4 VERIFICATION PLAN

The verification plan is a specification for the verification effort. It is used to define what is first-time success, how a design is verified, and which testbenches are written[1]. This chapter addresses the description of a verification plan for the UART specified in chapter 2 and with the implementation plan defined in chapter 3. The verification plan makes use of suggestions written in *Writing Testbenches* and *Reuse Methodology Manual*[2]. The types of verification tests can comprise of compliance, corner case, random, real code, and regression testing.

In addition to the verification plan, this chapter provides a discussion on verification languages, general verification requirements for components, and the rationale for the selection of VHDL for this book. This material is included because the verification plan addresses the verification language, and there is a growing trend[3] in the migration toward the use of languages specifically designed for verification, rather than HDL designs.

---

1 *Writing testbenches, Functional verification of HDL models*, Janick Bergeron, Kluwer Academic Publishers 2000

2 *Reuse Methodology Manual for System-on-a-Chip, Second Edition*, Michael Keating and Pierre Bricaus, Kluwer Academic Publishers 1999

3 *Verification Guild, http://janick.bergeron.com/guild*

4.1 METHODOLOGIES

4.1.1 What is a Verification Plan

A verification plan is a document that defines the following:

1. **Tests or transactions applied to the design**. These tests are used to verify the design functional correctness as specified in the requirement specification. This includes tests at the top-level of the design as well as the subblocks.
2. **Testbench environment for the design-under-test**. This includes the definition of the verification language, the structure of the testbench, and special instructions. The structure encompasses the component models (at the interface level), packages (at the declaration or higher level), and file structures (if files are used).
3. **Validation environment for the design-under-test**. This includes the definition of the verifying and predicting software, the error reporting methods, and the types of errors detected.

4.1.2 Why a Verification Plan

A verification plan provides a strawman document that can be used by the unit-under-test (UUT) design community to identify, early in the project, how the design will be tested. Early mistakes in the verification approach can be identified and corrected. A byproduct of the verification plan exercise is the revisit on the validity and definition of the requirements. This enforces the process of verifying those requirements, thus helping in the identity of poorly specified or ambiguous requirements.

4.1.2.1 Testbench Style

Style is important in the design of the testbench because style guides the verification approaches and reuse of testbench models. Reuse is an important consideration in the design of the verification models because the testbench must adapt to the lifecycle of the unit-under-test. The UUT will typically undergo several design iterations, refinements, and even changes in requirements. During the review process of the verification plan, poor forms of testbench architectures will be flagged.

4.1.2.1.1 Poor Testbench styles[4]

Some examples of poor testbenches for re-use would include any of the following, ordered from worst to workable-but-ugly methods.

Vector Stimulus

---

[4] *Test Benches: The Dark Side of IP Reuse,* Gregg D. Lahti, SNUG San Jose 2000

A vector set is a group of 1's and 0's that contain input stimulus to the input pins and usually expected output from the output pins. The ability to understand what the vectors are doing (i.e., documentation of the stimulus) as well as the ability to modify the tests to incorporate bug fixes or design improvements is lost due to the low level format. The worst possible use of vectors is to create a "golden set" of test vectors by visually verifying the waveforms, saving the stimulus vector file, and then subsequently verifying any future simulations against the "golden" vector set. This method of visually verifying the waveforms is not only time consuming, but very prone to human error. Applying stimulus with A/C timing in mind generally requires specific knowledge of the interfaces being tested. Once this happens, the testbench becomes non-portable due to frequency constraints –i.e., the tests cannot function at a faster frequency since the vector set will need to get scaled differently or the tests cannot be modified to support a different A/C timing environment.

Assembly Language Code

A proprietary, low-level language like assembly code to drive a Bus Functional Model within a system-like environment with a processor, bus controller and program memory is next on the worst possible usage list. Assembly code generally means a lower level of abstraction of the test and limits the engineer in easily creating a functional test description to perform large, complex testing operations. The assembly language code testbench will work, but the effort to reuse it requires more overhead in terms of tools used to compile the assembly to object code and the effort to create the test. By using an assembly-code driven testbench, reusability gets limited to a platform-specific tool for code compiling, i.e., the compiler only works on a Sun Solaris ® 2.5.1 solution or worse, a Windows NT ® solution. The full-system environment used (processor in BFM form, bus controller, and program memory) also limits reuse since the entire environment must be re-created as the testbench in order to reuse the tests. Finally, assembly code is not portable across different micro processors/controllers. If an engineer created a special function I/O block like a USB ® controller and wrote the tests in assembly targeting an X86 ® microprocessor, the tests would need to be recoded if the block was to be reused for a System On Chip (SOC) solution using a StrongArm ® core. The use of specific-architecture assembly code forces the whole X86 ® system architecture to be emulated in the X86 ® system testbench to test one block. Once again, testbench reuse is now limited.

Scripts and Environments from Hell

EDA tools are never perfect, and no testing solution will always fit the requirements. To patch problems at hand, the engineer winds up creating a script-based workaround, usually in Perl. What can turn a testbench into a non-reusable

nightmare is the when engineers break away from an industry standard, widely used, HDL language (VHDL, Verilog, C/C++) to do the testing and create a whole environment of support scripts, test language scripts, and pre/post-processing scripts. It is difficult to create a modular testbench for a design when the testbench needs to incorporate a dozen 3000-line Perl scripts relying on many environment variables, hardcoded paths, and a chain of scripts calling more scripts. It also turns into a support nightmare when the script and environment are ill documented and the engineer is no longer working in the department or company. Engineers like writing solution scripts, but commenting and documentation is usually sacrificed for quick implementation and schedule time.

4.1.2.1.2 Good Testbench styles

A good testbench design style has, at a minimum, the following characteristics:

1. The resultant code is readable and maintainable.
2. Code is written in an approved, portable, open, modern, and preferably object oriented language.
3. Code is abstracted to as high of levels as possible. Thus, instead of "waveforms", code must address the "transactions" or "tasks" that are then transformed into waveforms by subprograms, methods, or server component models. A transaction identifies the parameterized task that must be performed on the UUT. For example, a WRITE transaction would include an address and data. The waveforms used in the protocol to activate the WRITE (e.g., chip selects, write enables) are described in another structure.
4. The verifier model has knowledge of the transactions asserted on the UUT, and makes use of that information in the detection of errors.

4.1.3 Verification Languages

Studies on engineering design efforts have shown that more engineering time is spent validating than writing an RTL description and synthesis. There is at least a 1:1 validation to design engineering task ratio (Figure 4.1.3), and in some cases more of a ratio. Because of this heavy verification effort, several EDA vendors have introduced new proprietary verification languages such as Synopsys[5] *Vera-HVL*™ Hardware Verification Language, Verisity's *Specman Elite*™[6], and Chronology *QuickBench/Rave*[7]. Cadence Design Systems is making its *TestBuilder testbench class library*[8] available using open source licensing, thus

5 http://www.synopsys.com/

6 http://www.verisity.com/html/specmanelite.html

7 http://www.chronology.com/

8 http://www.testbuilder.net

allowing designers, IP developers and EDA vendors to develop interoperable testbenches for chip or system design verification. These languages are marketed as **verification languages** designed to provide the necessary abstraction level to develop reliable test environments for all aspects of verification: automatic generation of functional tests, data and temporal checking, functional coverage analysis, and HDL simulation control. These verification languages typically implementation object oriented programming methodologies to enhance the ability to work with complicated designs and sophisticated testbenches.

