Component Design by Example

… a Step-by-Step Process Using VHDL with UART as Vehicle

Ben Cohen

VhdlCohen Publishing
Los Angeles, California
http://www.vhdlcohen.com

Component Design by Example

… A Step-by-Step Process Using VHDL with UART as Vehicle

Published by:
VhdlCohen Publishing
P.O. 2362
Palos Verdes Peninsula CA 90274-2362
vhdlcohen@aol.com
http://www.vhdlcohen.com ***or***
http://members.aol.com/vhdlcohen/vhdl/

Library of Congress Cataloging-in-Publication Data
Library of Congress Card Number: 00-109498
Cohen, Ben
Component Design by Example
ISBN 0-9705394-0-1

Printed on acid-free paper

Printed in the United States of America

NOTES ADDED December 18, 2014

This book was written in 2001. Since then technology has changed, and the verification methodologies have improved. Specifically, Verilog matured to become SystemVerilog and assertion languages matured along with formal verification. However,m VHDL did not make such pogress and in MHO, is lagging behind. Below are some of the changes that occured in SystemVerilog. However, I need to caution the readers of this book that the verification methodologies offered here are outdated and are not recommended. The guidelines in writng specifications and requirements still hold.

SystemVerilog supports specific features for verifcation, and those are not supported by VHDL. Several aspects are needed in verification:
1) Stimulus generator. For that SV supports:
a) Constraint random generation with random gen stability.
FYI, on random stability: This explains it
http://www.testbench.in/CR_20_RANDOM_STABILITY.html
Basically, this deals with reproducability because of calls to RNG (random number generator (something not available in VHDL)

2) Verification of results:

This is supported by SystemVerilog Assertions (SVA) and by data structures needed for verification; these include associative arrays, queus, mailboxes, the generate statement (like VHDL), vectors.

3) Coverage to determine when done, or how much was done.
SV provides the SVA cover and the covergroup to help in that respect.

4) flexibility for changes
SV has classes that can be extended and dynamically linked to other classes to accommodate changes and tests.

5) standards: UVM was developed to facilitate the standardization for the development of TBs.

I authored several books on VHDL and Verilog (which represents the RTL part of SystemVerilog) and to me, the two languages are pretty much similar, except for syntax. After being a very strong advocate of VHDL, I changed my beliefs and switched to SystemVerilog, even for just the Verilog subset of it. It is less restrictive, but requires good coding guidelines. I am also a very strong advocate of SystemVerilog assertions (SVA) because they help in the clarification of the requirements and in the debugging and verification process. Formal verification is

gaining wide acceptance, and uses SVA for the definitions of the properties of the design. I also like the UVM methodology and approaches, though UVM is not very straightforward, and requires care, with a good understanding of SystemVerilog. However, with some guidance and rules, I believe that one can effectively use UVM with a lesser than full understanding of SystemVerilog. What I am talking about it is the use of predefined templates for the basic building blocks.

Ben Cohen
http://www.systemverilog.us/ ben@systemverilog.us

* SystemVerilog Assertions Handbook 3rd Edition, 2013 ISBN 878-0-9705394-3-6
* A Pragmatic Approach to VMM Adoption 2006 ISBN 0-9705394-9-5
* Using PSL/SUGAR for Formal and Dynamic Verification 2nd Edition, 2004, ISBN 0-9705394-6-0
* Real Chip Design and Verification Using Verilog and VHDL, 2002 isbn 0-9705394-2-8
* Component Design by Example ", 2001 ISBN 0-9705394-0-1
* VHDL Coding Styles and Methodologies, 2nd Edition, 1999 ISBN 0-7923-8474-1
* VHDL Answers to Frequently Asked Questions, 2nd Edition ISBN 0-7923-8115

Contents

FOREWORD

When Ben first asked me if I would be interested in reviewing his latest book, I was dually thrilled; once for the opportunity to contribute to the subject matter, second because it meant that Ben was taking on some new issues. In my many years with Synplicity®, I have had the opportunity to read or review many books. Of those, very few I appreciated enough to recommend. Two of Ben's earlier books, *VHDL Coding Styles and Methodologies* and *VHDL Answers to Frequently Asked Questions* are truly the best of the lot. They have long been on my technical recommendation list. Ben has an academic knowledge of the VHDL language, but utilizes that information with a practitioner's sense of reason. Both of these works are targeted toward the designer who utilizes VHDL. He fills these books with tips and recommendations, explanations as to why decisions are made and many references for further reading. What we gain from these books are a practical guide to applying VHDL with consideration for both the circuits to be implemented as well as the tools that you will used to create and verify the designs. I was anticipating that this new work would be similar in approach.

In *Component Design by Example*, Ben attacks the design reuse problem. This topic is timely and important. The Electronic Design Automation community has spent the most of the last decade foreshadowing the emergence and importance of design re-use and "IP" to obtain the next level of productivity gains. It is only in recently that we have seen more frequent occurrences of design reuse. In the past few years, our customers have begun to utilize various sizes and complexities of IP. With our customers, we have discussed, planned, pondered and solved various problems and futures for the development and reuse of design data and modular design flows. Consequently, we have observed that there is much design data that is reused, but only after significant effort. Often this is because the module was not successfully designed with reuse in mind. I suspect that many designers lacked resources and references broad enough to be useful on the topic of design for reuse.

Ben has created a pragmatic and useful book on design for reuse. It is useful because it brings lots of practical information and experience in one place. Useful because he dares to go beyond just the implementation phases of design, (which is more frequently addressed), and takes on the procedures from conception to specification and planning. PLEASE DON'T DISMISS THESE SECTIONS! Too many projects get into too much trouble down the line due to incomplete, ambiguous, or "undocumented" specifications and inadequate planning. It seems obvious, yet so many designers think of it as overhead that impedes progress.

Component Design by Example will be useful to any designer or design team. It may improve efficiency and improve products, or create disagreement in approach. My hope is it will stimulate discussion. I expect it will be the foundation for a future filled with IP. Read this before your next project. Then reread it afterward. You will benefit both times.

Andrew R. Dauman
Vice-President of Corporate Applications
Synplicity, Inc.
Sunnyvale, CA
September 28, 2000

PREFACE

As a VHDL trainer, consultant, and designer I recognized the need to demonstrate how to organize designs from conception to verified products. Many books address the design processes that include reuse methodologies for components, subblocks, and ASIC/FPGA designs. Since 1996, the *Virtual Socket Interface Alliance* ™ (VSIA)[1] has played a key role in the design reuse scenario by creating standards for the industry that permits even broader reuse. These books and standards provide guidelines and general recommendations, but lack the provisions of complete design examples (with code) that demonstrate all the front-end phases of a design process. These phases include definition of requirements, architecture, verification approaches, HDL coding, synthesis, verification, and documentation. This book covers this gap and addresses the process of defining requirements and translating these requirements into a verified soft component design.

A component is taken in the sense of a design unit, subblock (or partition of a larger design), and a commercial Intellectual Property (IP). This book recognizes that there are many methodologies adopted by industry to perform front-end designs. This book provides methodologies generally accepted and recommended by many textbooks, including: *Reuse Methodology Manual,* Michael Keating and Pierre Bricaud, *Writing Testbenches, Functional verification of HDL Models,* Janick Bergeron, and *Verification Methodology Manual for Code Coverage in HDL Designs* by Michael Stuart and David Dempster. Users can tailor their methodologies to what is required given the constraints of labor force, budgets, and available tools. Even though tools do not represent methodologies, tools are often used to guide methodologies.

This book serves the following goals:

1. It demonstrates, by example, the **processes** involved in specifying, implementing, and verifying a reusable soft component. Most of these front-end processes are independent of the HDL implementation or verification languages. Even though the disciplines involved in every project and company will vary, the presented processes provide a good modeling base that users can modify and build upon. This is the focus and purpose of the book.
2. It demonstrates how to write a **requirement specification** that defines the foundation of the design. Defining a good specification is a difficult task

[1] *Virtual Socket Interface Alliance* ™ http://www.vsi.org

as no single approach represents an exclusive best solution. It discusses potential specification methodologies, and provides a subset of this domain space as a modeling example to specify a serial interface, which is **UART-like in requirements**. These requirements meet those defined in the EIA (Electronics Industry Association) standard for serial data communication RS-232 UART. To emulate the challenges of real designs, such as parameterization and adaptability to different modes, this design adds additional performance requirements including: word widths, storage depths, and interrupt controller. This UART design is a soft component vehicle, and represents a model of moderate complexity with adaptability to many applications. A UART is a Universal Asynchronous Receiver Transmitter.

3. It demonstrates how to write an **architectural implementation** document, before writing any HDL code. This document defines the architectural approach for analysis and review.
4. It demonstrates how to write reusable and parameterized **VHDL synthesizable RTL code** for designs *using IEEE 1076.6 VHDL RTL for synthesis guidelines*[2]. This parameterization emulates typical components that require such flexibilities.
5. It demonstrates how to write a **verification plan** that defines the foundation of the verification approaches of a design. This document is essential because it guides the design of the testbench and verification models, and provides a forum for scrutinizing the validity and completeness of the tests.
6. It demonstrates how to verify a design using **reusable testbenches in VHDL**.[3] The issues of verification techniques are quite controversial,[4] particularly with the advent of new verification tools and languages. The elementary concepts of verification are independent of tools or languages, even though tools and languages are used to implement the verification. This book concentrates on the strict use of VHDL as the verification language because it is opened and portable. It illustrates advanced VHDL modeling techniques for the generation of stimulus vectors, including the use of text command files, client/server models, and pseudo-random transactions. The text commands for the control of transactions include a rich, but small, instruction set capable to recursively call command files defined as subroutines.
7. It demonstrates the **design and synthesis process, including results of synthesis and post-route with Altera tools.**

---

[2] See http://www.vhdl.org/siwg

[3] Those techniques are referenced in the following books:
VHDL Coding Styles and Methodologies, 2nd Edition, Ben Cohen, KAP, 1999.
Writing Testbenches: Functional Verification of HDL Models, Janick Bergeron, KAP 2000

[4] See http://janick.bergeron.com/guild

8. It provides, as a by-product of these methodologies, the design of a **high-speed, full-featured UART-like soft component** that can be tailored to applications that require an asynchronous or synchronous serial 8, 16, and 32-bit interface between two equipments.
9. It demonstrates the **integration of the soft component** into a subsystem.
10. It demonstrates the filling of the **OpenMore spreadsheet**. OpenMore is an assessment program developed by Synopsys and Mentor Graphics designed to enable a self-assessment of the reusability of commercial IP offerings.
11. It demonstrates an application of a ***Virtual Component Block Deliverables*** document, as described by *Motorola's Semiconductor Reuse Standard*[5].

All VHDL code described in the book is on a companion CD. All code was verified and simulated with *ModelSim version 5.4b*,[6] and synthesized with *Synplify* version *5.3.1*.[7] The CD also includes the **GNU toolsuite** with **EMACS** language sensitive editor (with VHDL, Verilog, and other language templates), and **TSHELL** tools that emulate a Unix shell.

This book is intended for:

1. **Engineers.** Book provides examples for the processes involved in defining components from requirements through verification and synthesis. It represents templates for the definition and implementation of a design. Engineers are better at copying and improving upon what is done, than from starting from scratch. This book will provide a head start in these processes.

2. **Application Designers**. Engineers who need a UART can use or modify the models described in this book.

3. **Tool Developers.** This book defines a well-specified and documented model of a common design of medium complexity, with hierarchy. Tool developers may exercise these requirements and HDL designs through new tools to demonstrate enhancements in the design processes.

4. **Trainers.** This book provides the focus of an advanced class for the definition and application of front-end methodologies and processes.

5. **College students**. Book demonstrates the design processes, from

---

5 Motorola: http://www.mot-sps.com/technology/srs/index.html

6 Model Technology: http://www.model.com

7 Synplicity http://www.synplicity.com

requirements to a verified implementation. It models real working industry design experiences

This book will be helpful as a guide through all the phases of a front-end design. It provides useful document templates for the definition of the requirement, implementation, and test plan documents. It also provides reusable code for the design of testbenches, and demonstrates by example, the application of this code for the synthesis and verification of a UART model.

About The Disk

Table 1 summarizes the contents of the enclosed CD.

Table 1 Contents of Enclosed CD

DIRECTORY NAME	DESCRIPTION
vhdl/rtl	clkcntrl.vhd -- clock controller subblock cpuif.vhd -- CPU Interface subblock fifo.vhd -- FIFO subblock rcvsublk.vhd – Receiver top-level with receiver subblock and FIFO receiver.vhd – Receiver subblock transmitter.vhd -- Transmitter subblock xmitsublk.vhd -- top-level with transmitter subblock and FIFO uart.vhd -- UART, top-level uart_level2.vhd – integration of UART into higher level
vhdl/tb	vsp.vhd -- miscellaneous package lfsrstd.vhd -- Linear feedback shift register package image_pb.vhd -- image package for conversion to strings size_pkg.vhd – For testbench use, global signals and constants parser_pb.vhd – Parser package for file I/O and command parsing uart_server.vhd -- UART Server for TB rcv_client.vhd -- Client for receive side of UART rcv_server.vhd -- Server for receive side of UART verifierpeek.vhd -- Verifier with use of global signals for synch verifierblkbox.vhd -- Verifier, black box approach uart_clientrndm.vhd – Uart client with good random tests uart_client_bad.vhd -- – Uart client with tests that produce errors uart8_tb.vhd -- Top level testbench for UART uart_c.vhd -- Configuration declarations for UART testbench fifo_server.vhd -- Fifo server for use with uart client fifo_tb.vhd-- fifo testbench filedata.vhd -- Reading data from a file through linked lists
vhdl/gates	uart.vho -- gate level model produced by Altera

Directory Name	Description
uart	cpu5to15.txt -- CPU subroutine command file instr1.txt -- CPU command file rcv11to15.txt -- Receive side subroutine command file rcvinstr.txt -- Receive side command file sw_reset.txt -- Reset subroutine command file
scripts	compile.do -- ModelSim compile scripts compile.log – ModelSim compile log run.do -- ModelSim run simulation
Altera	Synplify EDIF files, Files produced by Altera
simRuns	Gate_Sim – gate level simulation run output RTL_BlkBox -- RTL black box simulation run output RTL_GreyBoxRdmn -- RTL Gray box simulation run output RTL_Grey_OverrunError – RTL Gray box with errors
IEEE	NUMBIT.VHD, NUMSTD.VHD, STDLOGIC.VHD packages
Synopsys	attribut.vhd, bvarith.vhd, slmisc.vhd, stdarith.vhd, stdtxtio.vhd, std_cmpt.vhd, std_sign.vhd, std_unsg.vhd, synopsys.vhd packages
modelsim_ spy	PLI for ModelSim to access signals internal to a design
motorola_ Deliverable	srsmotdeliverable.pdf -- Motorola deleiverable document template
openMore	openmore-uart.xls – OpenMore spreadsheet for UART openmore.xls -- OpenMore spreadsheet -- unfilled
VHDL_ Syntax	VHDL'87 and VHDL'93 syntax in HTML format VHDL Help: VHDL Language Reference Guide
Verilog	CummingsSNUG2000SJ_NBA_rev1a.pdf, VerilogHDLCoding_Motorola.pdf, verilog_vs_vhdl.PDF, vlog1364- HDLCON-2000.pdf
PDF_FIles	ModelSIm5_2 reference guide, VHDL and Verilog reference cards, Std_Logic_1164 reference card, and European Space Agency Modeling guidelines
Usr	GNU toolset
man	GNU help files in Windows Help format. Root file is *ManPagesDir*
Etc	Csh.cshrc and my.cshrc startup files for TSHEL

Acknowledgements

Component Design by Example evolved from the recent recognition in books, technical articles, and presentations on the need to follow a process for large designs. In particular, I thank Michael Keating, author of *Reuse Methodology Manual,* and, Janick Bergeron, author of *Writing Testbenches, Functional verification of HDL Models* and organizer of the *Verification Guild* newsletter for bringing forward those important design issues.

I thank Model Technology for granting me a license of *ModelSim version 5.4b* with the built-in code coverage for the duration of the project. *ModelSim* is an excellent user-friendly HDL toolset that enabled the compilation and verification of this design.

I thank Synplicity for granting a license of *Synplify HDL Analyst* version *5.3.1* to synthesize the design and to extract the RTL views and delay paths for visual display and documentation of the logic and critical paths. *Synplify is* a very efficient, user-friendly, and insightful linting FPGA synthesis tool.

Altera's *MAX+PLUS® II ver 9.4* complemented *Synplify*'s EDF output because it routed the design and produced an accurate gate level model with routed timing. Altera was kind enough to grant me a tool license for this project.

I thank Novas for granting me a license of *Debussy 5.0* Total Debug (tm) system for complex designs at the gate, RTL and behavioral levels. Even though the license came in late in the project, I was still able to gain insightful views of the design and testbench.

I thank Reto Zimmermann from Synopsys for commenting on the manuscript, and for supporting the community on the excellent upgrades to *vhdl-mode* for *emacs* GNU text editor. The application of the language sensitive *vhdl-mode* significantly helped in the production of VHDL code for design and verification.

