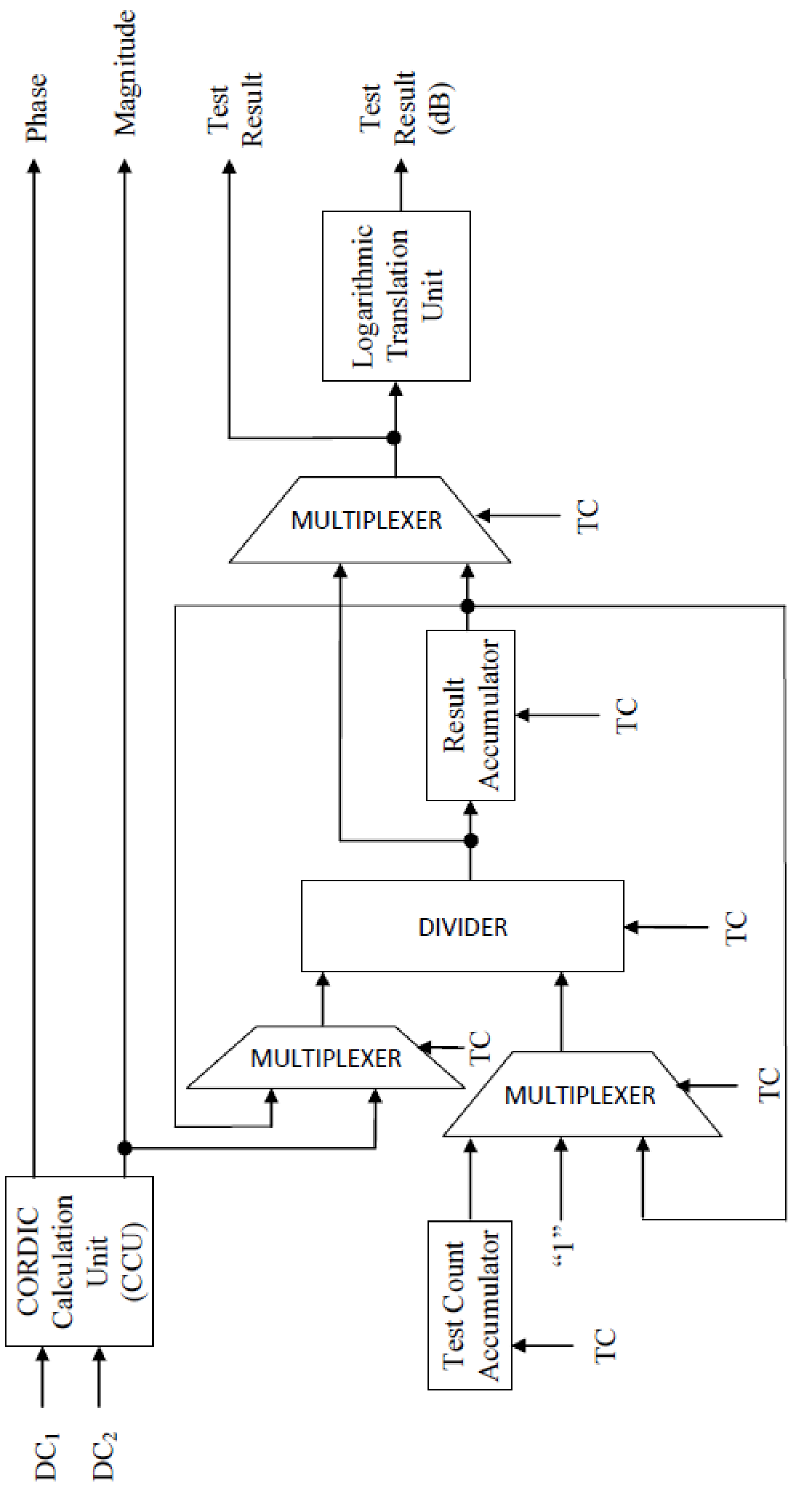


Figure 2.8: Calculation Circuit presented in [16]

(NF) tests. To do this multiple frequency measurements must be made by the BIST and stored in the calculation circuit. The final result from any such test is converted to decibels by the Logarithmic Translation Unit and made available to the system. The main feature of the calculation circuit is the divider. The divider performs several different functions depending on the test. In a IM3 test the divider is used to divide a current magnitude measurement by a previous measurement; likewise, for the SNR measurement it is used to divide the measured signal magnitude by the summation of the noise in the system[16]. The divider and accumulators are also used to find the average noise in the system which is used to calculate the SNR. While this circuit functions correctly and was able to solve the issue of on-chip calculations there are a number of issues which could be improved.