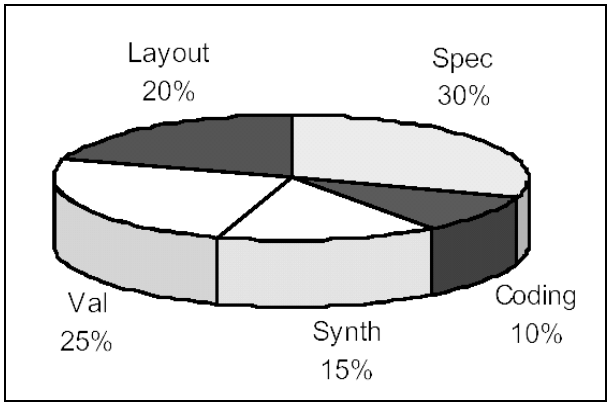

Figure 4.1.2 Breakdown of Engineering Effort[9]

Verification languages are beginning to make an impact on how designs are verified. People who have used them praise the efficiency of those tools[10]. Below is a quick overview of Spec-*Based* Verification[11], one of the commercially available verification languages.

"*Spec-based verification is an emerging methodology for functional verification that solves many of the problems design and verification engineers encounter with today's methodologies. This is done by capturing the rules embodied in the specifications (design/interface/functional test plan) in an executable form. An effective application of this methodology provides four essential capabilities to help break through the verification bottleneck:*

1. *Automates the verification process, reducing by as much as four times the amount of manual work needed to develop the verification environment and tests;*

9 Test Benches: The Dark Side of IP Reuse, Gregg D. Lahti, SNUG San Jose 2000

10 see http://janick.bergeron.com/guild

11 From http://www.verisity.com/

2. *Increases product quality by focusing the verification effort to areas of new functional coverage and by enabling the discovery of bugs not anticipated in the functional test plan;*
3. *Provides functional coverage analysis capabilities to help measure the progress and completeness of the verification effort;*
4. *Raises the level of abstraction used to describe the environment and tests from the RTL level to the specification level, capturing the rules defined in the specs in a declarative form and automatically ensuring conformance to these rules*."

Key benefits provided by *VERA*[12] Testbench Automation for Functional Verification include:

1. *Reduce verification time with automated testbench creation and analysis*
2. *Create modular, re-useable testbenches with VERA HVL -- the high-level language optimized for verification*
3. *Use the same testbench for VHDL and Verilog HDL designs*
4. *Perform thorough coverage analysis of even difficult corner cases*
5. *Increase simulation efficiency with closed-loop reactive tests*
6. *Increase simulation throughput with distributed processing capability*
7. *Do full system simulation through tight integration with Synopsys' industry-leading Synopsys Eaglei(R) HW/SW co-verification environment, and SmartModel(R) library*

Cadence's document on *Creating a C++ Library for Test Bench Authoring, Testbuilder*[8] states that *to support test bench authoring in C++, we have encapsulated three sets of concepts in a library: hardware concepts, testbench concepts, and transaction concepts. The resulting library provides an easy-to-use interface for writing test benches in C++, with transparent connection to an HDL simulator. Significant productivity gain in creating reusable benches and in debugging simulation runs have been achieved.*

The above information was intended only as a very brief introduction to verification languages. The reader is invited to get more information on that topic from the web.

12 from http://www.synopsys.com

This author's view of verification languages is as follows:

1. **Tools are not methodologies, yet methodologies may use tools to help in the implementation of the methodology.** A hammer is not a methodology in building a house, but a hammer is a tool used to build a house.

2. **Tools, by themselves, do not guarantee quality of work. Yet, tools may enhance the quality of work in the hands of a good artisan.** A high quality, state-of-the-art electric saw does not guarantee that a house will be framed correctly. However, in the hands of a good framer, that saw definitely helps.

3. **A good methodology with low technology tools is better than a poor methodology with high technology tools**. A house with a good building process can be built with low technology tools. However, a poor building process with high technology tools will not yield a good product.

4. **Verification languages are tools. By themselves, verification languages are not the panacea to verifying that a design is correct.**

5. **Verification languages can be very beneficial when mixed with a good methodology and in the hands of good craft persons.** This would be like building a house with an approved and reliable process, with advanced tools, and excellent craft persons.

6. **Users need to tradeoff the benefits of verification languages versus the costs associated with those tools, including the tool purchase/lease, training, and manpower for the verification specialists.** With qualified verification specialists, the manpower should be a lesser effort than if the verification were done in HDL. However, resource allocation may be an issue.

This book will use VHDL as the verification language because it is an open language (IEEE Standard 1076.6). VHDL provides powerful data and HDL constructs applicable to verification. The use of an open language for components provides greater portability for the verification and regression models. For this author, another reason for using VHDL is also economics, with access to VHDL tools (compilers and simulators), and the unrestricted freedom to publish code written in this open verification language. Good testbench design practices with reuse in-mind are applied, and could be migrated to other languages. This will include a self-checking testbench that is easily modifiable and well documented**. The concepts of verification are generic, and not language oriented.**

4.2 VERIFICATION PLAN

Header page

VERIFICATION PLAN FOR ASYNCHRONOUS OR SYNCHRONOUS 8 TO 32 BIT Universal Asynchronous Receiver/Transmitter

Pertinent logistics data about the requirements.

Document #: 01s
Release Date: __/__/__
Revision Number: _____
Revision Date: __/__/__

Conform to company policies and style

Originator
Name:
Phone:
email:

Approved:
Name:
Phone:
email:

Revisions History:
Date:
Version:
Author:
Description:

…

Note: The Header page will vary with each organization because of different needs. For example, a reviewer list (with name and signature only) may be more appropriate that a single "approved" entry. This page is a placeholder for a header page, and is not meant to represent an absolute format.

The numbering system for the verification plan starts at 1.0 because it is intended to represent a stand-alone document. Therefore, it does not follow the chapter numbering system.

1. SCOPE

1.1 Scope

Concise abstract of the coverage of the testplan

This document establishes the verification plan for the UART design specified in the requirement specification. It identifies the features to be tested, the test cases, the expected responses, and the methods of test case application and verification.

Target audience

The verification plan is primarily targeted for component developers, IP integrators, and system OEMs.

1.2 Purpose

Defining the verification plan often uncovers misunderstandings in the original requirements

The verification plan provides a definition of the testbench and verification environment, test sequences, application of test cases, and verification approaches for the Universal Asynchronous Receiver/Transmitter (UART) design as specified in the requirement specification number __01, and in the implementation specification number __01.

this plan is not only to provide an outline on how the component will be tested, but also to provide a strawman document that can be scrutinized by other design and system engineers to refine the verification approach.

1.3 Classification

This document defines the test methods for a hardware design.

2 DEFINITIONS

2.1 BFM

A Bus Functional Model that emulates the operation of an interface (i.e., the bus), but not necessarily the internal operation of the interface.