I thank YxI for providing me with more insights into advanced synthesis methodologies from higher-level HDL definitions.

I sincerely thank Andrew Dauman from Synplicity and Richard Hall from Cadence Design Systems, Inc for reviewing the book and providing many suggestions.

I especially thank my wife, Gloria Jean, for supporting me in this endeavor.

Sculpture Created by my Wife Gloria to
Express my Long Hours with a Laptop in the Creation of VHDL Books

About the Author

Ben Cohen is currently a VHDL language trainer and consultant. He has technical experience in digital and analog hardware design, computer architecture, ASIC design, synthesis, and use of hardware description languages for modeling of statistical simulations, instruction set descriptions, and hardware models. He applied VHDL since 1990 to model various bus functional models of computer interfaces. He authored *VHDL Coding Styles and Methodologies*, first and second editions, and *VHDL Answers to Frequently Asked Questions*, first and second editions. He was one of the pilot team members of the VHDL Synthesis Interoperability Working Group of the Design Automation Standards Committee who authored the *IEEE P1076.6 Standard for VHDL Register Transfer Level Synthesis*. He taught several VHDL training classes, and provided VHDL consulting services on several tasks.

VhdlCohen Training and Consulting
email: **VhdlCohen@aol.com**
Web page: **http://www.vhdlcohen.com/**
or **http://members.aol.com/vhdlcohen/vhdl/**

DISCLAIMER

Every attempt was made to ensure accuracy in the specifications and implementation of the models. However, All code provided in this book and in the accompanied CD is distributed with *ABSOLUTELY NO SUPPORT* and *NO WARRANTY* from the author. The author shall not be liable for damage in connection with, or arising out of, the furnishing, performance or use of the models provided in the book and CD.

The software media is distributed on an "AS IS" basis, without warranty.

If the media is defective, you may return it for a replacement.

Use or reproduction of the information provided in this book and on the enclosed CD for commercial gain is strictly prohibited.

1 OVERVIEW

This chapter introduces exemplary design processes required in typical front-end phases of a design. These phases include definition of requirements, architectural design, verification, behavioral modeling, RTL design, synthesis, timing analysis, design applications, documentation and delivery.

1.1 COMPONENT DESIGN PROCESS

Typical design process and phases

Figure 1.1-1 represents typical processes for the definition, implementation, and verification of a design. A design typically starts with an idea (e.g., a house, a car, an ASIC). A **requirement document** identifies what shall be built, and what kind of interfaces, performance, etc, it must meet (e.g., a house shall be 4000 square feet in size, two-stories, facing the ocean). Based on those design requirements, an **implementation specification** is followed to identify how the design will be built. This is a plan, but not the actual implementation (e.g., the house will have four bedrooms, with sliding windows facing the ocean, and prefabricated purchased appliances). The **verification plan** specifies what and how the product will be checked to insure that it meets the original requirements. The tests require information from both the requirements and the implementation documents. For example, a house may have several inspection steps (e.g., foundation, framing, electrical), and each purchased appliance may have its own separate inspections to insure quality before installation. Once the above documents are approved, the implementation and verification steps can proceed.

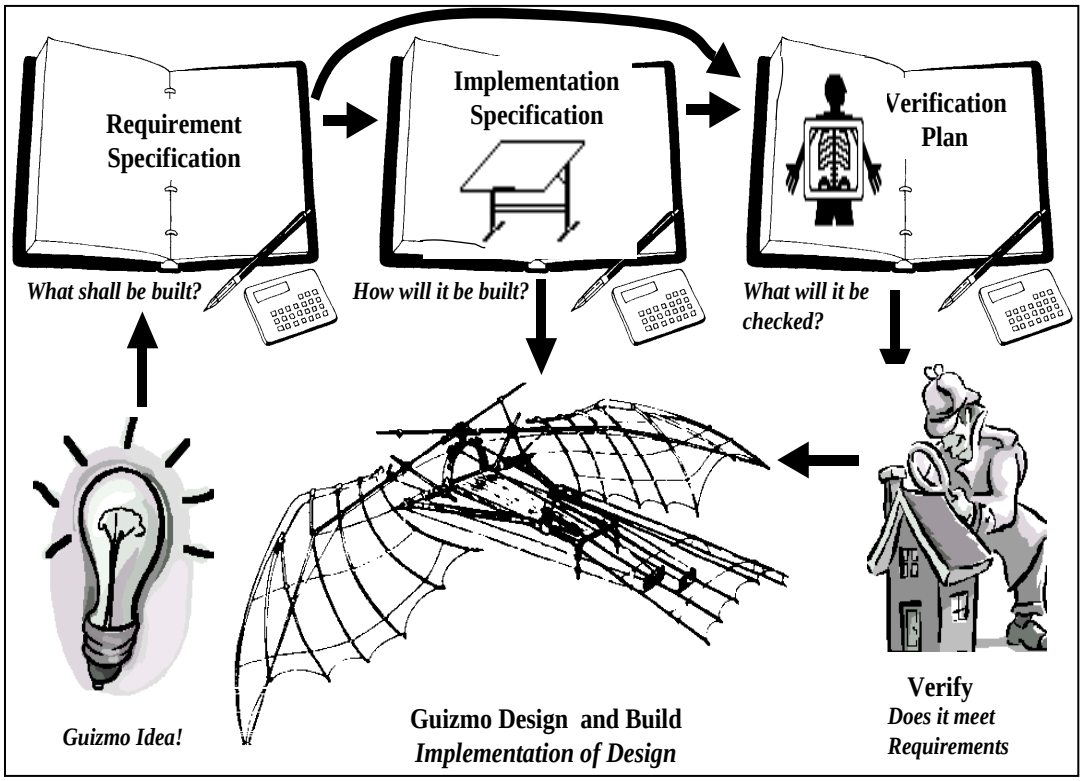


Figure 1.1-1 Typical Design Processes

Figure 1.1-2 represents typical component and subblock design processes, with emphasis on the front-end design aspects. Each of these processes is greatly expanded and demonstrated in the subsequent chapters, with complete samples for the documents, HDL code of the models and verification, and compilation scripts for simulation and synthesis.

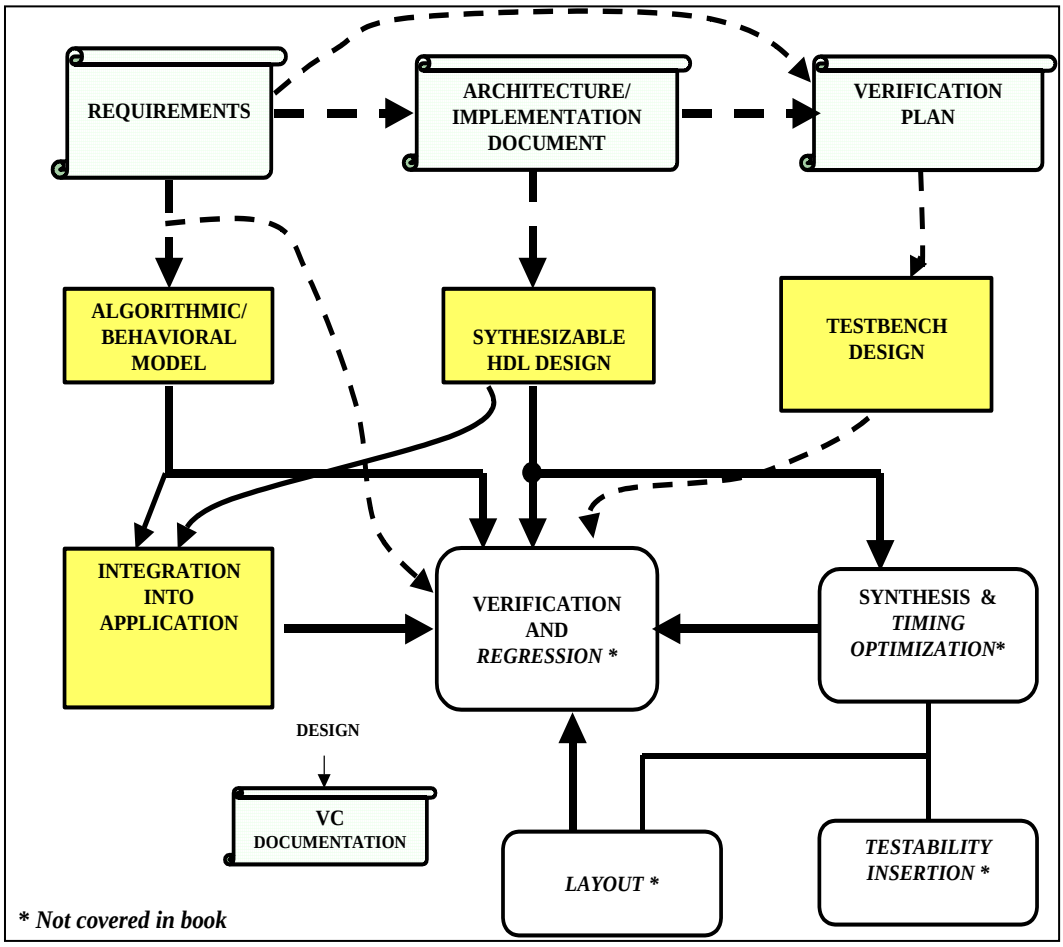


Figure 1.1-2 Component Design Processes

Requirement specification is necessary prior to proceeding with design implementation . It defines operations and required interfaces, but not the implementation .

A design typically starts with a **requirement specification** that defines the required operations and interfaces of the component, but not its implementation. This step is necessary to reaffirm the requirements and to remove any ambiguities between the original goals and the implementation. A well-reviewed requirement document brings all personnel involved in the project into an agreement as to what the design (Subblock, FPGA, or ASIC) must do, but **not how it is implemented.** The requirement document can be defined in many formats, and may include English descriptions, behavioral modeling definitions, or language specific definitions such as *C, C++, MathCad, VHDL, Verilog, SPEC C*[1], etc. Without this document, implementation designs typically must be iterated several times, or completely redone because the objectives of the design (i.e., the requirements) were not specified. That haphazard technique is often called the SPIRAL coding method where the design is hacked, iterated, and generally poorly documented until it meets requirements that are defined on the fly.

Avoid using the SPIRAL coding methodology.

Behavioral Model may validate the requirements

Once the requirements are defined, reviewed, and agreed upon, an algorithmic, and not necessarily cycle-accurate **behavioral model** of the design, may be started (if not done during the design requirements phase) and studied for further validation of the requirements.

Architectural implementation document defines subblocks, data flow, control flow, and reusable subblocks. Behavioral models may be used as source for the synthesizers

Following the requirement document, the **architectural implementation** document can be started. This document identifies the subblocks, data flow, and control flow of the design. These subblocks may also be other components or IPs. This step is necessary to reaffirm how the design will be partitioned, and architectured from a hardware viewpoint to meet the requirements. Again, this step, and the critical peer review of this step is necessary prior to starting a single line of synthesizable behavioral or RTL coding. Otherwise, the coding will typically be iterated several times, or completely redone because the objectives of the design and requirements were not implemented. It is common to mistake the requirement document from the architectural document, or to even accept the architectural implementation document as the requirement document. Those two documents serve different purposes, and it is good practice to create them separately.

---

1 *Spec C: specification Language and Methodology*, Daniel D Gajski, Jianwen Zhu, Kluwer Academic Publishers 2000, ISBN 0-7923-7822-9

With the advent of new compiler technology, it is now possible to directly synthesize higher-level definitions from behavioral HDL[2] or C[3] into either lower level RTL code or netlists. The inclusion of these tools into the implementation process needs to be considered during the architectural implementation phase.

Verification plan defines functional verification steps to insure design compliance

The **verification plan** defines how the component will be verified to insure that the design meets the requirements. The verification plan makes use of the requirement document for the source of requirements, and the implementation document for the definition of the interfaces of the components, subblocks, or ASIC. It extracts from the requirement document the features to be verified. It also defines the transactions and types of vectors (directed or random) to be applied to the design, the methods of test vector applications, and the techniques used to verify compliance to the functional specifications.

The synthesizable **behavioral** and **RTL designs** make use of the architectural requirement document as the source of design requirements. As each piece of the subblock is defined, it is recommended that the code be checked with both linting[4] and synthesis tools. The linting tool identifies errors in coding rules and style, and provides design warnings. A good synthesizer provides linting information, including synthesis-coding violations. In addition, the synthesizer provides information about the design including 1) identification of the registers, 2) unused inputs and outputs, 3) write-only hardware that gets optimized out, 4) and graphical views of the design and interconnects for use as a sanity check of the hardware inferred by the HDL. The goals of using the synthesis toolset at this phase of the design process is only to help in the debug and understanding of the design, but not necessarily to optimize the design for performance, unless blatant inadequacies are observed.

To maintain interoperability among commercial tools, it is important that the RTL design abides by coding rules and design style guides such as the *IEEE 1076.6 VHDL RTL for synthesis guidelines*[5], *IEEE P1364.1 Standard for Verilog® Register Transfer Level Synthesis,* and *Nonblocking Assignments in Verilog*

2 Example: YXI Y Explorations, Inc., http://www.yxi.com/

3 C Level Design, http://www.cleveldesign.com/

4 Examples: http://www.novas.com, http://www.Transeda.com

5 http://www.vhdl.org/siwg

Synthesis, Coding Styles that Kill[6]. Deviations from those standards need to be documented with rationales. Other proprietary guidelines, such as coding and documentation styles, are necessary to achieve design consistency and potential reuse.

Testbench makes use of requirements, verification plan, and implementation document

The **testbench design** typically occurs (or should occur) concurrently with the RTL design by a verification engineer. The testbench is written for the verification of the subblocks, the integration of subblocks, and the component design. The testbench is based on the verification plan, requirement and implementation documents. The purpose of the testbench is to provide an environment to verify that the implemented design meets the requirements.

Verification ensures design accuracy

The **verification** phase makes use of the testbench and the designs under test, including the source models (e.g., behavioral, RTL, C), the synthesized models, and any application software. Verification is a very demanding and important process to insure that the design functions correctly in the intended system.

Back-end processes must meet requirements

Synthesis, testability, layout, and timing analysis are important back-end processes necessary for the fabrication of the design to the desired requirements.

[6] http://www.deepchip.com/items/0347-01.html
http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1a.pdf
(included on CD in Verilog subdirectory for user's convenience)

2 REQUIREMENT SPECIFICATION

This section describes the requirement specification for an RS232 UART design. The purpose of this chapter is to demonstrate, by example, a model of a requirement specification. A serious attempt was made to include topics pertinent to all components. However, every design has its own peculiarities, and additional entries need to be identified for specific designs. The reviewers may disagree with the features presented in this specification. If the reviewers have comments, then that specification has met its goals of providing a forum for a good design review. Again, the purpose of this book is to demonstrate methodologies and provide techniques using the UART model as an model, rather than just designing a UART. There are several methods to define requirement specifications including English language and programming languages such as *VHDL, C, C++, SPEC C, MathCad, Foresight*[1], etc. Programming languages have the advantages of creating *an executable specification*. However, there are several requirements that are difficult to express in a programming language. For example, operating conditions (e.g., clock speed, operating range, radiation levels, size, reliability, reset values, latency, packaging) are design attributes rather than processing algorithms, which cannot be expressed in a programming language that does not support attributes. In general, it is best to use an **English specification** document as a baseline to cover all the requirements of the design. This English document can reference other *executable specification* models for the definition of specific

1 http://www.foresight.com

algorithms that are best expressed mathematically. This creates a universal document that does not require knowledge of programming languages as the baseline for the requirements, but allows the inclusion of other requirement models for further definitions. The English specification used in this book is modeled after a modified structure proposed in *Military Standard MIL-STD-490A*[2], *4 June 1985, Specification Practices* document. This is a Department of Defense (DoD) document approved for public release with unlimited distribution. *This Military Standard sets forth practices for the preparation, interpretation, change, and revision of program-peculiar specifications prepared by or for the Departments and Agencies of the Department of Defense. It defines good practices for defining requirements*.

Another source of specification modeling template used in this sample requirement document is MIL-STD-1553 standard (*Aircraft Internal Time Division Command/Response Multiplex Data Bus*), another DoD document approved for public release with unlimited distribution.

2.1 LANGUAGE

Engineers are notorious for being poor writers. This section emphasizes important English points to consider when writing a document because they typically are major sources of errors and poor style.