First the area of the calculation circuit was large, particularly the divider took up a large amount of chip area. Second the calculation circuit could not run at the same speed as the BIST circuitry. Our goal for our BIST implementation was to achieve a 1GHz operation frequency. Unfortunately the calculation circuit could not achieve this speed which necessitated two clock domains one for the BIST which could operate at upto 1GHz and one for the calculation circuit which could operate at upto 500 MHz. This made the interface circuitry between the BIST and the calculation circuit more complex and error prone. A third issue which occurred was the potential for divide by zero errors. Specifically when measuring the noise in the system, divide by zero errors could cause errors in the measurement if the magnitude measured by the BIST was small.

To overcome these problems a new calculation circuit was developed. Drawing upon the basic operation of the existing circuit the new circuit (Figure 2.9) removes the divider in favor of performing all operations in the logarithmic domain. Like the previous calculation circuit it uses a slightly modified calculation CORDIC based on the design by [16]. Unlike the previous circuit, the Logarithmic Translation Unit has been moved to the front of the circuit so that all inputs are converted their log value. This allows for the removal of the divider circuitry as the division required is performed by the subtraction of the two log values.

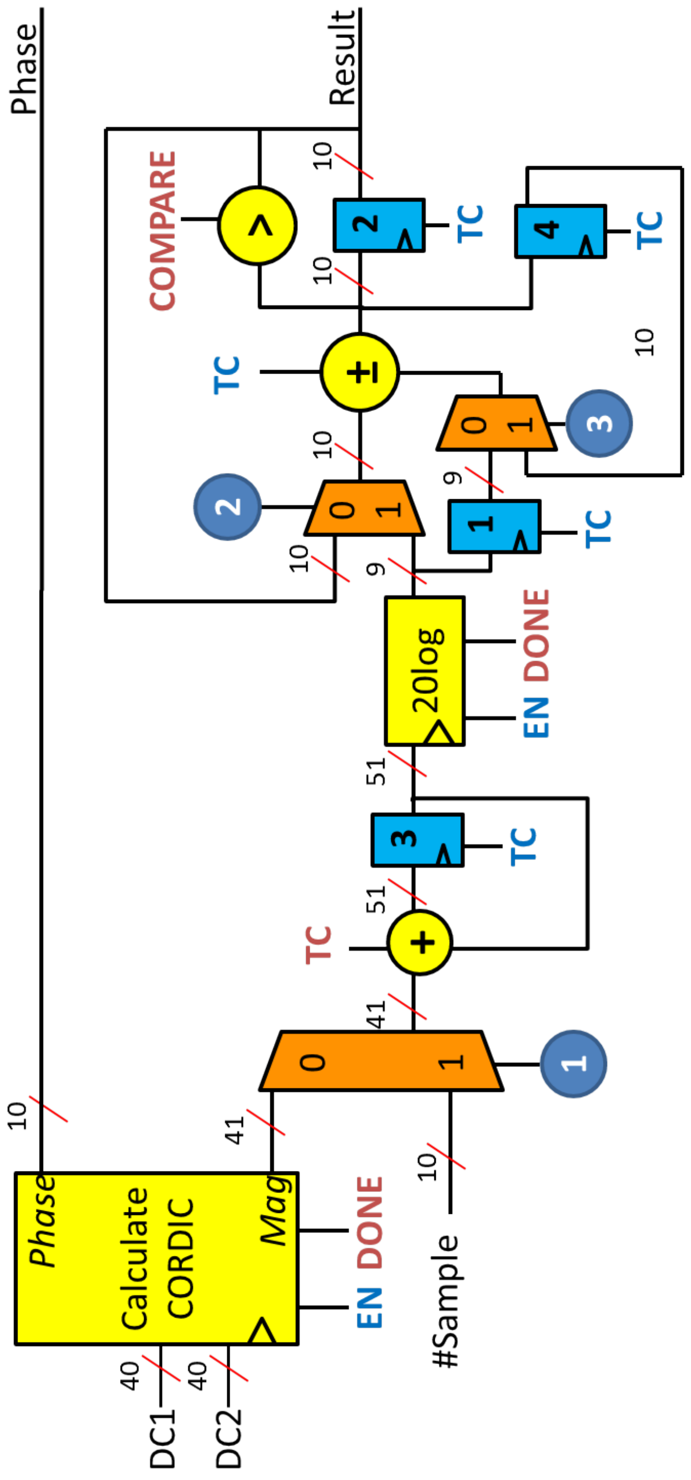


Figure 2.9: New Calculation Circuit

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 2.9: SPI Write Command

Consequently, there is no longer an opportunity to perform division by zero since no division operation is occurring. The new circuit is also able to operate at the same clock frequency as the BIST, eliminating the need for two clock domains. Finally due to the removal of the divider and other changes in the BIST, the calculation circuit was able to be shrunk to a much smaller size (approximately a 33% reduction in area and power).

2.2.3 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 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 2.9 shows an overview of the SPI word. In addition the test control word is broken down in Table 2.10.

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

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 CUT 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 2.10: Test Control Word

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

Table 2.11: Read Address Values

BIST each containing different calculation data related to the test. The values retrievable are shown in Table 2.11. 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.

2.2.4 Conclusion