2.2 Client

An interface that is responsible for the definition and creation of the transactions to be asserted onto the UUT.

2.3 Transaction

Tasks and parameters that need to be executed. An example of a transaction would be a WRITE at a specified ADDRESS, with specified DATA, onto a specific interface, with specific FAULT modes, and at a specific time.

2.4 Server

A model or process that is responsible for executing the transaction issued by a

client. The server provides the appropriate waveforms onto the BFM interface. The server may alert the client of the completion of a requested transaction. The server may also be involved in the collection of data from the bus, and in the offering of this collected data to a client, typically in the form of a record.

3. APPLICABLE DOCUMENTS

3.1 Government Documents

None.

3.2 Non-government Documents

Document #: ____01, Requirement Specification for an Asynchronous Or Synchronous 8 To 32 Bit Universal Asynchronous Receiver/Transmitter

3.3 Executable specifications

None.

3.4 Reference Sources

1. *VHDL Coding Styles and Methodologies*, *2nd Edition*, Ben Cohen, KAP, 1999.
2. *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic Publishers (2000), Janick Bergeron, http://janick.bergeron.com
3. *Reuse Methodology Manual (RMM), 2nd Edition*, Kluwer Academic Publishers, 1999, Michael Keating and Pierre Bricaud.

4. COMPLIANCE PLAN

VHDL will be used as the verification language because it is an open language (IEEE Standard 1076.6). This plan consists of the following:

- Feature extraction and test strategy
- Test application approach for the UART and its subblocks
- Test verification approach

4.1 Feature Extraction and Test Strategy

Features are implicitly or explicitly defined in the requirements specification.

The design features are extracted from the requirement specification. For each feature of the design, a directed test strategy is recognized, and a test sequence is identified. A verification criterion for each of the design feature is documented. This feature definition, test strategy, test sequence, and verification criteria forms the basis of the functional verification plan. Table 4.1 summarizes the feature extraction and verification criteria for the functional requirements.

For corner testing, pseudo-random transmit and receive transactions will be simulated to emulate a UART in a system environment. The CPU will perform the following transactions at pseudo-random intervals:

1. Write transmit messages
2. Respond to transmit and receive interrupts by reading the PIR registers

The testbench environment will send receive-data at pseudo-random intervals.

Table 4.1 Feature Extraction and Verification Criteria

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
1	**Fixed Parameterization** - Word size - Buffer Depth - Buffer Almost Empty - Buffer Almost Full - Synchronous/ Asynchronous Mode - Instantiation transmit function - Instantiation receive function	8.1	1	**Configuration Setup** 8 bits/word 4 1 3 asynchronous transmit function instantiated receive function instantiated	Design compiles and elaborates correctly. Configuration to be used in testcases
	With test configuration #1. DO tests #2 thru #25				
2	**RESET** . UART to be in idle state, all software visible registers to be reset, no interrupt outputs	5.1.2.7 8.2.3 8.2.5	1	- Resetn = 0, - Resetn = 1 after 1 cycle - READ 01 -- *RCV PIR* - READ 10 – *XMT PIR*	No Interrupt outputs □□(7..0) = □□□□□□□□ □□(7..0) = □□□□□□□□

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
3	**Modem Status** . I/O BFM to Toggle modem status: *RINn CTSn DSRn DCDn* . CPU to read data . I/O BFM to Toggle modem status: *RINn CTSn DSRn DCDn* . CPU to read data	8.2.1	1	- Set RINn CTSn DSRn DCDn = 0000 - READ 00 -- *Modem status* - Set RINn CTSn DSRn DCDn = 0001 - READ 00 -- *Modem status* - Set RINn CTSn DSRn DCDn = 0010 - READ 00 -- *Modem status* - Set RINn CTSn DSRn DCDn = 0100 - READ 00 -- *Modem status* - Set RINn CTSn DSRn DCDn = 1000 - READ 00 -- *Modem status* - Set RINn CTSn DSRn DCDn = 1111 - READ 00 -- *Modem status*	 00(3 ..0) = 0000 00(3 ..0) = 0001 00(3 ..0) =0010 00(3 ..0) =0100 00(3 ..0) = 1000 00(3 ..0) = 1111
4	**Modem Control** .CPU to Toggle DTRn Set no parity mode	8.2.2	1	- WRITE 00 000 - WRITE 00 100 – *NO parity* - WRITE 00 000	DTRn = 0 DTRn = 1 DTRn = 0

5	**Transmit protocol** **CPU writes 1 word into buffer, interrupt on empty** . Set modem interface to disabled transmission mode. . Enable empty transmit interrupt. . Write 1 random data to transmit buffer . Modify modem interface until Enable of transmission mode. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 6.0, 5.1.1 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0111 - WRITE 10 010000 – *Xmt buffer setup* - WRITE 11 RandomData – *fill xmt buffer* - Wait for 2 baud cycles - Set RINn CTSn DSRn DCDn = 0011 - Wait for 51 cycles - Set RINn CTSn DSRn DCDn = 0001 - Wait for 51 cycles - Set RINn CTSn DSRn DCDn = 0000 - Enable all transmit interrupts - Read 10 – read/clr xmt PIR status until message is sent. (PIR(4) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- No serial output - No serial output - No serial output - Serial transmission Protocol per 3.3. Interrupt at end of transmission,
6	**Transmit protocol . CPU writes "n" words into buffer, interrupt on empty (MT)** . Set modem interface to enabled transmission mode. . Enable empty transmit interrupt. . Write buffer-depth random data to transmit buffer. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 10 10 -- hex – – *Interrupt on MT* - Fill xmt buffer with random data for K in 1 to buffer_depth loop WRITE 11 RandomData – *fill xmt buffer* end loop; - Read 10 – read/clr xmt PIR status until message is sent. (PIR(4) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- Serial transmission. Protocol per 6.0. Interrupt at end of transmission of all words in buffer,

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
7	**Transmit protocol. CPU writes "n" words into buffer, interrupt on Almost empty** . Set modem interface to enabled transmission mode. . Enable empty transmit interrupt. . Write buffer depth random data to transmit buffer. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 10 001000 – *Intrpt Almost empty* - Fill xmt buffer with random data for K in 1 to buffer_depth loop WRITE 11 RandomData *– fill xmt buffer* end loop; - Read 10 – read/clr xmt PIR status until message is sent. (PIR(3) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- Serial transmission. Protocol per 6.0. Interrupt when transmit buffer reaches down to the almost empty level.

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
8	**Transmit protocol. CPU writes "n" words into buffer, interrupt on half-full** . Set modem interface to enabled transmission mode. . Enable empty transmit interrupt. . Write buffer depth random data to transmit buffer. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 10 000100 – *Intrpt half-full* - Fill xmt buffer with random data for K in 1 to buffer_depth loop WRITE 11 RandomData – *fill xmt buffer* end loop; - Read 10 – read/clr xmt PIR status until message is sent. (PIR(2) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- Serial transmission. Protocol per 6.0. Interrupt when transmit buffer reaches down to the half-full.
9	**Transmit protocol. CPU writes "n" words into buffer, interrupt on almost-full** .Set modem interface to enabled transmission mode. . Enable empty transmit interrupt. . Write buffer depth random data to transmit buffer. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 10 000010 – *intrpt almost-full* - Fill xmt buffer with random data for K in 1 to buffer_depth loop WRITE 11 RandomData – *fill xmt buffer* end loop; - Read 10 – read/clr xmt PIR status until message is sent. (PIR(1) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- Serial transmission. Protocol per 6.0. Interrupt when transmit buffer reaches down to the almost-full level.