STYLE: MIL-STD-490 section 3.2.3, *language style,* states the following:
The paramount consideration in a specification is its technical essence, and this should be presented ***in language free of vague and ambiguous terms*** *and using the simplest words and phrases that will convey the intended meaning. Inclusion of essential information shall be complete, whether by direct statements or references to other documents. Consistency in terminology and organization of material will contribute to the specification's clarity and usefulness. Sentences shall be as short and concise as possible. Punctuation should aid in reading and prevent misreading. Well-planned word order requires a minimum of punctuation. When extensive punctuation is necessary for clarity, the sentence(s) shall be rewritten. Sentences with compound clauses shall be converted into short and concise sentences.*

2 Mil-STD-490A is included on CD

Commonly used words and phrasing. *Certain words and phrases are frequently used in a specification. The following rules shall be followed:*

a. Referenced documents shall be cited thus "conforming to ..." "as specified in ..." or "in accordance with ...".

b. "Unless otherwise specified" shall be used to indicate an alternative course of action. The phrase shall always come at the beginning of the sentence, and if possible, at the beginning of the paragraph. This phrase shall be used only when it is possible to clarify its meaning by providing a reference such as to Section 6 of the specification for further clarification in the contract or order or otherwise.

c. When making reference to a requirement in the specification and the requirement referenced is rather obvious or not difficult to locate, the simple phrase "as specified herein" is sufficient and should be used.

d. The phrase "... to determine compliance with ..." or "... to determine conformance to ..." should be used in place of "... to determine compliance to ...". In any case use the same wording throughout.

e. In stating positive limitations, the phrase shall be stated thus: "The diameter shall be no greater than ...".

The emphatic form of verb shall be used throughout the specification*; i.e., state in the requirements section that "The indicator shall be designated to indicate ...", and in the section containing test provisions "The indicator shall be turned to zero and 230 volts alternating current applied." For specific test procedures, the imperative form may be used provided the entire method is preceded by "the following tests shall be performed," or related wording. Thus, "Turn the indicator to zero and apply 230 volts alternating current."*

Use of "shall," will," "should," and "may". *Use "shall" whenever a specification expresses a provision that is binding. Use "should" and "may" wherever it is necessary to express non-mandatory provisions. "Will" may be used to express a declaration of purpose on the part of the contracting agency. It may be necessary to use "will" in cases where the simple future tense is required, i.e., power for the motor will be supplied by the ship.*

THAT verses WHICH: Another common mistake is the misunderstanding in the

use the pronoun *THAT* and *WHICH*. The Merriam-Webster[3] dictionary defines:

> **that** *pron, pl* **those** : 1: the one indicated, mentioned, or understood <*that is my house*> 2: the one farther away or first mentioned <*this is an elm, that's a maple*> 3 : what has been indicated or mentioned <*after that, we left*> 4 : the one or ones : IT, THEY <*those who wish to leave may do so*>
>
> **which** *pron* 1 : which one or ones <*which is yours*> … 3 — used to introduce a relative clause and to serve as a substitute therein for the noun modified by the clause <*the money, which is coming to me,*>

Microsoft *Word* grammar check explains the application rule as: *"if the marked group of words is essential to the meaning of the sentence, use **that** to introduce the group of words. Do not use a comma. If these words are not essential to the meaning of your sentence, use **which** and separate the words with a comma"*. For example,

> Books, which are generally expensive, can be purchased over the Internet.
> *Note: The qualifier "which are generally expensive" is not essential to the meaning of the sentence.*
>
> The book that describes VHDL guidelines is the Cohen book.
> *Note: The qualifier "that describes VHDL guidelines" is essential in the sentence.*

In some sentence structures, the qualifier *THAT* can be deleted, thus shortening the sentence, without affecting the meaning of the sentence. For example,

> Spec C is a language that is suitable for specifying systems. // can delete "that is"
>
> Spec C is a language suitable for specifying systems.

[3] © 1995 Zane Publishing, Inc. The Merriam-Webster Dictionary © 1994 by Merriam-Webster, Incorporated

2.2 UART REQUIREMENT SPECIFICATION

Header page Pertinent logistics data about the requirements

REQUIREMENTS FOR AN ASYNCHRONOUS OR SYNCHRONOUS 8 TO 32 BIT Universal Asynchronous Receiver/Transmitter

Document #:
Release Date: __
Revision Number: _____
Revision Date: __
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 requirement specification 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 specification

This document establishes the requirements for a component that provides a bridge between a microprocessor interface and a transmit/receive asynchronous or synchronous, parameterized eight to thirty-two bit serial interface, emulating a UART-like protocol.

Target audience

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

1.2 Purpose

Purpose of specification

These requirements shall apply to a modified Universal Asynchronous Receiver/Transmitter (UART) interface for inclusion as a component.

1.3 Classification

System, hardware, software

This document defines the requirements for a hardware design.

2. DEFINITIONS

Terms used in this document. Organize definitions alphabeticall y

2.1 Asynchronous transmission[4]

The transmitted clock is not sent to the receiving logic. Instead, asynchronous transmission relies on some other mechanism to synchronize the receiver to the data stream. In the case of a UART, *asynchronous transmission relies on the use of a start bit and stop bit(s), in addition to the bits representing the character (and an optional parity bit), to distinguish separate characters.*

2.2 Baud rate[4]

The baud rate is the number of events, or signal changes, that occur in one second.

2.3 DTE

Data Terminal Equipment, such as terminals.

2.4 DCE

Data Communication Equipment, such as modems.

2.5 Framing Error

A condition where the received data stream is not properly framed between a START bit and a STOP bit.

2.6 Overrun Error[4]

An error that occurs when a device receiving data cannot handle or make use of

the information as rapidly as it arrives.

2.7 Parity[4]
The quality of sameness or equivalence, in the case of computers usually referring to an error-checking procedure in which the number of ONEs must always be the same—either even or odd—for each group of bits transmitted without error.
Even parity The number of ONEs in each successfully transmitted set of bits (data plus parity) must be an even number.

Odd parity The number of ONEs in each successfully transmitted set of bits (data plus parity) must be an odd number.

No parity No parity bit is used.

Space parity A parity bit is used and is always set to zero.

Mark parity A parity bit is used and is always set to one.

2.8 Start Bit
A low level bit to indicate the start of a transmission. The receiver uses this negative transition to synchronize its internal clock to the transmitted data.

2.9 Stop Bit
A high level bit to indicate the end of a transmission, and to guarantee that a new START bit will be initiated with a negative edge. STOP bits are also asserted when no data is sent.

2.10 Synchronous Transmission (with UART) [4]
A UART that supports synchronous serial transmission, where the sender and receiver share a timing signal.

2.11 Underrun Error (with UART)
An error that occurs when the CPU attempts to read data that was not yet received.

2.12 Word (with UART)
A data element that is one, two, or four bytes in size, depending upon the parameterization of the component.

3. APPLICABLE DOCUMENTS

Sources of applicable documents

3.1 Government Documents

3.1.1 TIA/EIA-232-F

Interface between Data Terminal Equipment and Data Circuit-

Standard and government specifications and requirements

[4] *Microsoft Press® Computer and Internet Dictionary* © & 1997, 1998 Microsoft Corporation.

Non-government

Terminating Equipment Employing Serial Binary Data Interchange (ANSI/TIA/EIA-232-F-1997), October 14, 1997.
http://www.tiaonline.org/standards/search_results2.cfm?document_no=TIA/EIA-232-F
Electronic Industries Alliance, 2500 Wilson Boulevard
Arlington, VA 22201-3834, (703) 907-7500

3.2 Non-government Documents

Technical Aspects of Data Communication, John E. McNamara, 1977, Digital Press.

A Practical Guide to RS-232 Interfacing, Lawrence E. Hughes. Mycroft Labs, Inc., P.O. Box 6045, Tallahassee, FL 32301

RS-232 Protocols and Computer Networks, Dr. D. Koren / Tel-Aviv University, http://www.rad.com/networks/1995/rs232/rs232.htm

3.3 Executable Specifications

None.

Essential requirements and descriptions that apply to performance, design, reliability, etc

4. ARCHITECTURAL OVERVIEW

4.1 Introduction

The UART component shall represent a design written in an HDL (VHDL or Verilog) that can easily be incorporated into a larger design. The UART shall provide the function of a bridge between a processor (CPU) and an interface that meets an RS-232 like protocol. Figure 4.1 represents a high level view of the interfaces. A variation to the standard RS-232 shall be incorporated for the transfer of 8, 16, or 32-bit data blocks, instead of the standard 7 or 8-bit transfer imposed by the standard. This variation shall maintain the START/STOP bit synchronization bits, but shall allow the parameterization of the data word to 8, 16, or 32 bits. In addition, this UART shall support either the asynchronous transmission, as defined by the standard, or synchronous transmission where the synchronization signals are externally supplied to the component. The selection between synchronous or asynchronous transmission shall be defined as a parameter before the elaboration of the design into hardware. The UART shall include the following features:

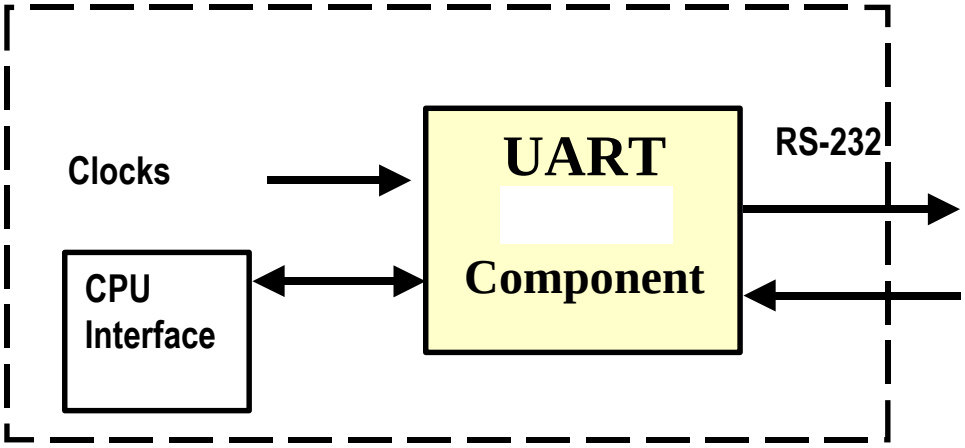


Figure 4.1 High Level View of the UART Interfaces

1. Parameterized storage space for both the transmitter and receiver buffers
2. Interrupt controls with programmable masking capability.
3. CPU controlled characteristics including parity definition.
4. Framing and parity error detection of received messages.
5. Overrun and underrun error detection.
6. Full implementation of modem control functions

The CPU interface shall be a simple interface that emulates an asynchronous random access memory (RAM) device with *address*, *read* control, *write* control, *enable* control, *data in*, *data out* and *Tri-state* control. This will allow the adaptation of this CPU interface to a specialized processor interface through bridging logic.

The UART shall support a maximum baud rate of 25 Mbauds in synchronous mode, and 1.5 Mbauds in asynchronous mode.

Information about how data is moved across the UART (i.e., UART in the system environment)

4.2 System Application

The UART can be applied in a variety of system configurations. Figure 4.2-1 demonstrates one such configuration where the UART interfaces on one side to a host controller or another controller. On the other interface, the component can connect to a modem for a link onto a telephone line or a network. Figure 4.2-2 shows another system application of the UART used as a serial hardwired interface between two subsystems.

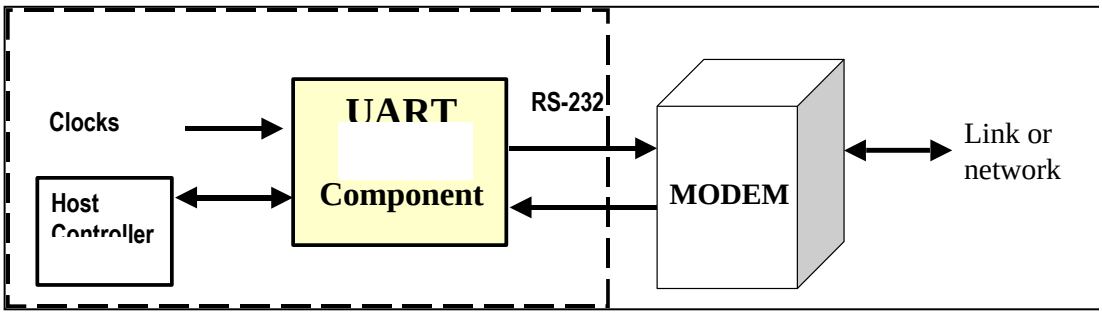


Figure 4.2-1 UART Applications with a Modem

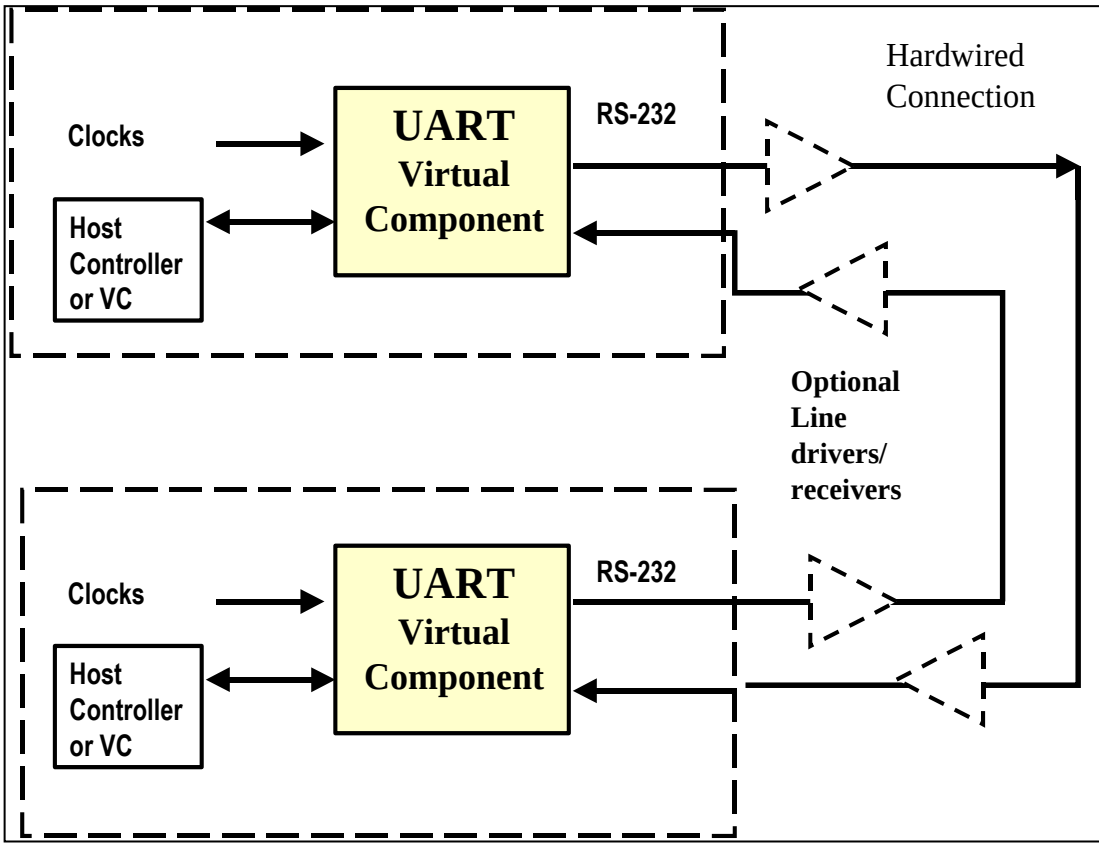


Figure 4.2-2 Hardwired Application of UART

Interfaces of VC as seen from the pinout viewpoint

5. PHYSICAL LAYER

The physical hardware interfaces shall be as shown in Figure 5.0. These interfaces are partitioned as *RS-232* for the serial port interfaces, and *CPU* for the processor and clock interface.

Show all ports that are anticipated as requirements

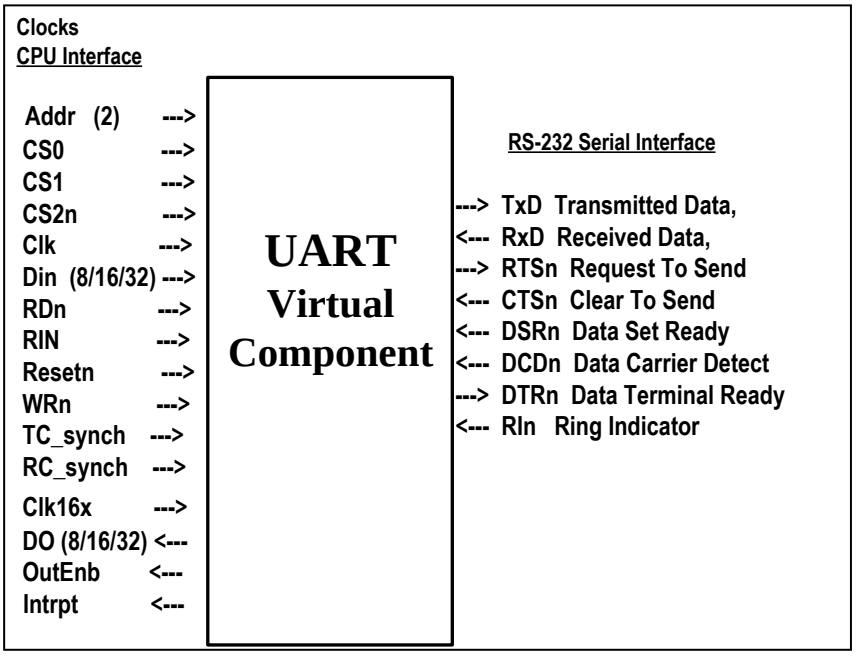


Figure 5.0 Interfaces of the UART

Organize ports by function.