While these descriptions of the BIST 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 and away from the underlying architecture except where it relates to potential improvements in fault coverage and the coverage that is achieved. Overall the goal of the techniques and methods discussed next are to show an effective method to testing mixed-signal built-in self-test systems using our BIST approach both as a benchmark and for context.

Chapter 3

Fault Simulations

Chapter 4

Simulation Results

Chapter 5

Conclusions and Summary

Bibliography

[1] Charles Stroud, *A Designer's Guide to Built-In Self-Test*, Vishwani D. Agrawal, Ed. Dordrecht: Kluwer Academic Publishers, 2002.

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

[3] Yervant Zorian, "Testing the monster chip," *Spectrum, IEEE*, vol. 37, no. 7, pp. 54-60, July 1999.

[4] Louis Ungar and Tony Ambler, "Economics of Built-In Self-Test," *Design & Test of Computers, IEEE*, vol. 18, no. 5, pp. 70-79, Sep-Oct 2001, `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=953274&isnumber=20606`.

[5] Laung-Terng Wang, Nur Touba, and Charles Stroud, Eds., *System On Chip Test Architectures*. Burlington, MA: Elsevier, 2008.

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

[7] Charles Stroud, Piyumani Karunaratna, and Eugene Bardley, "Digital Components for Built-In Self-Test of Analog Circuits," in *10th ASIC Conference and Exhibit*, Portland, 1997, pp. 47,51.

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

[9] Charles E. 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 , vol., no., pp.i, 1996 doi: 10.1109/IEEESTD.1996.81542 URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=803556&isnumber=12005`

[11] Jie Qin; Stroud, C.; Dai, F.; , "Test time of multiplier/accumulator based output response analyzer in built-in analog functional testing," System Theory, 2009. SSST 2009. 41st Southeastern Symposium on , vol., no., pp.363-367, 15-17 March 2009 doi: 10.1109/SSST.2009.4806795 URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4806795&isnumber=4806773`

[12] Jie Qin; Stroud, C.; Dai, F.; , ”Noise Figure Measurement Using Mixed-Signal BIST,” Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on , vol., no., pp.2180-2183, 27-30 May 2007 doi: 10.1109/ISCAS.2007.378606 URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4253104&isnumber=4252535`

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

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

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

[16] Joey’s Thesis

Appendix A

Verilog To ASL Parser

This appendix reviews the Verilog to ASL parser tool in more detail. The parser is 837 total lines of code, as such only select portions will be reviewed in this appendix.

TBD, expect 2-3 pages

Appendix B

Vecgen Commands

This appendix provides a list of all commands usable by the Vecgen program to create test vector files for AUSIM.

TBD, expect 3-4 pages