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
10	**Transmit protocol. CPU writes "n" words into buffer, interrupt on full** . Set modem interface to enabled transmission mode. . Enable empty transmit interrupt. . Write buffer depth random data to transmit buffer. . Check for interrupt. . Read transmit PIR	8.2.6 8.2.8 5.1.2.11 6.0 8.2.5	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 10 000001 – *intrpt full* - Fill xmt buffer with random data for K in 1 to buffer_depth loop WRITE 11 RandomData – *fill xmt buffer* end loop; - Read 10 – read/clr xmt PIR status until message is sent. (PIR(0) = '1' - Verify serial output sequence. - Verify Transmit interrupt (bit 1) is active and is reset with xmt PIR read	- Serial transmission. Protocol per 6.0. Interrupt when transmit buffer reaches off to the full level.

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
11	**Receive protocol interrupt on not empty** . Set modem interface to enabled transmission mode. . Enable empty receive interrupt. . Send buffer-depth words to data to Rxd port. . Check for interrupt. . Read receive status and PIR	8.2.6 6.0 8.2.3 5.1.2.11 8.2.7	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 01 010000 *intrpt on not MT* - send buffer-size words to RxD port with random data - Wait for interrupt within serial transmission time - Read 01 – read/clr rcv buffer status until message is received. (PIR(4) = '1' - Verify receive interrupt (bit 0) is active and is reset with xmt PIR read - Read data	Received data (all words) = transmitted data.(all words) Status register is as expected.
12	**Receive protocol interrupt on almost-empty** . Set modem interface to enabled transmission mode. . Enable empty receive interrupt. . Send buffer-depth words to data to Rxd port. . Wait for interrupt.	8.2.6 6.0 8.2.3 5.1.2.11	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 01 001000 *intrpt on almost- MT* - send buffer-size words to RxD port with random data - Wait for interrupt within serial transmission time - Read 01 – read/clr xmt buffer status	Received data (all words) = transmitted data.(all words) Status register is as expected.

	. Read receive status and PIR	8.2.7		- Read data	

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
13	**Receive protocol interrupt on half-full** . Set modem interface to enabled transmission mode. . Enable empty receive interrupt. . Send buffer-depth words to data to Rxd port. . Wait for interrupt. . Read receive status and PIR	8.2.6 6.0 8.2.3 5.1.2.11 8.2.7	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 01 000100 *intrpt on half-full* - send buffer-size words to RxD port with random data - Wait for interrupt within serial transmission time - Read 01 – read/clr xmt buffer status - Read data	Received data (all words) = transmitted data.(all words) Status register is as expected.
14	**Receive protocol interrupt on Almost-full** . Set modem interface to enabled transmission mode. . Enable empty receive interrupt. . Send buffer-depth words to data to Rxd port. . Wait for interrupt. . Read receive status and PIR	8.2.6 6.0 8.2.3 5.1.2.11 8.2.7	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 01 000010 *intrpt on almost-full* - send buffer-size words to RxD port with random data - Wait for interrupt within serial transmission time - Read 01 – read/clr xmt buffer status - Read data	Received data (all words) = transmitted data.(all words) Status register is as expected.

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
15	**Receive protocol interrupt on full** . Set modem interface to enabled transmission mode. . Enable empty receive interrupt. . Send buffer-depth words to data to Rxd port. . Wait for interrupt. . Read receive status and PIR	8.2.6 6.0 8.2.3 5.1.2.11 8.2.7	1	- Set RINn CTSn DSRn DCDn = 0000 - WRITE 01 000001 *intrpt on full* - send buffer-size words to RxD port with random data - Wait for interrupt within serial transmission time - Read 01 – read/clr xmt buffer status - Read data	Received data (all words) = transmitted data.(all words) Status register is as expected.
16	**Tests 2 to 15, with PARITY ON, ODD**	8.2.2	1	- WRITE 00 111 -- *Odd parity* - Repeat tests 1 through 15	DTRn = 0
17	**Tests 2 to 15, with PARITY ON, EVEN**	8.2.2	1	- WRITE 00 110 – *Even parity* - Repeat tests 1 through 15	DTRn = 0
18	**Receive framing error, Even Parity**	7.1.1	1	Same conditions as 15, except: Force a framing error on the receive RxD, READ Status	
19	**Receive parity error, Even Parity**	7.1.2	1	Same conditions as 15, except: Set parity ON and force a parity error on the receive RxD data	
20	**Receive buffer overrun error, Even Parity**	7.1.3	1	Same conditions as 15, except: Do not flush receive buffer.	

Table 4.1 Feature Extraction and Verification Criteria (Continued)

Tst #	FEATURE & DIRECTED TEST STRATEGY	SPEC #	Priority	TEST SEQUENCE	VERIFICATION CRITERIA
21	**Transmit buffer overrun error, even Parity**	7.1.4	1	Same conditions as 10, except: Force a more data write into the transmit buffer than the buffer can hold.	
22	**Receive framing error, Odd Parity**	7.1.1	2	- WRITE 00 111 -- *Odd parity* Same conditions as 15, except: Force a framing error on the receive RxD	
23	**Receive parity error, Odd Parity**	7.1.2	2	Same conditions as 15, except: Set parity ON and force a parity error on the receive RxD data	
24	**Receive buffer overrun error, Odd Parity**	7.1.3	2	Same conditions as 15, except: Do not flush receive buffer.	
25	**Transmit buffer overrun error, Odd Parity**	7.1.4	2	Same conditions as 10, except: Force a more data write into the transmit buffer than the buffer can hold.	

4.2 Testbench Architecture

Several architectural elements must be considered in the definition of the testbench environment, including the following:

- Reusability / ease of use / portability / verification language
- Number of BFMs to emulate the separate busses
- Synchronization methods between BFMs
- Transactions definition and sequencing methods
- Transactions driving methods
- Verification strategies for design and its subblocks

4.2.1 Reusability / ease of use / portability / verification language

VHDL code will be used for this design because VHDL is an IEEE standard language, and is portable across platforms. A reusable design style will be applied as discussed in the following subsections.

4.2.2 Number of BFMs

The UART consists of two independent channels, one channel representing the CPU interface to send data (onto the *TXD* line) and read status and received data, and the other channel representing the RECEIVE data (onto the *RXD* line). To maintain the modeling integrity of the system, it is important that those two channels be modeled with BFMs that can emulate the asynchronism of those channels. However, a synchronization scheme between those BFMs is essential to control order of execution, when necessary.

For this design, two BFMs will be modeled as shown in Figure 4.2.2. One BFM will represent the HOST or CPU environment that initializes the UART, reads status information, sends transfer data (to be sent by UART over the TXD signal), and reads collected data (to be collected by the UART over the *RXD* signal). The other BFM will represent the RECEIVE BFM to emulate the serial interface sent over the *RXD* signal.