5.1 Interface Port Description

5.1.1 RS-232 Serial Interface

5.1.1.1 TxD, *Transmit Data*

Describe each port, including name, direction (source -> destination), size, active level, and description.

Direction: *Output, DTE -> DCE*; Size: *1 bit*; Active level: *High*

Serial data to be sent from the DTE to the DCE. The DTE shall hold this line at logic '1' when no data are being transmitted. An *ON* (logic '0') condition must be present on all of the following signals before data can be transmitted on the TxD signal:

Request To Send, Clear To Send, Data Set Ready, Data Terminal Ready.

5.1.1.2 RxD, *Receive Data*

Direction: *Input, DCE -> DTE*; Size: *1 bit*; Active level: *High*

Serial data to be received from the DCE to the DTE. This port will be held at logic '1' when no data are being transmitted.

Add pin numbering if required

ı, *Request-To-Send*

Direction: *Output, DTE -> DCE*; Size: *1 bit*; Active level: *Low*

This signal shall enable the transmission circuits. The DTE shall assert this signal (logic '0') when it wants to transmit to the DCE. This signal, in combination with the *Clear-To-Send* signal, coordinates data transmission between the DTE and the DCE. A logic '0' on this line keeps the DCE in transmit mode. The DCE will receive data from the DTE and transmit it on to the communication link. The *Request-To-Send* and *Clear-To-Send* signals relate to a half-duplex telephone line. A half duplex line is capable of carrying signals on both directions but only one at a time. When the DTE has data to send, it shall assert *Request-To-Send* and then waits until the DCE changes from receive to transmit mode. This "On" to "Off" transition instructs the DCE to move to "transmit" mode, and when a transmission is possible, the DCE will set *Clear-To-Send* (to logic '0') to start the transmission. On a full duplex line, like a hard-wired connection, where transmission and reception can occur simultaneously, the *Clear-To-Send* and *Request-To-Send* signals may be held to a constant "On" level. An "On" to "Off" transition on this line instructs the DCE to complete the transmission of data that is in progress, and to move to a "receive" (or "no transmission") mode. The *Request-To-Send* shall be asserted by the UART when the UART has data in its transmission buffer. The RTSn can be deasserted any time after the START bit is sent.

5.1.1.4 CTSn, *Clear-To-Send*

Direction: *Input, DCE -> DTE* ; Size: *1 bit*; Active level: *Low*

When this signal is active (logic '0'), it shall inform the DTE that it can start to transmit on the *TxD* port. When this signal is *ON* (logic '0') and the *Request To Send, Data Set Ready,* and *Data Terminal Ready* are all *ON* (logic '0'), the DTE is assured that its data will be sent to the communications link. When *OFF* (logic '1'), it is an indication to the DTE that the DCE is not ready, and therefore data should not be sent. When the *Data-Set-Ready* and *Data-Terminal-Ready* signals are not implemented, such as a local connection not using a modem, the *Clear-To-Send* and *Request-To-Send* signals will be sufficient to control data transmission. In that case, the *Data-Set-Ready* and *Data-Terminal-Ready* will be hardwired at the ports of the component to the active (logic '0') state.

5.1.1.5 DSRn, *Data-Set-Ready*

Direction: *Input, DCE -> DTE*; Size: *1 bit*; Active level: *Low*

This signal is active when at logic '0', and informs the DTE that the DCE communication channel is available (i.e., in an automatic calling system, the DCE (modem) is not in the dial, test or talk modes and therefore is available for transmission and reception). It reflects the status of the local data set, and does not indicate that an actual link has been established with any remote data equipment.

5.1.1.6 DCDn, *Data-Carrier-Detect*

Direction: *Input, DCE -> DTE;* Size: *1 bit*; Active level: *Low*

The DCE uses this line to signal the DTE that a good signal is being received (a "good signal" means a good analog carrier, that can ensure demodulation of

received data).

5.1.1.7 DTRn, *Data-Terminal-Ready*

Direction: *Output, DTE -> DCE*; Size: *1 bit*; Active level: *Low*

When ON (logic '0'), this signal shall indicate that the DTE is available for receiving. This signal must be "On" before the DCE can turn Data Set Ready "On", thereby indicating that it is connected to the communications link. The Data-Terminal-Ready and Data-Set-Ready signals deal with the readiness of the equipment, as opposed to the Clear-To-Send and Request-To-Send signals that deal with the readiness of the communication channel. When "Off", this signal will cause the DCE to finish any transmission in progress and to be removed from the communication channel. The value of the DTRn signal shall be defined by the CPU. It shall indicate to the DCE that the CPU is in a ready mode to process data.

5.1.1.8 RIn, *Ring Indicator*

Direction: *input, DCE -> DTE*; Size: *1 bit*; Active level: *Low*

On this line, the DCE signals the DTE that there is an incoming call. This signal will be maintained *off* at all times, except when the DCE receives a ringing signal.

5.1.2 CPU Interface

5.1.2.1 Addr, *Address*

Direction: *input, CPU -> UART ;* Size: *2 bits*; Active level: *High*

The *Addr* ports shall represent two bits of the CPU address to define the personality characteristics of the UART and the DCE control information. The *Addr* ports shall also be used to read information about the DCE interfaces and the UART pending interrupts. The *Addr* bits shall be used in conjunction with the chip select bits (*CS0, CS1, CS2n*) and the READ (*RDn)* and WRITE (*WRn*) control signals.

5.1.2.2 CS0, *Chip Select 0*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

The *CS0* pin shall represent one of three chip select pin to enable a READ or WRITE operation into the UART. Access to the UART shall require that the *CS0 = High, CS1 = High, and CS2n = Low.*

5.1.2.3 CS1, *Chip Select 1*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

The *CS1* pin shall represent one of three chip select pin to enable a READ or WRITE operation into the UART. Access to the UART shall require that the *CS0 = High, CS1 = High, and CS2n = Low.*

5.1.2.4 CS2n, *Chip Select 2*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *Low*

The *CS2n* pin shall represent one of three chip select pin to enable a READ or WRITE operation into the UART. Access to the UART shall require that the *CS0 = High, CS1 = High, and CS2n = Low.*

5.1.2.5 Din, *Data Input*

Direction: *input, CPU -> UART;* Size: *8/16/32 bits*; Active level: *High*

The *Din* shall represent the CPU data to be asserted into the UART. The width of that data shall be parameterized in the component design to accommodate at least the following widths: 8, 16, and 32 bits. That same width shall also be used to define the UART data widths. Therefore, an 8-bit CPU width shall also define an 8-bit UART data width, whereas a 32-bit CPU width shall specify a 32-bit UART data width.

5.1.2.6 RDn, *Read*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *Low*

The *RDn* shall represent a read strobe signal, used for reading data and status from the UART. The *RDn* signal shall be operational only when the chip select bits are set to the "selected" values. It shall be an operational error if the RDn and WRn are both asserted.

5.1.2.7 Resetn, *Reset*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *Low*

The *Resetn* signal shall represent the master reset for the *UART*. The *Resetn* signal shall be synchronous to the system clock. Once activated, the *Resetn* signal shall force the UART into a benign state and in an idle mode with no data in the UART buffers (transmit or receive). A benign state or idle mode shall be represented by the hardware as logical zero. This requirement shall provide the support of scanable registers where the reset condition can be forced through a serial scan interface with zeros being forced into the scan serial stream.

5.1.2.8 WRn. *Write*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *Low*

The *WRn* signal shall represent a write strobe signal, used for writing data and control information to the UART. The *Wrn* signal shall be operational only when the chip select bits are set to the "selected" values. It shall be an operational error if the RDn and WRn are both asserted.

5.1.2.9 DO, *Data Output*

Direction: *output, UART -> CPU;* Size: *8/16/32 bits*; Active level: *High*

The *DO* shall represent the UART data output to be sent to the CPU. The width of that data shall use the same parameter as the *Din* width parameter in the component design. It shall accommodate at least the following widths: 8, 16, and

32 bits

5.1.2.10 OutEnb, *Output Enable*

Direction: *output, UART -> CPU;* Size: *1 bit*; Active level: *High*
The *OutEnb* signal defines the timing at which the UART *DO* is valid. It shall allow a buffer, external to the component, to enable the *DO* data onto a tri-state bus. This feature will allow the component to be integrated into a larger subsystem with a tri-state CPU data interface.

5.1.2.11 Intrpt, *Interrupt*

Direction: *output, UART -> CPU;* Size: *2 bits*; Active level: *High*
Bit 1 (MSB) shall represent the interrupt from the transmit hardware.
Bit 0 (LSB) shall represent the interrupt from the receive hardware.

The sources for the transmit interrupts shall be identified in the pending interrupt register (PIR), as shown below. A READ of the transmit PIR shall reset the register, but any new PIR value that occurs during the READ cycle shall set the PIR for that value. Upon the activation of a RESET (soft or hard), the transmit PIR register must be reset to the inactive state. The sources for the transmit interrupts shall be maskable with a CPU loadable mask register.

PIR(5) Transmit buffer error: *write to full buffer*
PIR(4) Transmit buffer OFF of *non-Empty* state reached (i.e., just emptied)
PIR(3) Transmit buffer OFF of *Almost Empty* state reached (i.e., data sent to serializer)
PIR(2) Transmit buffer OFF of *Half-Full* state reached (i.e., data sent to serializer)
PIR(1) Transmit buffer OFF of *Almost Full* state reached (i.e., data sent to serializer)
PIR(0) Transmit buffer OFF of *Full* state reached (i.e., data was to serializer)

The sources for the receive interrupts shall be as shown below. A READ of the transmit PIR shall reset the register, but any new PIR value that occurs at the READ cycle shall set the PIR for that value. Upon the activation of a RESET (soft or hard), the receive PIR register must be reset to the inactive state. The sources for the receive interrupts shall be maskable with a CPU loadable mask register

PIR(7) Receive buffer error: *framing error*
PIR(6) Receive buffer error: *Parity error* , or read of an *empty buffer*
PIR(5) Receive buffer error: *Overrun error*, or read of an *empty buffer*
PIR(4) Receive buffer *Not-Empty* state reached
PIR(3) Receive buffer *Almost Empty* state reached
PIR(2) Receive buffer *Half-Full* state reached
PIR(1) Receive buffer *Almost Full* state reached
PIR(0) Receive buffer *Full* state reached

5.1.3 Clock Interface

5.1.3.1 Clk, *System Clock*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

The *Clk* shall represent the system clock for all synchronous transfers. The design shall support a minimum system clock frequency from 0 HZ to 25 MHz.

5.1.3.2 TC_synch, *Transmit Clock for Synchronous Transmission*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

The *TC_synch* shall represent the synchronous transmit clock enable signal operating at the transmit clock frequency. The *TC_synch* signal shall be a derived signal from the system clock (*CLk*), and shall be active for one system clock period for every baud period, as shown in Figure 5.1.3.2. The selection between synchronous and asynchronous clock mode shall be defined through a parameter to be used by the synthesis process. If asynchronous transmission mode is selected, the *TC_synch* signal shall be ignored.

Include timing diagrams to emphasize requirements

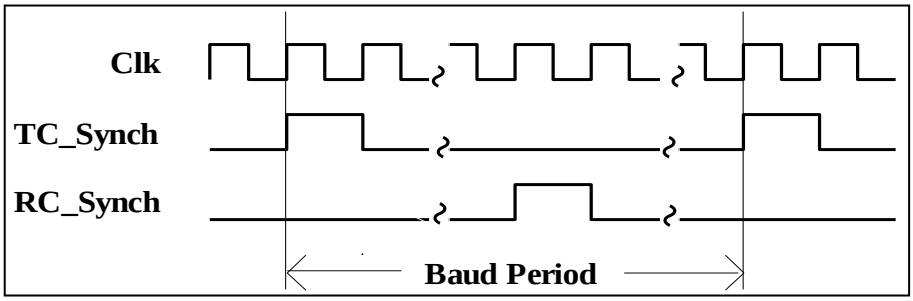


Figure 5.1.3.2 Synchronous Transmit and Receive Clock Timing

5.1.3.3 RC_synch, *Receive Clock for Synchronous Transmission*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

The *RC_synch* shall represent the synchronous receive clock enable signal operating at the receive clock frequency. The *RC_synch* signal shall be a derived signal from the system clock (*CLk*), and shall be active for one system clock period for every baud period, as shown in Figure 5.1.3.2. The selection between synchronous and asynchronous clock mode shall be defined through the same parameter as the *TC_synch* signal. If asynchronous transmission mode is selected, the *RC_synch* signal shall be ignored.

5.1.3.4 Clk16x, *Sixteen Times Clock for Asynchronous Transmission*

Direction: *input, CPU -> UART;* Size: *1 bit*; Active level: *High*

In asynchronous transmission mode, the *Clk16x* shall represent the input clock used for internal baud rate generation. The *Clk16x* clock shall be at a rate sixteen times the baud rate. The *Clk16x* clock shall not necessarily be synchronous to the system clock. In synchronous transmission mode, the *Clk16x* signal shall be ignored.

6. PROTOCOL LAYER

A serial message shall consist of the following bits transmitted at the requested baud rate:

Provide definition of interface protocol that represents a requirement

START: This bit is set to a *low* state to initiate bit synchronization of the message at the receiver.

Data Word: These bits shall represent the data bits of the word, as defined by the word size parameter of the component. Bits shall be sent out onto the bus least-significant bit (LSB) first, followed by the next LSB, through to the most-significant bit (MSB) last. For each bit, a *high* level shall represent a logical ONE, and conversely, a *low* level shall represent a ZERO. The word size shall be parameterized and used in the build of the design.

Identify algorithm if that algorithm is a requirement

PARITY: If parity is enable, then this bit shall represent the even or odd parity of the data word. The definition of *even* or *odd* parity shall be defined by the CPU as a control parameter. For *even* parity, the parity bit shall be such that the number of ONEs in the word message and the parity bit will be an even number. For *odd parity*, the parity bit shall be such that the number of ONEs in the word message and the parity bit will be an odd number. The parity bit shall be the exclusive OR of the desired parity mode ('0' for *even, or* '1' for *odd)* and the data word. If parity is disabled, then this bit shall be omitted.

STOP Bit: This bit is set to a *high* state to provide message-framing indication for use in bit synchronization at the receiver.

Figure 6.1-1 demonstrates the interface format of the serial data.

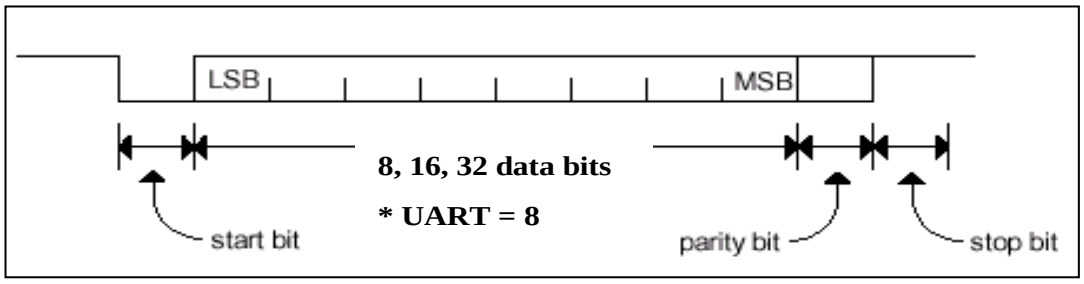


Figure 6.1-2 Interface Format of Serial Data

If all the conditions for transmission are satisfied, and no transmission is in progress, then a new serial message shall be started within two baud cycles. If a message is queued no later than two baud cycles from the completion of an on-going message, and all the conditions for the new message are satisfied, then the new message shall immediately follow the on-going message with no additional STOP bits between the two messages.

7. ROBUSTNESS

7.1 Error Detection

The errors identified in the following subsections shall be reported.

7.1.1 Receive Framing Error

A framing error shall be reported if a message is started, but is not terminated with a STOP bit. Whenever possible, this error shall be applicable to the correspondent received message.

7.1.2 Receive Parity Error

A parity error shall be reported if parity check is enabled and parity fails on the received data. This error shall be applicable to the correspondent received message.

7.1.3 Receive Buffer Overrun Error

A buffer overrun error shall be reported if data is received, but the receive buffer is full. When this error occurs, the received word shall be rejected.

7.1.4 Transmit Buffer Overrun Error

A transmit buffer overrun error shall be reported if the CPU attempts to write into a full transmit buffer.

7.2 Error Handling

No error correction or automatic transmission retries shall be provided. All errors shall be reported to the pending interrupt registers, and to two maskable interrupts as specified herein.

8. HARDWARE AND SOFTWARE

8.1 Fixed Parameterization

The UART shall provide for the following parameters used in the definition of the implemented hardware (i.e., during hardware build):

Identify the parameters that affect the produced design.

Word size: Word size shall be user defined, but must accommodate the following sizes: 8, 16, or 32 bits/word. This word size shall be used to identify the width of the CPU data interface, and the size of the serial message transmitted and received by the UART. For example, a 16-bit word size shall represent a 16-bit CPU interface and a 16-bit serial word

message.

Buffer Depth: This parameter shall define the storage space for the transmitter and receiver buffers. The same parameter shall be used for each of the buffers (transmit and receive). This will reduce the number of CPU interrupts for the received data, and will allow the CPU to fill the buffer for transmit data before continuing other tasks. Buffer depth shall be of size divisible by two, and of size two or greater.

Buffer Almost-Empty Threshold: This threshold shall be used in the identity of the buffer almost-empty flag. It shall be used in the generation of interrupts.

Buffer Almost-Full Threshold: This threshold shall be used in the identity of the buffer almost-full flag. It shall be used in the generation of interrupts.

Synchronous or Asynchronous Transmission Mode: This parameter shall identify the transmission and receive mode of the UART as synchronous or asynchronous transmission.

8.2 Software Interfaces

Identify the interfaces and the machine as seen from the software.

The CPU shall write into the UART information to control the modem modes, data for transmission, and interrupt masks. The CPU shall also read from the transmit and receive PIR, and modem status. Table 8.2 represents a summary of the functions as seen from software.

Table 8.2 Summary of Software Addresses and Functions

Addr	RdF	WrF	OPERATION
00	0	1	Read Modem status
00	1	0	Write Modem Control and Parity Definition
01	0	1	Read Receive PIR
01	1	0	Write Receive buffer control
10	0	1	Read transmit PIR
10	1	0	Write transmit buffer control
11	1	0	Write transmit Data
11	0	1	Read Receive data

8.2.1 Address "00", CPU READ, Modem Status

A CPU READ at address "00" shall return to the CPU the following four modem status bits (little endian, bit '0' is the LSB). Spare bits shall be of value ZERO.

Identify each bit read by the address

Bit 3 : RINn, Ring indicator
Bit 2 : CTSn, Clear to send
Bit 1 : DSRn, Data set ready
Bit 0 : DCDn, Data carrier detect

8.2.2 Address "00", CPU WRITE, Modem Control / Parity Control

A CPU WRITE at address "00" shall characterize to the UART the modem control and settings. These include:

Identify each bit to be written at the specified address

Bit 2 : Invert of DTRn, Data terminal ready
('0' = not ready, '1' = ready)
Bit 1 : Parity disable/enable
('0' = no parity, '1'= parity)
Bit 0 : Parity bit
('0'= Even parity, '1'= Odd parity)

8.2.3 Address "01", CPU READ Receive PIR

A CPU READ at address "01" shall return the status of the UART receive pending interrupt register (PIR). The receive PIR shall be automatically reset following a READ (i.e., Read and clear) of that register. However, if a new setting is to be set at the same read-and-clear cycle, the new setting shall be asserted following the READ action. The PIR shall only be reset via a READ at address "01", or via a hard reset, or soft reset of the receive section (see 8.2.4). The data format shall be as follows. Spare bits shall be of value ZERO.

Bit 7: Receive framing error

This error shall be applicable to the received data being read.

Bit 6: Receive parity error

This error shall be applicable to the received data being read.

Bit 5: Receive FIFO buffer overrun or read of empty buffer error

This bit shall be set upon a receive overrun error, or a CPU read of an empty buffer.

Bit 4: Receive FIFO Buffer Not-Empty

This bit shall '1' if the number of words in the receive buffer is ONE or more than ONE.

Bit 3 : Receive FIFO Buffer Almost-Empty

This bit shall be '1' if the number of words in the receive buffer is equal to, or is above the predefined almost-empty threshold. For example, if the almost-empty threshold is set to 2, then the almost-empty bit shall be set to '1' when the number of words in the receive buffer equals, or exceeds 2.

Bit 2: Receive FIFO Buffer Half-Full

This bit shall be '1' if the number of words in the receive buffer is equal to, or is above the predefined half-full threshold. For example, if the buffer size is 4, then the half-full threshold is set to 2, and the half-full bit shall be set to '1' when the number of words in the receive buffer equals, or exceeds 2.

Bit 1: Receive FIFO Buffer Almost-Full

This bit shall be '1' if the number of words in the receive buffer is equal to, or is above the predefined almost-full threshold. For example, if the almost-full threshold is set to 3, then the almost-full bit shall be set to '1' when the number of words in the receive buffer equals, or exceeds 3.

Bit 0: Receive FIFO Buffer Full

This bit shall be '1' if the number of words in the receive buffer is equal to the maximum size of the receive buffer. For example, if the receive buffer size is set to 4, then the almost-full bit shall be set to '1' when the number of words in the receive buffer equals 4.

8.2.4 Address "01", CPU WRITE, Receive Buffer Control

The CPU WRITE of the receive buffer control at address "01" shall define the following parameters for the receive buffer. The data format shall be as follows:

Bit 6: (MSB) Software reset of the receiver function

A logic '1' shall reset the receive function to the idle state, and all received messages aborted. When this bit is set to logic '1', the CPU must not send another transaction (read or write) to the component for a minimum of five clock cycles to allow the reset to take effect. The CPU shall set this bit to a logic '0' to enable operation of the receive logic.

Bit 5: Error interrupt enable

A logic '1' shall enable the interrupt function for a receive error that can comprise of one or more of the following: Receive framing error, receive parity error, and receive buffer overrun error. A logic '0' shall mask that interrupt.

Bit 4: Not-empty interrupt enable

A logic '1' shall enable the interrupt function for the receive not-empty. The interrupt shall be asserted when the receive buffer reaches the not-empty level. A logic '0' shall mask that interrupt.

Bit 3: Almost-empty interrupt enable

A logic '1' shall enable the interrupt function for the receive almost-empty flag. The interrupt shall be asserted when the receive buffer reaches the almost-empty level. A logic '0' shall mask that interrupt.

Bit 2: Half-full interrupt enable

A logic '1' shall enable the interrupt function for the receive half-full flag. The interrupt shall be asserted when the receive buffer reaches the half-full level. A logic '0' shall mask that interrupt

Bit 1: Almost-full interrupt enable

A logic '1' shall enable the interrupt function for the receive almost-full flag. The interrupt shall be asserted when the receive buffer reaches the almost-full level. A logic '0' shall mask that interrupt.

Bit 0: (LSB) Full interrupt enable

A logic '1' shall enable the interrupt function for the receive full flag. The interrupt shall be asserted when the receive buffer reaches the full level. A logic '0' shall mask that interrupt.

8.2.5 Address "10", CPU READ, Transmit PIR

A CPU READ at address "10" shall return the value of the UART transmit pending interrupt register (PIR). The transmit PIR shall be automatically reset following a READ (i.e., Read and clear) of that register. However, if a new setting is to be set at the same read-and-clear cycle, the new setting shall be asserted following the READ action. The PIR shall only be reset via a READ at address "10", or via a hard reset, or soft reset of the receive section (see 8.2.6). The data format shall be as follows. Spare bits shall be of value ZERO.

Bit 5: (MSB) Transmit error

This error shall indicate that the CPU attempted to write into a full buffer.

Bit 4: Transmit FIFO Buffer OFF of Non-Empty (i.e., To Empty state)

This bit shall be '1' when the number of words in the transmit FIFO buffer becomes empty from a non-empty condition, i.e., the transmit FIFO buffer became zero and can hold up to the as many new words for transmission as the transmit FIFO buffer size. For example, if the transmit FIFO buffer size is 4, then the CPU could send up to 4 words for serial transmission.

Bit 3: Transmit FIFO Buffer OFF of Almost-Empty

This bit shall be '1' when the number of words in the transmit FIFO buffer gets off the almost-empty state, i.e., the transmit FIFO buffer had reached the almost-empty value and is now one less that that value. For example, if the transmit FIFO buffer size is 8, and the almost-empty size is 3, then the OFF of almost empty flag means that the buffer was holding 3 words, but now holds one less than 3. The CPU could send up to (8 - 3 + 1), or 6 words. .

Bit 2: Transmit Buffer OFF of half-full

This bit shall be '1' when the number of words in the transmit buffer gets off the half-full from a higher level, i.e., the transmit FIFO buffer reached one level below the half-full and can hold as many new words for transmission as the transmit FIFO buffer size minus the half-full value plus one. For example, if the buffer size is 4, half-full is 2, then the CPU could send up to (4 –2) + 1, or 3 words.

Bit 1: Transmit Buffer OFF of Almost-full

This bit shall be '1' when the number of words in the transmit buffer gets off the almost-full from a higher level, i.e., the transmit FIFO buffer reached one level below the almost-full and can hold as many new words for transmission as the transmit FIFO buffer size minus the almost-full value minus one. For example, if the FIFO buffer size is 8 and the almost-full level is 6, then the CPU could send up to (8 –(6-1), or 3 words.

Bit 0: Transmit Buffer OFF Full

This bit shall be '1' when the number of words in the transmit FIFO gets off the full level, i.e., the transmit FIFO buffer flushed one word off the buffer and the CPU could send one additional word for serial transmission.

8.2.6 Address "10", CPU WRITE, Transmit Buffer Control

The CPU WRITE of the transmit buffer control at address "10" shall define the following

parameters for the transmit buffer. The data format shall be as follows:

Bit 6: (MSB) Software reset of the transmit function

A logic '1' shall reset the transmit function to the idle state, and all outgoing messages aborted. When this bit is set to logic '1', the CPU must not send another transaction (READ or WRITE) to the component for a minimum of five clock cycles to allow the reset to take effect. The CPU shall set this bit to a logic '0' to enable operation of the transmit logic.

Bit 5: Error Interrupt enable

A logic '1' shall enable the interrupt function for a CPU attempting to write a data word into a full transmit buffer. A logic '0' shall mask that interrupt.

Bit 4: Empty interrupt enable

A logic '1' shall enable the interrupt function for the transmit empty flag. The interrupt shall be asserted when the buffer drops down to the empty level from a level higher than empty. A logic '0' shall mask that interrupt.

Bit 3: Almost-empty interrupt enable

A logic '1' shall enable the interrupt function for the transmit almost-empty flag. The interrupt shall be asserted when the buffer drops down to the almost-empty level from a level higher than almost-empty. A logic '0' shall mask that interrupt.

Bit 2: half-full interrupt enable

A logic '1' shall enable the interrupt function for the transmit half-full flag. The interrupt shall be asserted when the buffer drops down to the half-full level from a level higher than the half-full level. A logic '0' shall mask that interrupt

Bit 1: Almost-full interrupt enable

A logic '1' shall enable the interrupt function for the transmit almost-full flag. The interrupt shall be asserted when the buffer drops down to the almost-full level from a level higher than the almost-full level. A logic '0' shall mask that interrupt.

Bit 0: (LSB) full interrupt enable

A logic '1' shall enable the interrupt function for the transmit full flag. The interrupt shall be asserted when the buffer drops off the full level. A logic '0' shall mask that interrupt.

8.2.7 Address "11", CPU READ, Read Received Data

The CPU READ at address "11" shall send to the CPU "DO" interface the data stored into the received buffer in the order the words were received.

8.2.8 Address "11", CPU WRITE, Write Transmit data

The CPU WRITE at address "11" shall write into the transmit buffer data to be transmitted by the UART. It shall be an error if the CPU attempts to write into a full transmit buffer. Under that condition, the error shall be indicated, but no write function shall occur.

8.3 Modes of Operation

Tactically, the UART shall be in one of the following modes.

- **NO Parity:** This mode shall be applied to the receiver and transmit functions of the UART.
- **Even Parity:** This mode shall be applied to the receiver and transmit functions of the UART.

- **Odd Parity:** This mode shall be applied to the receiver and transmit functions of the UART.
- **Forced Reset:** This reset shall represent a CPU command to reset either or both the transmitter and receiver functions to the idle state. Upon a transmit reset, all outgoing messages shall be aborted. Upon a receiver reset all received messages shall be aborted.

9. PERFORMANCE

9.1 Frequency

The UART shall support a maximum baud rate of 25 Mbauds in synchronous mode, and 1.5 Mbauds in asynchronous mode.

9.2 Power Dissipation

The power shall be less than 1 watt at 25 MHz.

9.3 Environmental

Does not apply.

9.4 Technology

The design shall be adaptable to any technology because the design shall be portable and defined in an HDL.

10. TESTABILITY

None required.

11. MECHANICAL

Does not apply.

3 ARCHITECTURAL PLAN

This section defines the architectural plan that represents the implementation approach of the design as defined in the requirement document. This plan serves several purposes:

1. Clarifies how the design will be implemented and what technology and devices will be used.
2. Provides an opportunity, early in the design process, to modify the design approaches. For example, during the review process, other more efficient or economical approaches may be evaluated. These may include reuse or purchase of intellectual property devices, definition of design at a level high enough to allow direct synthesis to gates, or change in hierarchy or structure to allow distribution or sharing of the design.
3. Allows management or senior design engineers to validate the design, and provide design guidelines (i.e., get their wisdom!)
4. Refines design issues that may be ambiguous.
5. Identifies tools, and availability of tools, in the design process.

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

Document #: __
Release Date: ___
Revision Number: ___
Revision Date: _
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 architectural plan starts at 1.0 because it is intended to represent a stand-alone document. Therefore, it does not follow the chapter numbering system.

Concise abstract of the coverage of the specification

1. SCOPE

1.1 Scope

This document defines the architectural plan for the component UART that provides a bridge between a microprocessor interface and a transmit/receive asynchronous or synchronous, parameterized eight to thirty-two bit serial interface, emulating a UART-like protocol.

Target audience

This plan is primarily targeted for designers.

Purpose of specification

1.2 Purpose

This plan will help assess the validity of the design and explore alternatives.

System, hardware, software

1.3 Classification

This document defines the requirements for a hardware design.

2.0 DEFINITIONS

See section 3.2 for referenced document.

3. APPLICABLE DOCUMENTS

3.1 Government documents

3.1.1 TIA/EIA-232-F

Sources of applicable documents

Interface between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange (ANSI/TIA/EIA-232-F-1997), October 14, 1997.

Standard and government specifications and requirements

http://www.tiaonline.org/standards/search_results2.cfm?document_no=TIA/EIA-232-F

Electronic Industries Alliance, 2500 Wilson Boulevard
Arlington, VA 22201-3834, (703) 907-7500

3.2 Non-government documents

Requirements for an asynchronous or synchronous 8 TO 32 BIT Universal Asynchronous Receiver/Transmitter,
Document #: 01

3.3 Executable specifications

None.

4. ARCHITECTURAL OVERVIEW

The UART model will consist of four partitions or subblocks that represent the separate functional elements of the design, as shown in Figure 4.0.

1. **CPU Subblock**. This interface will provide the data transfer between the UART and the CPU that controls it. It will also include the configuration registers for the modem control *(DTRn)*, and the receive and transmit interrupt masks.
2. **Receiver Subblock.** This interface will include the receiver de-serialization logic and the receive buffering of the received data over the serial *RxD* interface.
3. **Transmitter Subblock.** This interface will include the transmitter serialization logic and the transmit buffering of the data to be transmitted, as commanded by the CPU.
4. **Clock Subblock.** This partition will provide the baud clocks, and bit synchronization for the transmit and receive serial data.

4.1 CPU Subblock

The CPU interface will consist of the following logic:

1. Address decoding logic for the control of the UART.
2. Configuration registers for the storage of configuration information of the design, including:
    a. Data set ready
    b. Interrupt masks for the transmit function
    c. Interrupt masks for the receive function
3. Interrupt control logic for the generation of interrupts based on the status of the transmit and receive buffers, and the interrupt masks.

4.2 Receiver Subblock

The receiver subblock will de-serialize the incoming data on the *Rxd* port, and will store it into a FIFO buffer. This logic will alert the CPU interface of the status of the FIFO, and the receive errors detected for each received word. These errors will include:

1. Parity error, when parity is enabled
2. Overrun error, when the receive FIFO is full and a new word is received.
3. Framing error when the received data is not properly framed. The unframed data, as expected from the state machine, will be stored in the buffer.

The FIFO design will be custom designed to the requirements, rather than the use of an intellectual property (IP) design because of limited financial resources, and the desire to freely distribute the code for this book. The FIFO design will be reused in the transmit subblock.