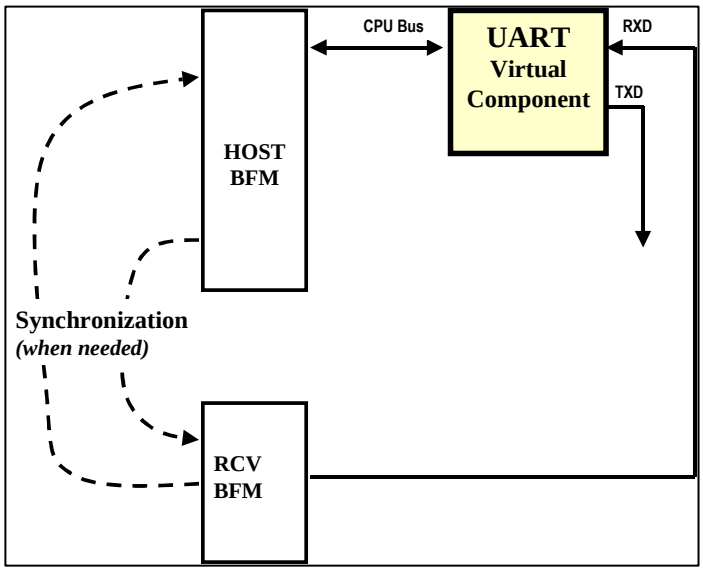

Figure 4.2.2 BFMs for UART Model

4.2.3 BFMs Synchronization Methods

The synchronization of transactions is often referred as concurrency control. There are several synchronization methods known in the computer science field that can be implemented in VHDL and in many verification languages. These concurrency control techniques include the following:

1. **Fork and Join**. The fork statement and join statement is similar to Verilog *Fork Join* [Verilog LRM 9.8.2], and allows the execution of two or more parallel threads in a parallel block. VHDL does not directly have this syntax. However, it can be emulated with events.
2. **Events.** Events are signals used to synchronize concurrent processes. For example, one signal (e.g., Sync) can be used to block a process, and another signal (e.g., Trigger) can be used to release, or unblock the blocked process. Another example is a handshake where the requesting process that wishes to execute a transaction makes a request to an arbiter logic. Because VHDL allows user defined resolution functions, the events synchronization method can be implemented with a single resolved signal of resolved integer type, where the signal gets resolved to the lowest value being driven. A unit asserting a value on this resolved bus must wait until that bus has a value equal to the asserted value. In VHDL, the algorithm is as follows:

```
SYNCS    <= IntegerValue; -- new value asserted onto resolved integer
 Wait until Clk = '1'; -- SYNCS is updated
 while SYNCS /= IntegerValue loop
      -- Wait until level adjusts to required sync level
   Wait until Clk = '1';
 end loop;
```

Figure 4.2.3 demonstrates this concept via an example. Driver A initially holds the resolved integer signal to a –100 value, preventing process B from continuing since it awaits a ZERO. When Driver-A finally assigns a ZERO onto the SYNCS bus, the resolved integer signal resolves to ZERO, thus enabling process B to continue. Now Driver B executes for a period, and then assigns a ONE onto the resolved integer signal. In the meantime, process A continues, and assigns a ONE after a period. If the signal gets resolved to ONE, then process B is granting process A permission to continue. Process A assigns a TWO after some period of work. The signal is resolved to ONE, and process A must now wait until process B enables process A to continue by assigning a TWO.

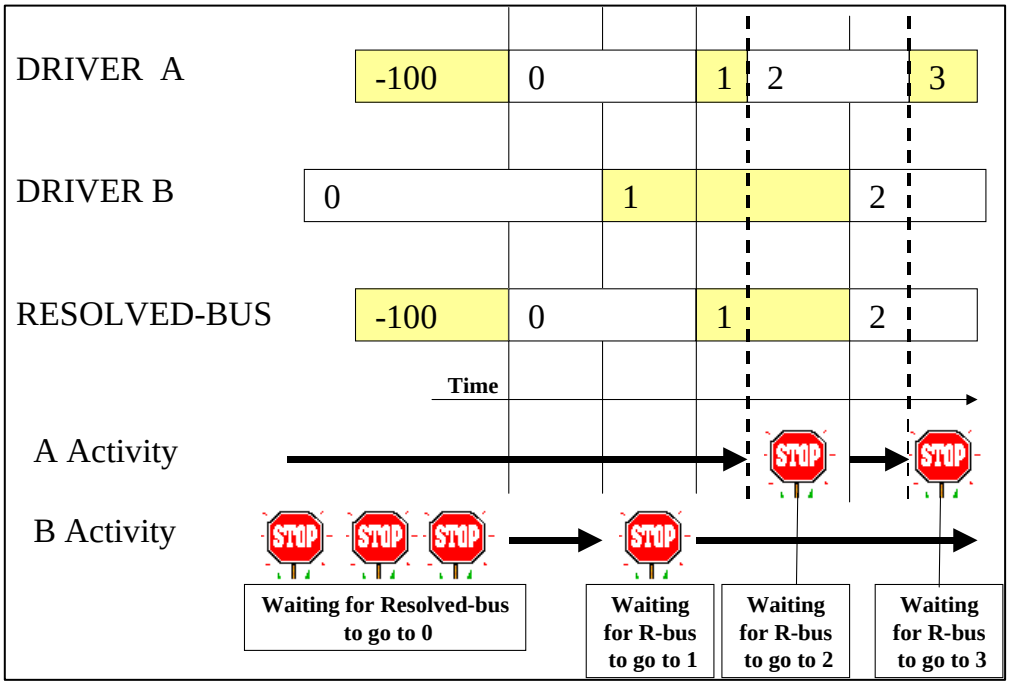

Figure 4.2.3 Application of Resolved Integer Signal where Low Value Wins

1. **Semaphore.** A semaphore is a primitive operation used for mutual exclusion and synchronization. A flag variable is used to govern access to shared system resources. A semaphore indicates to other potential users that a file or other resource is in use and prevents access by more than one

user. Semaphores can be created in VHDL, and are typically supported in verification languages.

4. **Mailbox.** A mailbox is a mechanism to exchange messages between processes. Data can be sent to a mailbox by one process and retrieved by another. This can be implemented in VHDL with shared variables and link lists.
5. **Timeout.** This represents the maximum amount of time a process will wait for a request or synchronization. VHDL supports the timeout concept with the statement:

   wait until *condition* ***for*** *timeout_time;*

Resolved integer is adaptable to mutiple processes

The UART design will use the event synchronization technique using the resolved integer method. This technique will be used because it is relatively simple and is automatically adaptable to the synchronization of additional concurrent BFMs.

4.2.4 Transactions definition and Sequencing methods

There are several methods to define the transactions asserted by the BFMs. Potential methods considered for the UART testbench include the following:

1. **Command files with Instruction Set (ISA).** This technique defines a high level ISA for the commands that are stored in a file. This is typically a limited set of instructions with parameters. For example, a WRITE instruction at a specific address, with some data. This method requires a parser to parse the instructions into its components.
2. **Command procedures.** This technique uses procedures to define the waveforms asserted onto the formal parameters of the procedure. The procedure calls (e.g., WRITE, READ) can be initiated from either VHDL code, or from the parsed instruction read from a file.
3. **VHDL code.** This technique uses the diverse features of VHDL to assert the desired waveforms. Subprograms may be used to enhance reusability. All transactions sequencing is defined in VHDL.