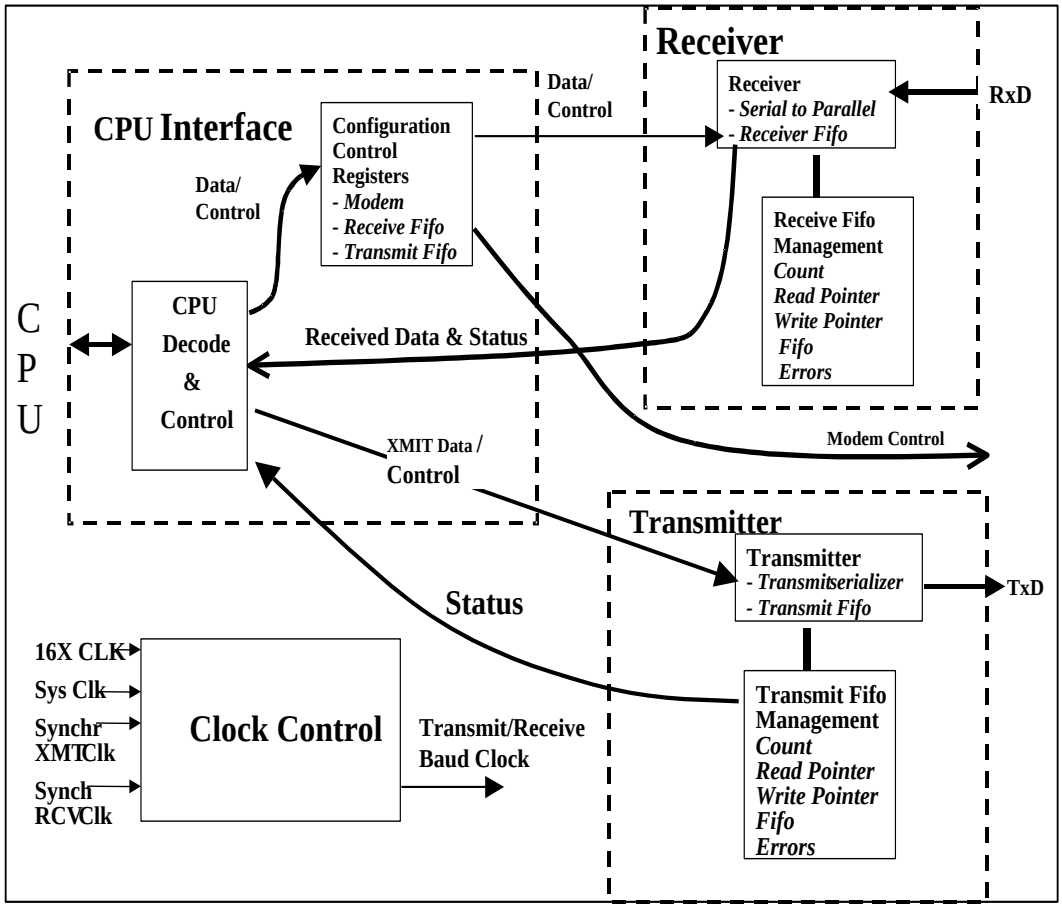


Figure 4.0 High Level View of UART Architecture

4.3 Transmit Subblock

The transmit subblock will serialize the CPU data stored into FIFO buffer onto the *Txd* port, as per the protocol described in the requirement specification. This logic will alert the CPU interface of the status of the FIFO, and the FIFO overrun error. An overrun error will occur when the CPU attempts to store data into a full FIFO.

The FIFO design will use the design defined for the receive subblock.

4.4 Clock Subblock

This partition will provide the baud clocks, and bit synchronization for the transmit and receive serial data.

Interfaces as seen from the pinout viewpoint

5. PHYSICAL LAYER

The physical hardware interfaces will be as specified in the requirement specification. Figure 5.1 reiterates the interfaces. Refer to section 5.0 of the requirement specification for the description of the interface ports.

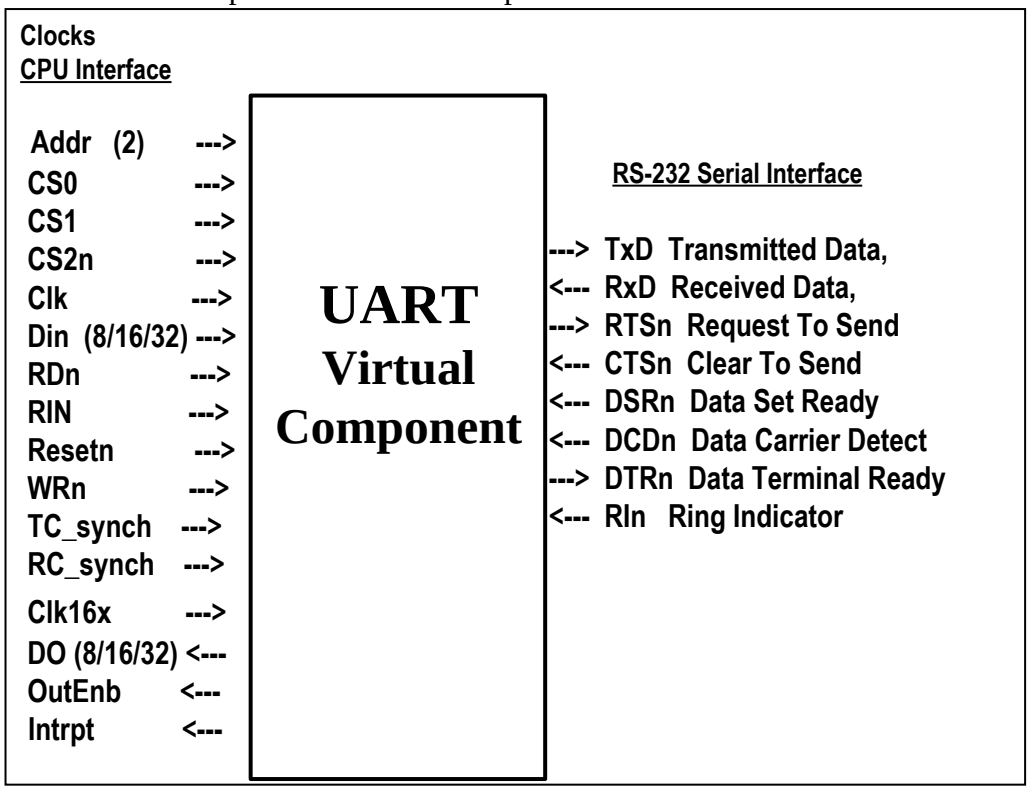


Figure 5.1 Interfaces of the UART

The design will be built with the FPGA device defined in Table 5.1 because this component appears to meet the speed and power requirements, and it is currently available in stock. The VHDL code is however portable, and is adaptable to any technology supported by the compiler and the layout tools.

Table 5.1 Selected FPGA Component

FPGA CHARACTERISTIC	VALUE
Technology	**Altera FLEX10K**
Part	**Altera EPF10K10**
Package	**LC84**
Speed grade	**3**

6. PROTOCOL LAYER

The transmit and receive logic will abide by the rules of the protocol, as defined in the requirement specification.

7. ROBUSTNESS

The error detection types, as defined in the requirement specification, will be observed.

8. HARDWARE AND SOFTWARE

8.1 Fixed Parameterization

Parameterization of the word size, buffer depth, and buffer threshold (e.g., almost-empty) will be specified with generics.

8.2 Software Interfaces

The address decoding and control fields, as defined in the requirement specification, will be observed.

9. PERFORMANCE

Design rules that enable higher operating frequency will be followed. These rules are defined in the RMM[1]. They include rules such as sourcing outputs off subblocks through registers. Additional constraints will be imposed on the synthesizer and layout tools to achieve the desired performance. The synthesis and layout tools for the targeted device will verify frequency performance.

10. TESTABILITY

No testability hardware is required.

---

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

11. DESIGN TOOLS

Table 11.1 summarizes the design tools to be used in this project.

Table 11.1 Tools Used in Design Process

TOOL	VENDOR	FUNCTION
ModelSim EE 5.4b with Code Coverage	Model Technology, Mentor Graphics	VHDL/Verilog co-simulator with code coverage
Synplify	Synplicity 5.3.1 with HDL Analyst	VHDL and Verilog synthesis for FPGAs and CPLDs. Produces EDIF files, performance and timing reports, RTL and gate level views, and Extraction views of desired paths.
Max+plusII version 9.4	Altera	Layout tool for Altera devices
Renoir V99.6	Mentor Graphics	Design environment
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. Other tools, not included here include debuggers, lint, memory makers, behavioral compilers, ASIC layout, timing analyzers, testability insertion, power estimator, etc.

An example of a degugging tool is Novas' Debussy[2](r) Total Debug (tm) system for complex designs at the gate, RTL and behavioral levels. Debussy compiles the original HDL source files and uses pre-synthesis techniques to extract knowledge of the design such as flip-flops, latches, and multiplexers. It uses this knowledge to intelligently generate schematic diagrams that isolate any portion of the RTL or netlist, such as a single logic cone, a critical path, or the interconnection between particular blocks, and to analyze the reasons for problems such as unknown (X) values. Open interfaces allow integration with a wide range of simulators as well as formal verification and timing analysis tools.

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

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

A vector set is a group of 1's and 0's that contain input stimulus to the input pins

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

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 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

---

5 http://www.synopsys.com/

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

7 http://www.chronology.com/

8 http://www.testbuilder.net

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.

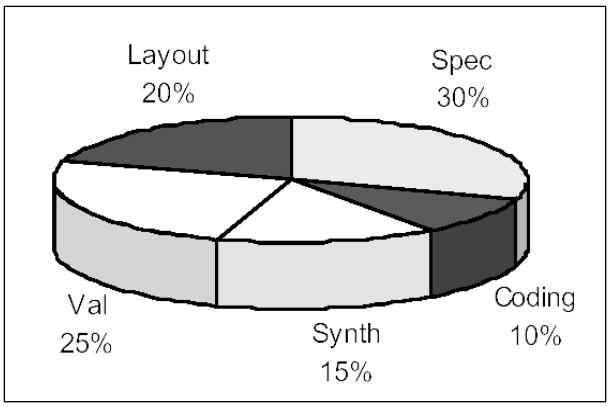


Figure 4.1.2 Breakdown of Engineering Effort[9]

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;*
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;*

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/

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

Concise abstract of the coverage of the testplan

1.1 Scope

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 DO(7..0) = 00000000 DO(7..0) = 00000000

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	8.2.6 8.2.8 6.0, 5.1.1	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	 - No serial output - No serial output - No serial output

	. Modify modem interface until Enable of transmission mode. . Check for interrupt. . Read transmit PIR	5.1.2.11 6.0 8.2.5		- 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	- 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 &	SPEC #	Prio	TEST SEQUENCE	VERIFICATION CRITERIA

	DIRECTED TEST STRATEGY		rit y		
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. . 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 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 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.

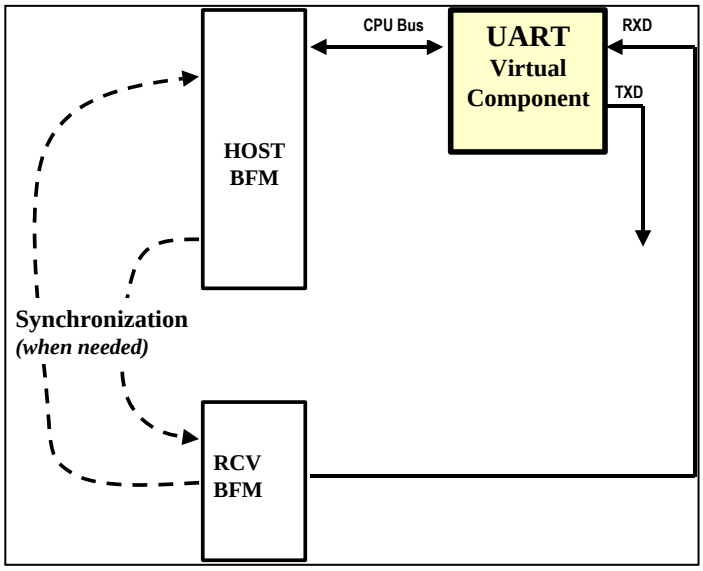


Figure 4.2.2 BFMs for UART Model

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.

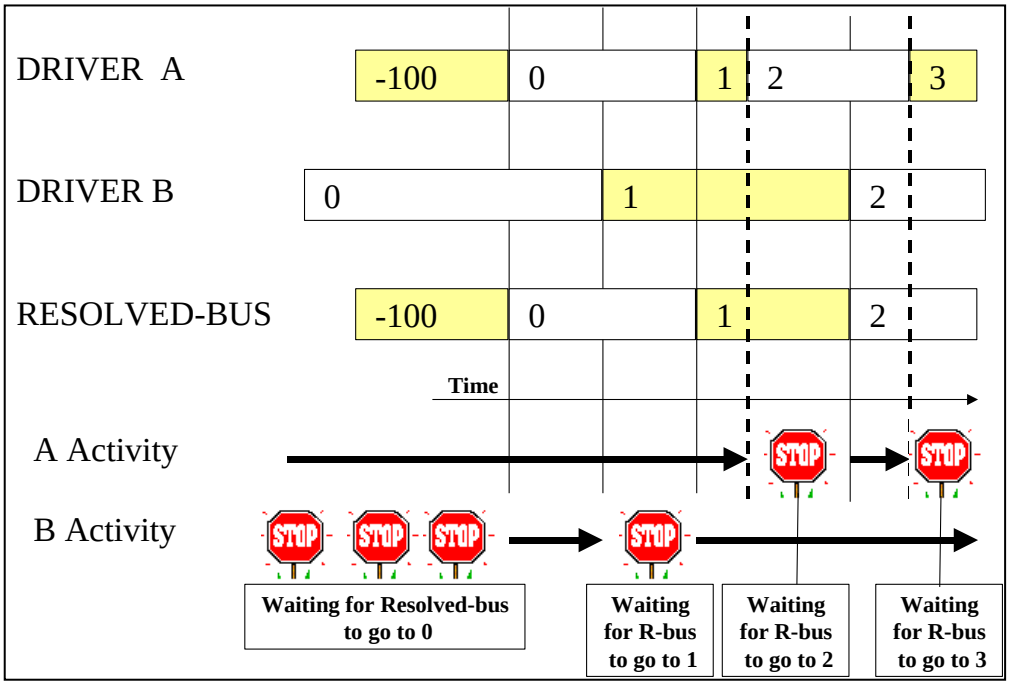


Figure 4.2.3 Application of Resolved Integer Signal where Low Value Wins

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.

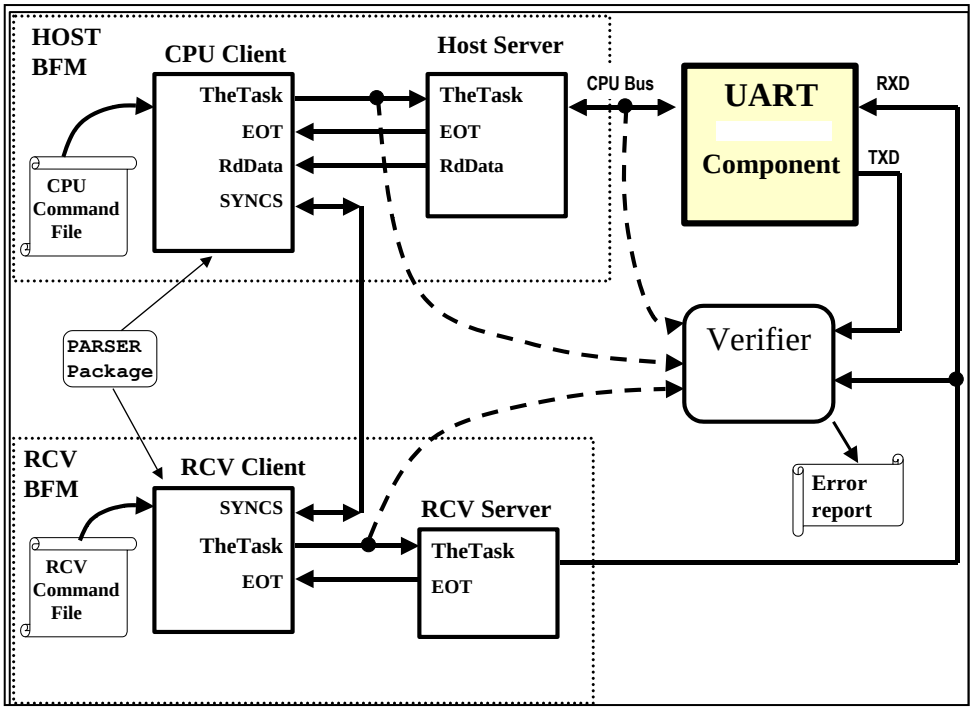


Figure 4.2.7-1 Testbench Architecture Overview using BFMs and Automatic Verification

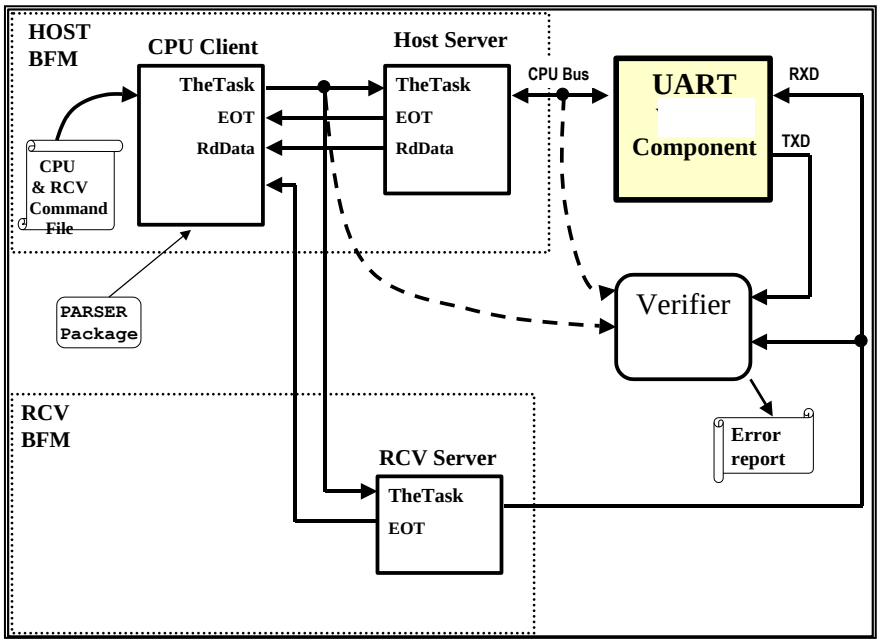


Figure 4.2.7-2 Testbench Architecture Potential using Single-Command File (Not used)

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.