The UART testbench will use the command files with an instruction set because this technique is easier to maintain since the user does not need to know or modify VHDL code to update the transaction sequences. Section 4.2.7 expands the application of command files for the UART.

4.2.5 Transactions driving methods

Another tradeoff to make in the architecture of BFMs is the driving methods of transactions. The low level transactions can emanate from procedures, components, or inline VHDL code. In the UART BFMs, each BFM (host and receive) will consist of two components, the *client* component and the *server*

component[1]. The *client*, or executive, makes high-level transaction requests (e.g., *Read, Write*) to the *server*. The *server* detects the arrival of new messages and honors the requests by providing the actual bus interfaces (the *handshakes* and *protocol*) to the UUT. The *server* also collects any interface data (using the *protocol*) and transfers that information to the *client* through a signal. The *clients* and *servers* are modeled as components. This object-oriented approach for the design of scenario generators enhances the concepts of model reusability and maintainability. The advantages of this approach are:

- **Separation between the tasking** of jobs (e.g., WRITE) by the client, and the **execution** of the jobs (i.e., protocol, or twiddling of the many interface signals by server).
- **Reuse of client** when interface changes because of changes or testing of subblocks. The client is unchanged when the protocol changes.
- **Reuse of server** when different transactions or tasks are needed. The server is unchanged when the sequence of tasks (in client) is changed.

4.2.6 Verification strategies for design and its subblocks

The UART design will be verified with an automatic verifier component that performs the automatic detection of protocol violations and transaction logging. Section 4.3 discusses the verifier model.

4.2.7 Detailed Testbench Architecture

The testbench architecture for the UART consists of the following functional elements, as shown in Figure 4.2.7-1.

1. **UART**, representing the UUT.
2. **HOST BFM**, emulating the host interface to the UART
3. **RECEIVE BFM**, emulating the UART RXD receive serial port
4. **VERIFIER**, providing the verification and reporting of the UART behavior.

The two clients are synchronized with a resolved integer SYNCS signal, as discussed in section 4.2.3. An alternate approach to the dual-command file method is to use a single-command file that control the host server and the receive server, as shown in Figure 4.2.7-2. The single-command file is easier conceptually since there is no synchronization between multiple BFMs. However, more fields are required to identify which server is the recipient of the command. In addition, this technique is less flexible in generating asynchronous transactions in each BFM because of the sequential dependency in the control of the transactions (e.g., from one source). The single-command approach is not selected for this testbench because it is less flexible in the control of concurrent transactions.

---

1 *VHDL Coding Styles and Methodologies, 2nd Edition*, Ben Cohen, KAP, 1999.

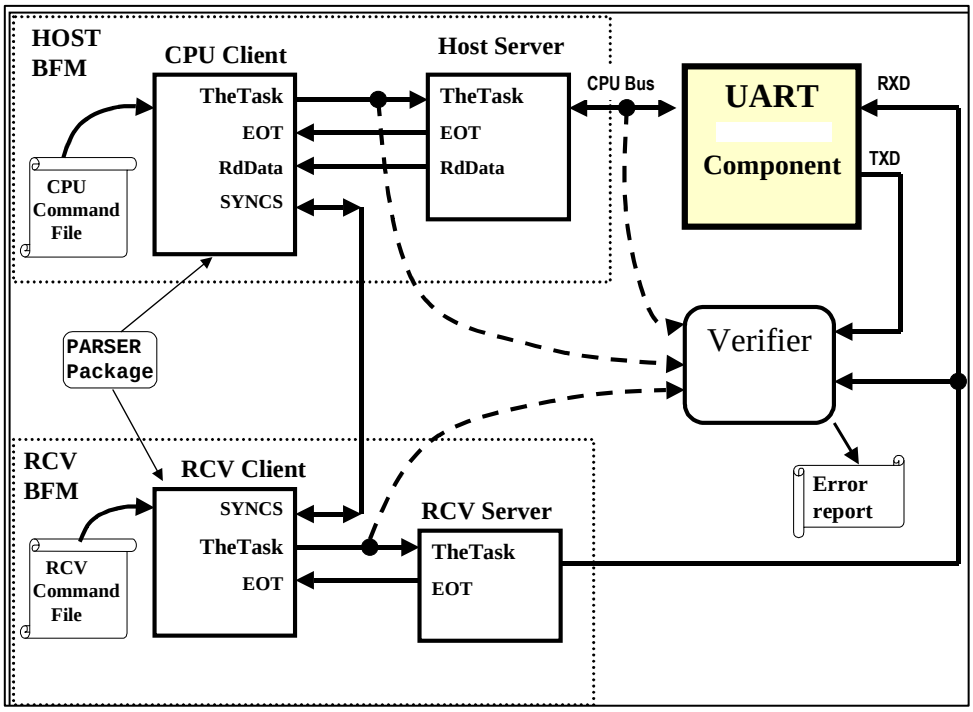

Figure 4.2.7-1 Testbench Architecture Overview using BFMs and Automatic Verification

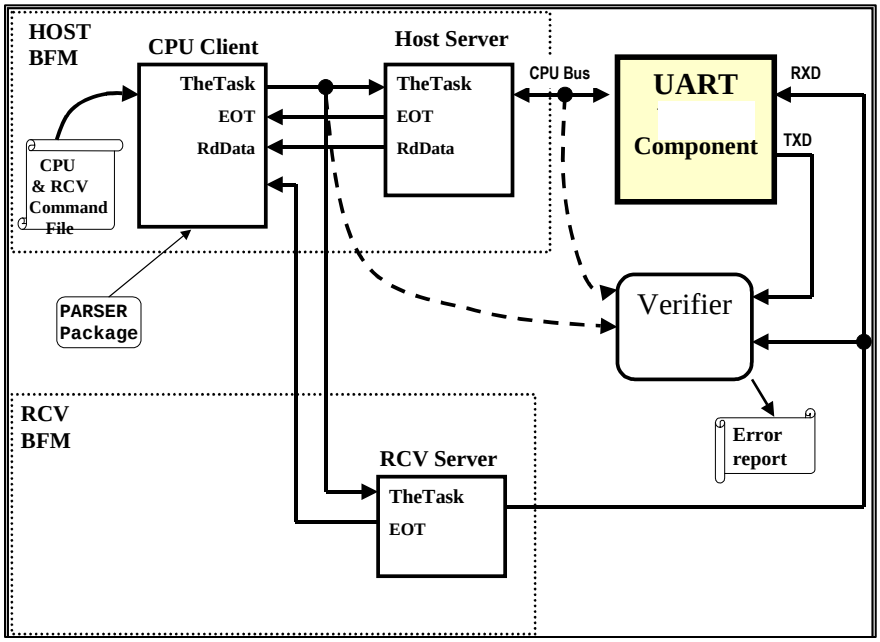

Figure 4.2.7-2 Testbench Architecture Potential using Single-Command File (Not used)

4.2.8 Subblock Verification

For this project, a verifier will be used for verification of the UART. However, the subblocks will be verified visually, with test vectors generated from BFMs intended for the UART testbench. For this project, visual, instead of automatic, verification of the subblocks will be performed for economic and scheduling reasons. BFM reuse will be emphasized for the project. Specifically, the client model will be reused for all the subblock models except for the clock control, which only requires simple clocks. However, since every subblock has different interfaces, separate subblock servers will be built. Figure 4.2.8 represents a view of this concept. Each server will interpret the command tasks differently, depending on the type of server it represents. For example, a WRITE task to a CPU interface implies a WRITE protocol using the CPU bus protocol, whereas a WRITE to a FIFO implies a PUSH of data into the FIFO.