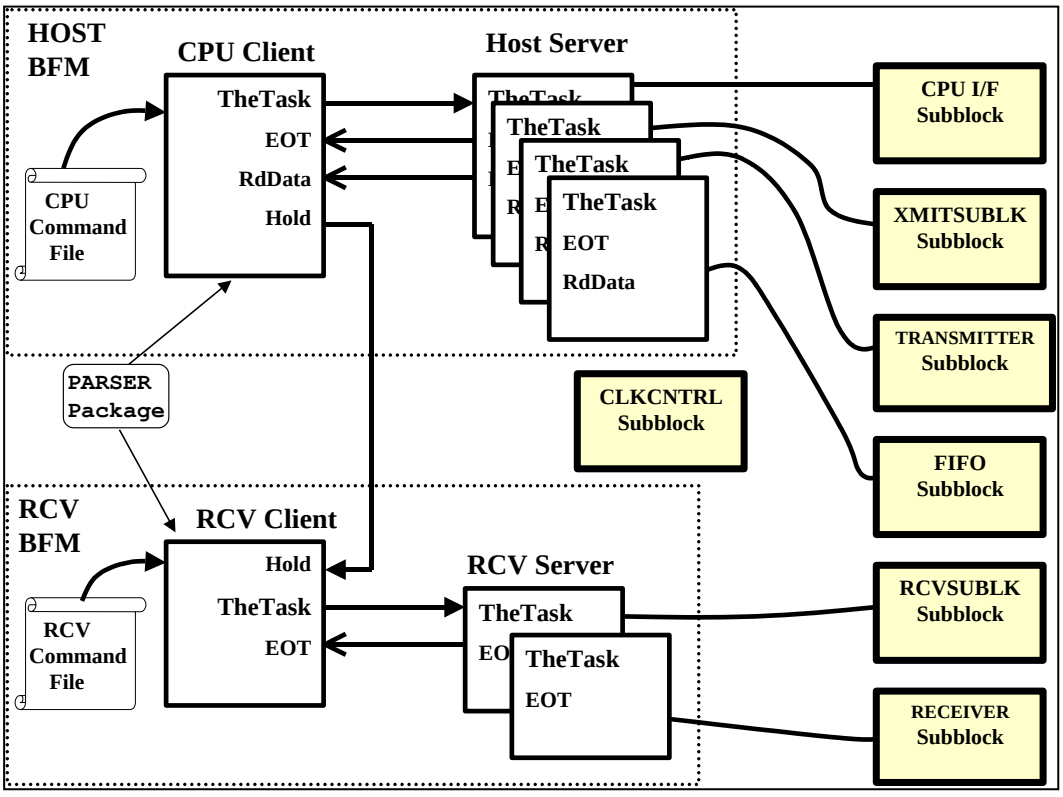


Figure 4.2.8 Subblock Verification Overview using BFMs and Visual Verification

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 Buffer Status**	**2, 11, 12, 13, 14, 15**

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/

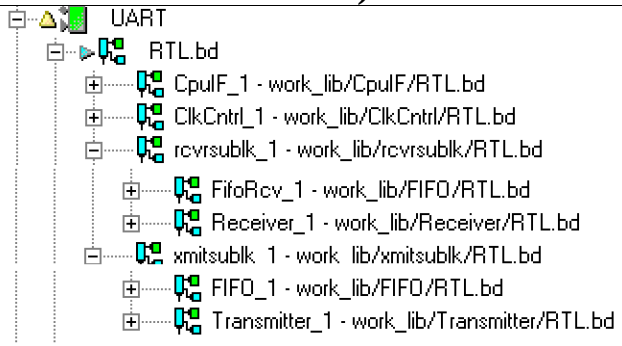
5 DESIGN AND SYNTHESIS

This chapter describes the UART RTL model, the VHDL code edited with *emacs* and *vhdl-mode*, and the synthesis and FPGA layout process using Synplicity's *Synplify 5.3.1* along with Altera's *MAX+PLUS® II ver. 9.4.* The design was also compiled with *ModelSim EE 5.4b* with Code Coverage. Mentor Graphics' *Renoir v99.6* was also used for the documentation of the design.

5.1 RTL DESIGN

Per the architectural plan, the UART consists of the following major subblocks as shown in Figure 5.1 (page 76). The UART makes use of two buffers, or FIFOs, to store the data sent and the data received. Since this provides an opportunity for reuse, a FIFO subblock is defined, and reused in the transmitter subblock and the receiver subblock. The design Hierarchy is shown is Table 5.1. The function of each level of the design is further described in the following subsections.

Table 5.1 UART Design Hierarchy

Uart Hierarchy (*from Renoir's Design Browser*)	**FILES**
	**Uart.vhd** • **cpuif.vhd** • **clkcntrl.vhd** • **rcvsublk.vhd** ○ **fifo.vhd** ○ **receiver.vhd** • **xmitsublk.vhd** ○ **fifo.vhd** ○ **transmitter.vhd**

5.1.1 CPU Interface (*CpuIf*) Subblock Design

Figure 5.1.1-1 represents a closer view of the *CpuIF* subblock component. This subblock provides several services:

1. Setups the *Data Terminal Ready (DTRn)* signal to the modem, as commanded by the CPU.
2. Stores and transfers to other partitions the configuration definitions for the type of data transfer for the serial transmission, and for the control of the UART. This includes:
    a. Parity enable (*EnbParity_r*)
    b. Parity bit *(ParityBit_r)*
    c. Transmit interrupt masks for enables or masks of interrupts for the transmit hardware (e.g., off-full, empty).
    d. Receive interrupt masks for enables or masks of interrupts for the receive hardware (e.g., full, not-empty).
    e. Reset of the transmit logic from a hard or soft (i.e., CPU initiated) reset. This signal is called *XmtForcedResetn.*
    f. Reset of the receive logic from a hard or soft (i.e., CPU initiated) reset. This signal is called *RcvForcedResetn.*
3. Loads CPU data *(TxData_r)* to be serialized into the transmit subblock (*xmtsulk*) FIFO. This control signal is called *LdXmtFifo_r* and represents a *push* into the transmit FIFO.
4. Transfers received data from the receive subblock (*rcvsublk)* FIFO to the CPU data bus. This control signal is called *RdRxData,* and represents a *pop* of the receive FIFO.
5. Provides interrupts to the CPU.

6. Provides tri-state control of CPU data. This separate control is used because the component may be applied into another level of hierarchy that may have its own output buffer or drivers.

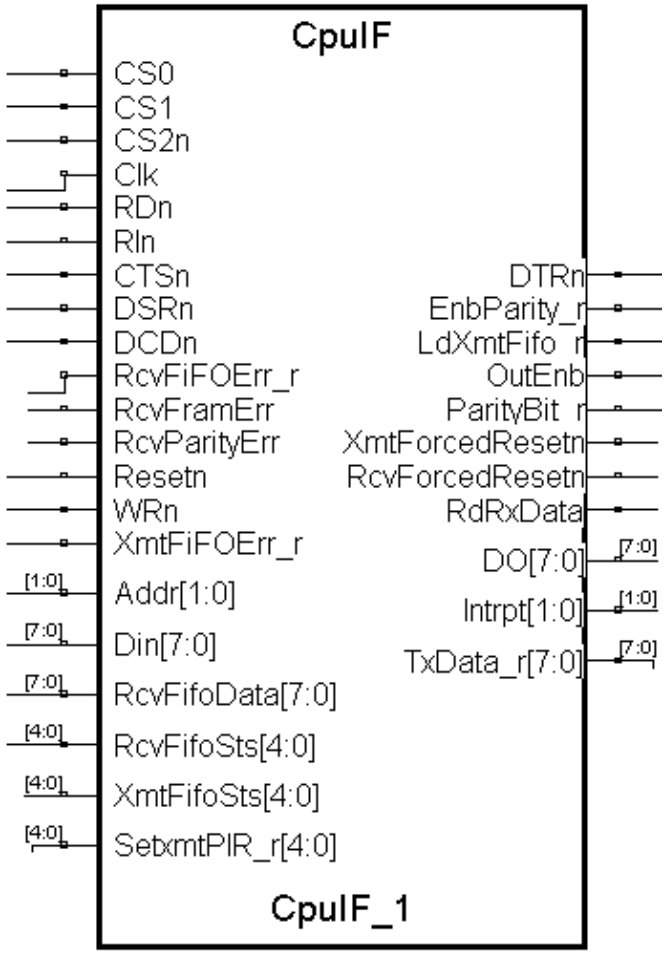


Figure 5.1.1-1 CpuIF Subblock (from Synplicity's *Synplify*)

The *CpuIF* subblock consists of five blocks with seven processes and one set of concurrent signal assignments, as summarized in Figure 5.1.1-2 and Table 5.1.1 on pages 77 and 78. Figures 5.1.1-1 (page 77) represents the registers sourced from the CPU data bus. A graphical view of the process blocks within the *CpuIf* are demonstrated in Figure 5.11-3x (pages 79 through 81).

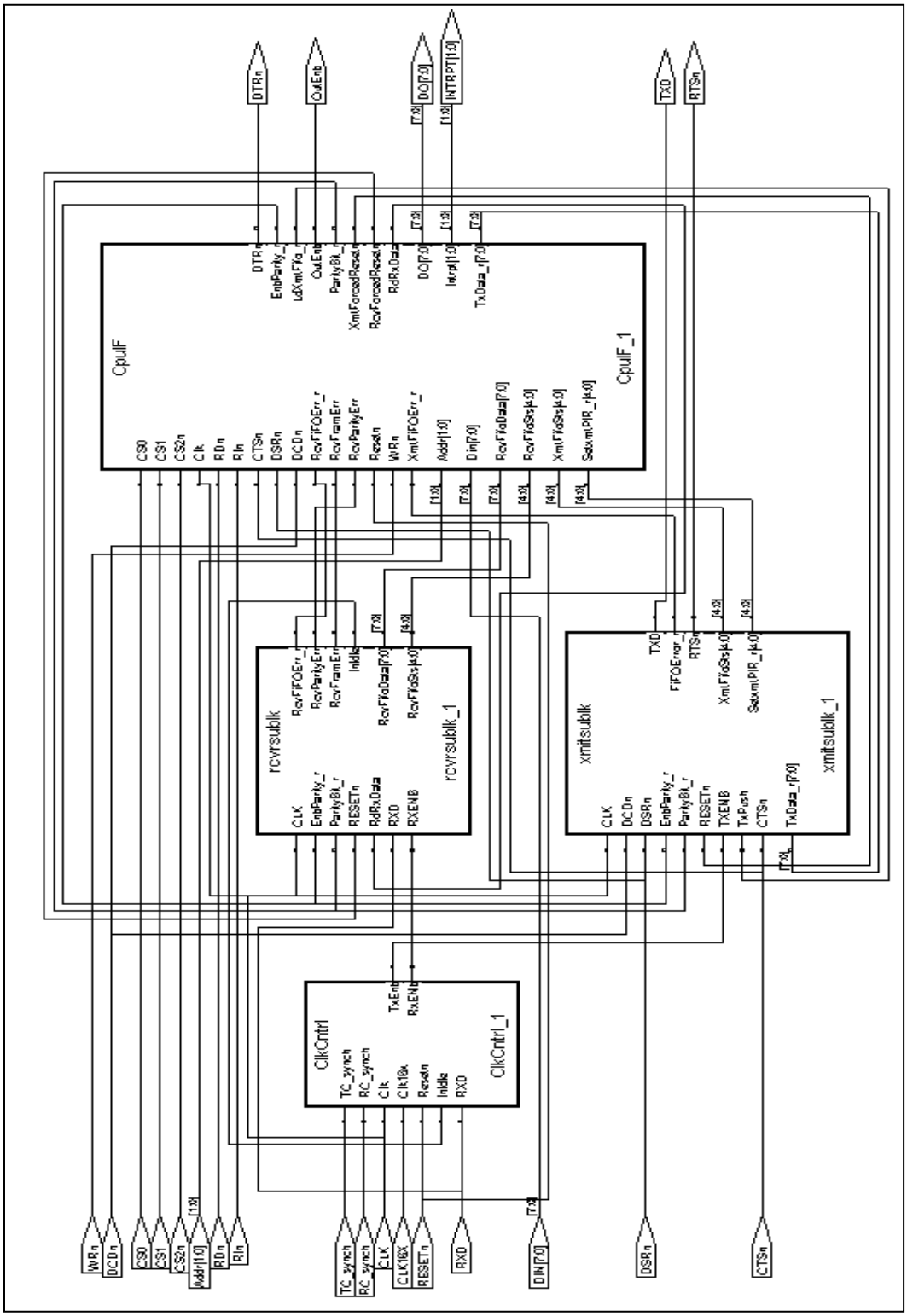


Figure 5.1 UART Subblocks (from Synplicity's *Synplify*)

Table 5.1.1 CpuIF Processes

CpuIF Processes *(from Renoir's Design Browser)*	**Function**
CpuIF_1 - work_lib/CpuIF/RTL.bd 8 - RcvPIR_Proc 7 - StatusREg1_Proc 6 - XmtPIR_Proc 5 - StatusREg_Proc 4 - Delayreg_Proc 3 - ControlReg_Proc 2 - RcvFifo_Proc 1 - eb1	Receive Pending Interrupt Register Receive status register for generation of PIR Transmit Pending Interrupt Register Transmit status register for generation of PIR Delayed version of the forced reset Control data from CPU UART data to CPU Chip Enable decode

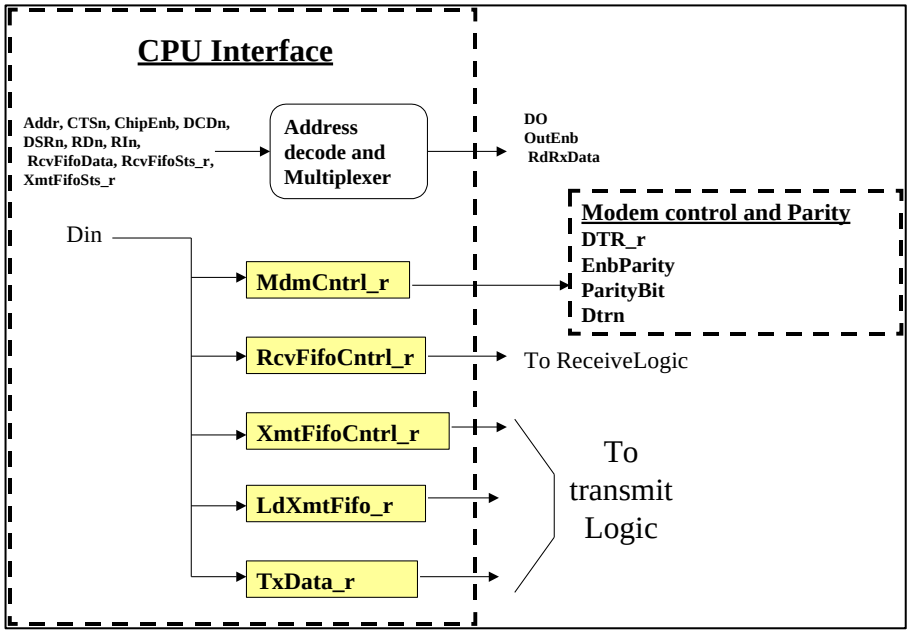
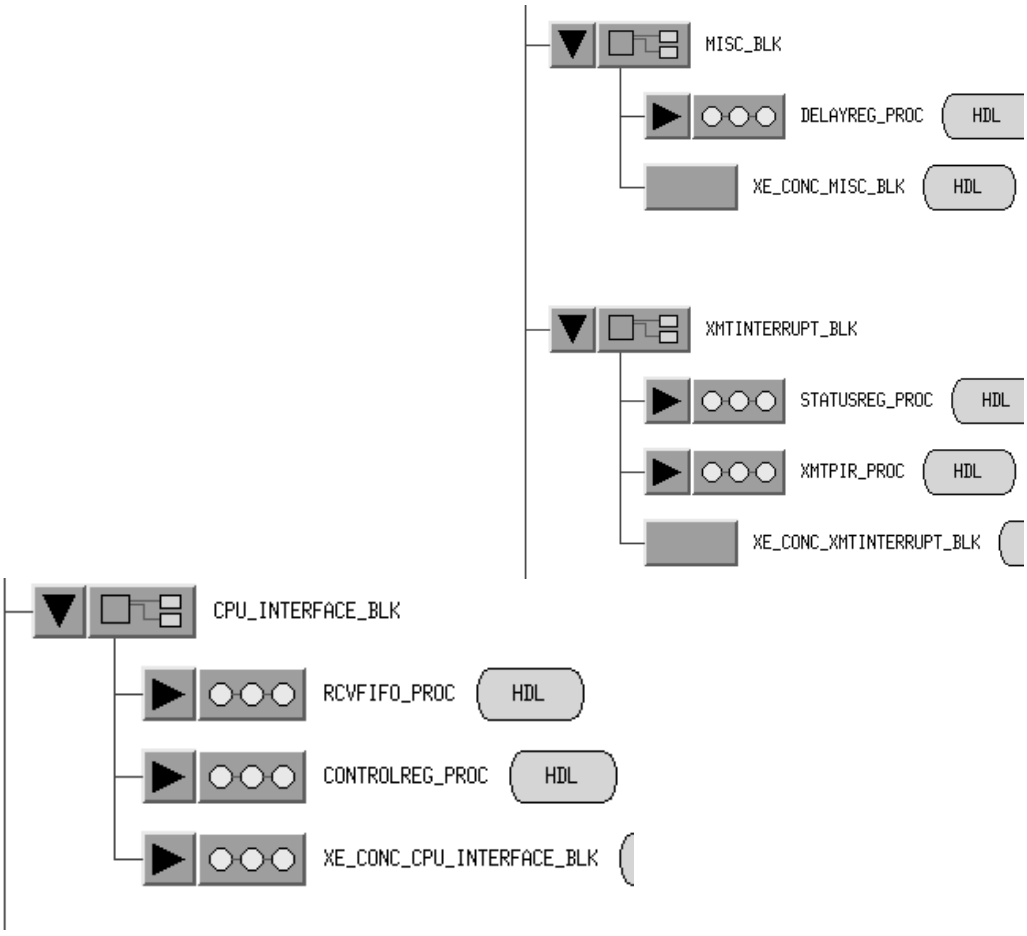


Figure 5.1.1-1 High Level Functional View of the CPU Bus Interface for Reception of Data

Figure 5.1.1-4 (page 82) represents a high level view of the logic generating the transmit and receive interrupts. Data from the receive FIFO status (e.g., empty, almost empty, full) is stored into a register, and edge detected for storage into the pending interrupt register. The PIR and the interrupt masked are then processed to generate the receive interrupt. The Transmit subblock provides the data to set the transmit PIR.