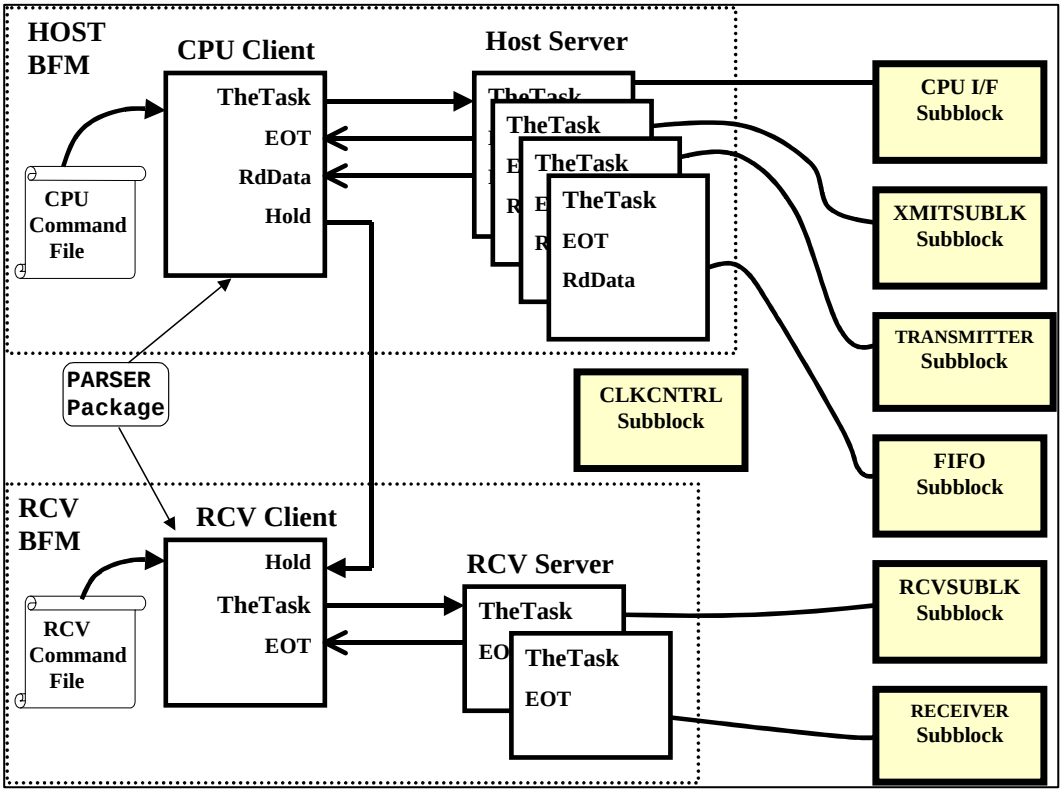

Figure 4.2.8 Subblock Verification Overview using BFMs and Visual Verification

4.2.9 Instruction File

When verifying compliance to the specifications, the directed testcases will use instruction files to define the high-level test sequences. The *client* model reads the instruction file, and the *parser* package parses those instructions. The *client* transfers the parsed instructions through the *TheTask* signal to the *server* that decodes the instructions, and provides the waveform protocol to the design under test (e.g., UART or subblocks). An end-of-transfer (*EOT*) is emitted by the *server* to the *client* to identify the end-of-execution of the requested instruction. The benefits of this approach include:

- **Readability**. Instructions are English-like in mnemonics, and allow comments for documentation. They represent high-level tasks, unlike low-level assembly-language instructions. During simulation, the task can be displayed on the wave-view of the simulator to identify which high-level instruction is executed. The *EOT* pulses easily identify the end of transactions.
- **Maintainability/flexibility**. A user can easily modify the instruction sequence with a text editor, with no need to know or modify the VHDL code.
- **Compilation/elaboration speed**. A change to the contents of the file requires no recompilation or re-elaboration of the testbench code.

For corner and random testing, the testcases will first use instruction files for the initial setup of the UUT environment. This will then be followed by VHDL code for the generation of pseudo-random transactions at pseudo-random time intervals. VHDL code is used because it is a powerful language with appropriate constructs for looping and iterations. VHDL code and instructions defined in files can freely be intermixed. The file instructions will be called from a procedure call.

Table 4.2.2 defines the mnemonics used in the instruction files, and provides application examples for those instructions.

Table 4.2.2 Transaction Instructions used in Files

#	INSTRU-CTION	FUNCTION	EXAMPLE
1	**WRITE WRIT ***	**Write a single word @ address (binary) with data (hex)**	**WRITE 10 1F -- Reset, xmt Intrpt enb(4..0)**
2	**RNDM_DATA RNDM ***	**Write a single word @ address (binary) with random data, sized to width of UART**	**RNDM 11**

Table 4.2.2 Transaction Instructions used in Files (Continued)

#	INSTR	FUNCTION	EXAMPLE
3	READ	READ a single word @address (binary),	READ 01 -- Read RcvFifoSts 5 bits
4	IDLE	Stay in IDLE for n system clocks	idle 10000 – wait for 1K cycles
5	RESET RESE *	hardware reset for "n" cycles	RESET 6
6	DISP	Displays a message.	DISP End of XYZ test sequence -- Client asserts the message.
7	MODE	Sets the BFM to one of the following modes: NORMAL, FRAME_ERR, PARITY_ERR	MODE NORMAL
8	RDUNTIL RDUT *	Read @ADDR (in bit) MASK (in Hex) Interval (in natural) until the received masked data has a ONE.	RDUNTIL 11 02 50 – addr="11", mask = X"02", -- Interval = 50 cycle -- Read data from address, -- Temp := MASK AND fetched_data -- If any bit in Temp = '1' then continue -- else wait for Interval clock cycles -- Repeat
9	ENVSETUP ENVS *	Sets the environment for the uart . Four-bit data in binary.	ENVSETUP 0000 – binary (3) = RIn – Ring (2) = CTSn -- Clear To Send (1) = DSRn -- Data Set Ready (0) = DCDn -- Data Carrier Detect
10	CALL	Jump to subroutine	CALL c:/uart/tests3to5,txt
11	SYNC	Assigns a sync integer to a resolved integer signal	SYNC 3
12	WT4INTRPT WT4I *	Wait for interrupt for n cycles Instruction continues after "n" cycles if no interrupt occurs	WT4INTRPT 10 100 – xmt intrpt, up to 100 clk
13	STOP	STOP Simulation	STOP -- Client asserts the message. -- Server stops simulation if STOPSIM -- generic is set to TRUE, else instruction -- is ignored

* Optional Instruction mnemonic. Parser considers only the first four characters

4.3 Verifier

The verifier model will provide several services to facilitate the automatic verification and debug of the UUT. The functions performed by the verifier willl include:

- **Verify compliance to requirements**
- **Reporting of errors linked to requirements**
- **Reporting of environment and transactions**

The verifier will perform its automatic checking by first scoreboarding (i.e., keeping track of) all commanded transactions and setups instructed by the *client* through the tasks. It will then monitor the interfaces of the UART, and will verify that the expected UART transactions do occur within the allotted or required latencies. For example, a *WRITE* task from the *client* should cause the data written into the *UART* to be issued onto the *TxD* serial port within 2 baud cycles, provided all the transmit conditions are satisfied. The verifier will then check that this output event does occur, and that the RS232 protocol (i.e., serialization, parity, format) is abided.

4.3.1 Error Detection by Verifier

When an error is detected, the *verifier* will report each error in the format shown in Table 4.3.1-1.

Table 4.3.1-1 Error Reporting Format and Example

TIME ns	ERROR	REQUIREMENT	OBSERVED DATA	EXPECTED DATA
13700	**UART Fails to detect parity error**	**8.2.3**	**01100001**	**00110001**

Table 4.3.1-2 provides the list of errors to be reported by the verifier.

4.3.2 Transaction Log

The verifier will log the transactions and errors in one log file. The information to be logged will include:

1. **Simulation time**
2. **Transaction**, including
   - CPU: Read, Write
   - Serial data word sent out: Txd
   - Serial data word read in: Rxd
   - Modem control: RTSn, CTSn, DSRn, DCDn, DTRn, RIn
3. **Errors** See Table 4.3.1-2

Table 4.3.1-2 List of Errors Reported by Verifier

```
UART Fails to detect parity error
UART detects Parity error when none
Parity error in sent message on TXD
UART Fails to detect framing error on RXD
Framing error detected when none
Framing error in sent message on TXD
Failure to receive a message on RXD
Failure to send a message on TXD
Sending a message illegally on TXD
PIR Receive in error
RCV interrupt error
XMT Interrupt error
ERROR in read of modem data @ addr  =  00
DTR output /= CPU commanded data
RCV PIR Empty Error
RCV PIR almost Empty Error
RCV PIR Half-full Error
RCV PIR Almost-full Error
RCV PIR full Error
XMT Data error
PIR transmit error or write to full buff
XMT PIR Empty Error
XMT PIR almost Empty Error
XMT PIR Half-full Error
XMT PIR Almost-full Error
XMT PIR full Error
Received data unequal to expected data
```

4.3.3 Coverage

A minimum of statement coverage will be executed, and 99% of statement coverage will be verified.

4.3.4 Compliance Matrix

Table 4.3.4 is a summary of the compliance matrix for each requirement, and tests that verify the requirement.

Table 4.3.4 Compliance Matrix

REQ #	REQUIREMENT	VERIFIER TESTCASE #
1.0	**Scope**	**NA**
2.0	**Definition**	**NA**
3.0	**Applicable documents**	**NA**
4.0	**Architectural Overview**	**NA**
5.0	**Physical Layer**	**-**
5.1	**Interface Port Description**	**-**
5.1.1	**RS-232 Serial Interface**	**-**

Table 4.3.4 Compliance Matrix (Continued)

REQ #	REQUIREMENT	VERIFIER TESTCASE #
5.1.1.1	TxD, Transmit Data	5, 6,7,8,9,10
5.1.1.2	RxD, Receive Data	11,12,13,14,15
5.1.1.3	RTSn, Request To Send	5, 6,7,8,9,10
5.1.1.4	CTSn, Clear To Send	5, 6,7,8,9,10
5.1.1.5	DSRn, Data Set Ready	3
5.1.1.6	DCDn, Data Carrier Detect	3
5.1.1.7	DTRn, Data Terminal Ready	3
5.1.1.8	RIn, Ring Indicator	3
5.1.2	CPU Interface	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.1	Addr	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.2	CS0	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.3	CS1	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.4	CS2n	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.5	Din	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.6	RDn	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.7	Resetn	2
5.1.2.8	WRn	5, 6,7,8,9,10,
5.1.2.9	DO	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.10	OutEnb	5, 6,7,8,9,10, 11,12,13,14,15
5.1.2.11	Intrpt	6,7,8,9.10.11,12,13,14,15
5.1.3	Clock Interface	ALL
5.1.3.1	Clk	ALL
5.1.3.2	TC_synch	Not tested
5.1.3.3	RC_synch	Not tested
5.1.3.4	Clk16x	ALL
6.0	Protocol Layer	5, 6,7,8,9,10, 11,12,13,14,15
7.0	Robustness	-
7.1	Error Detection.	-
7.1.1	Receive framing error	18, 22
7.1.2	Receive parity error	19,23
7.1.3	Receive buffer overrun error	20,24
7.1.4	Transmit buffer overrun error	21,25
7.2	Error Handling	NA
8.0	Hardware and Software	-
8.1	Fixed Parameterization	-
8.2	Software Interfaces	-
8.2.1	Address "00", CPU READ, Modem Status	3

Table 4.3.4 Compliance Matrix (Continued)

REQ #	REQUIREMENT	VERIFIER TESTCASE #
8.2.2	Address "00", CPU WRITE, Modem Control	3, 16, 17
8.2.3	Address "01", CPU READ, Receive	2, 11, 12, 13, 14, 15

	Buffer Status	
8.2.4	**Address "01", CPU WRITE, Receive Buffer Control**	**11,12,13,14,15**
8.2.5	**Address "10", CPU READ, Transmit Buffer Status**	**2,5, 6,7,8,9,10**
8.2.6	**Address "10", CPU WRITE, Transmit Buffer Control**	**5, 6,7,8,9,10, 11, 12, 13**
8.2.7	**Address "11", CPU READ, Read Receive Data**	**11,12,13,15**
8.2.8	**Address "11", CPU WRITE, Write Transmit data**	**5,6,7,8,9,10,14**
8.3	**Modes of Operation**	**ALL**
9.0	**Performance**	**-**
9.1	**Frequency**	**Synthesis and layout tools**
9.2	**Power Dissipation**	**Not performed**
9.3	**Electrical**	**Not performed**
9.4	**Environmental**	**Not performed**
9.5	**Technology**	**NA**
10.0	**Testability**	**Not performed**
11.0	**mechanical**	**NA**

5.0 Design Tools

Table 5.0 summarizes the list of tools used for verification.

Table 5.0 Tools Used for Verification

TOOL	VENDOR	FUNCTION
ModelSim EE 5.4b with Code Coverage	**Model Technology, Mentor Graphics**	**VHDL/Verilog co-simulator with code coverage**
Emacs 20.6.1 with Vhdl-mode 3.31.6 beta	**GNU**	**Language sensitive editor**

This list summarizes the tools used in the design of the UART for this book. Users need to identify the tools they intend to use in their project. Degugging tools such as Novas' Debussy[2](r) Total Debug (tm) system might be very helpful during the debugging stage.

[2] http://www.novas.com/