MISC_BLK
DELAYREG_PROC
HDL
XE_CONC_MISC_BLK
HDL
XMTINTERRUPT_BLK
STATUSREG_PROC
HDL
XMTPIR_PROC
HDL
XE_CONC_XMTINTERRUPT_BLK
CPU_INTERFACE_BLK
RCVFIFO_PROC
HDL
CONTROLREG_PROC
HDL
XE_CONC_CPU_INTERFACE_BLK

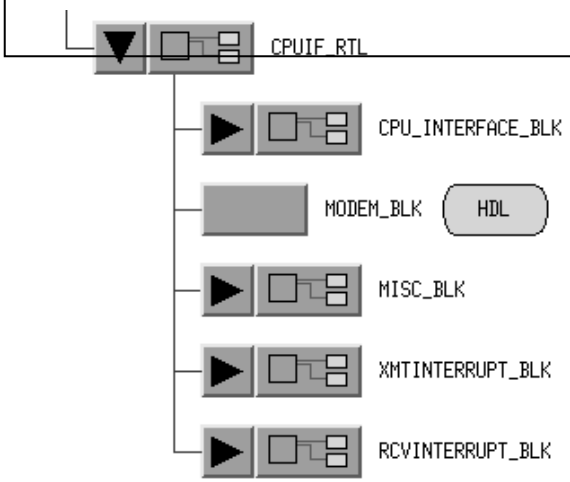
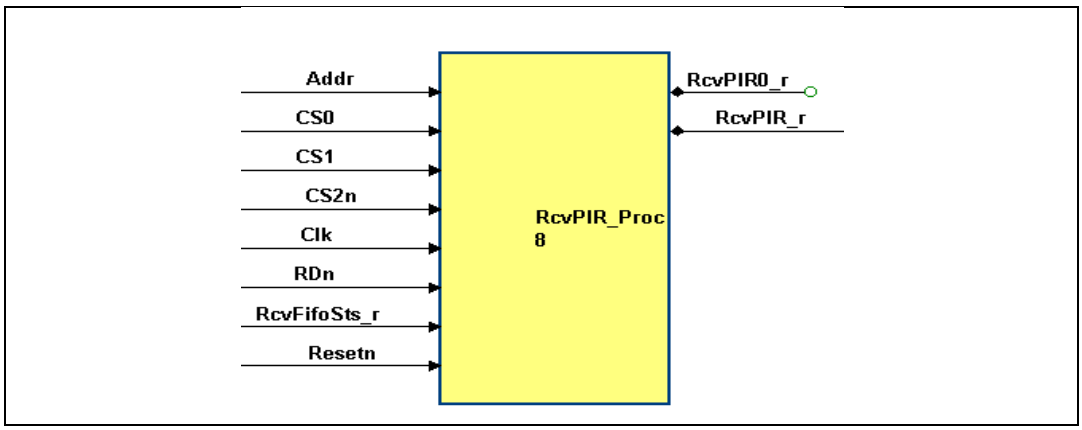


Figure 5.1.1-2 CPU Interface Blocks and Processes
(generated with YxI XE tools)



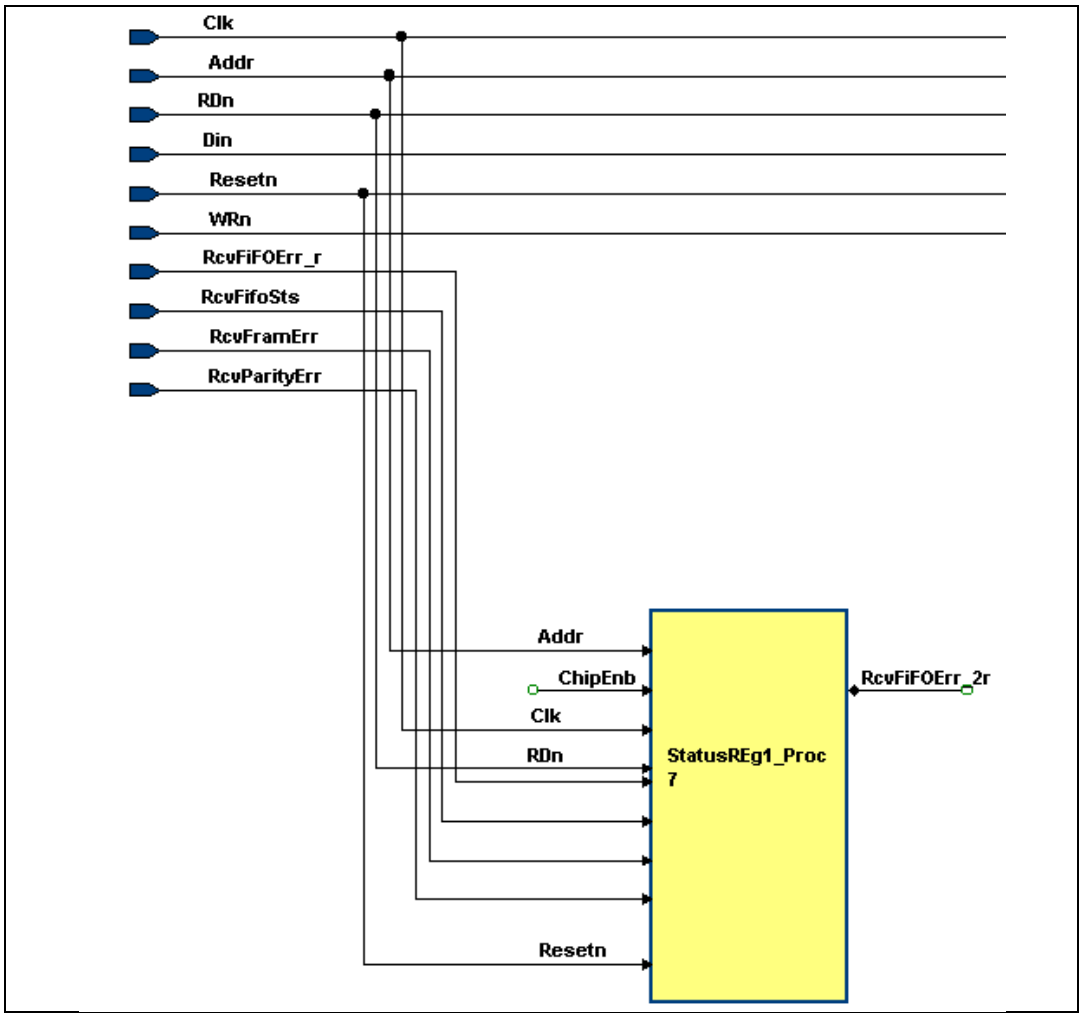


Figure 5.1.1-3a Graphical View of CpuIF processes (*from Renoir's BD view)*

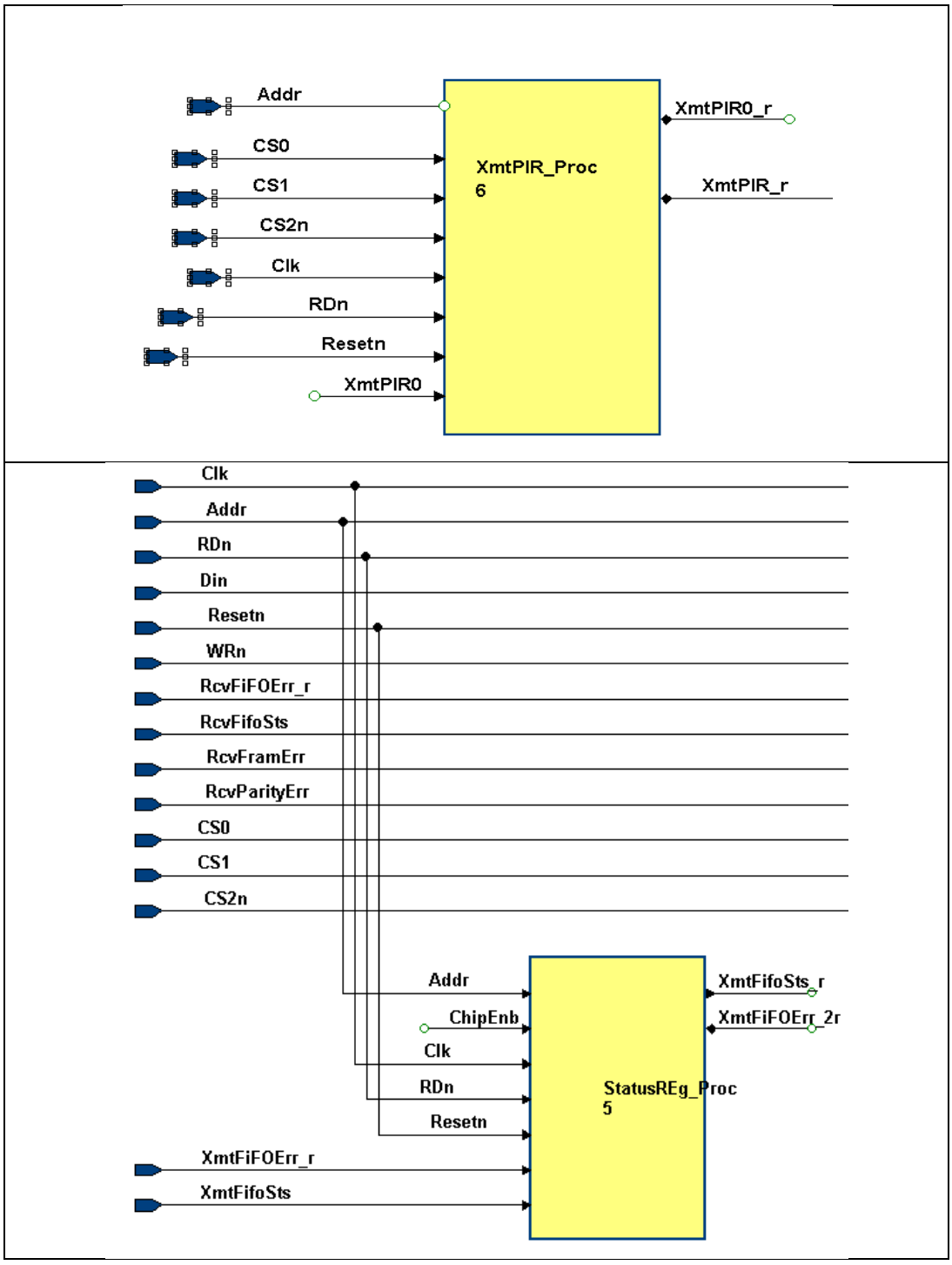


Figure 5.1.1-3b Graphical View of CpuIF processes (*from Renoir's BD view)*

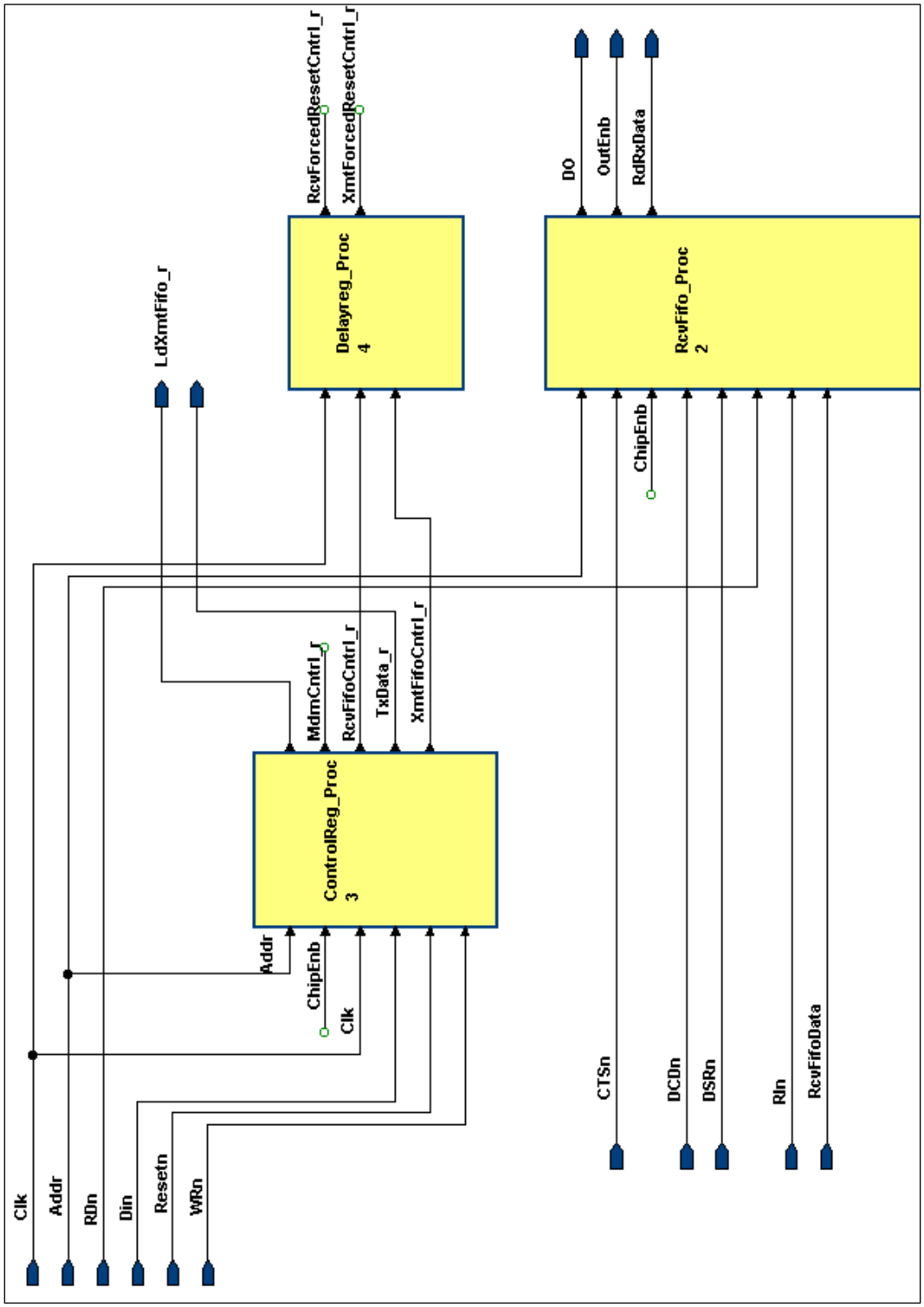


Figure 5.1.1-3c Graphical View of CpuIF processes (*from Renoir's BD view)*

TRANSMIT INTERRUPT	RECEIVE INTERRUPT

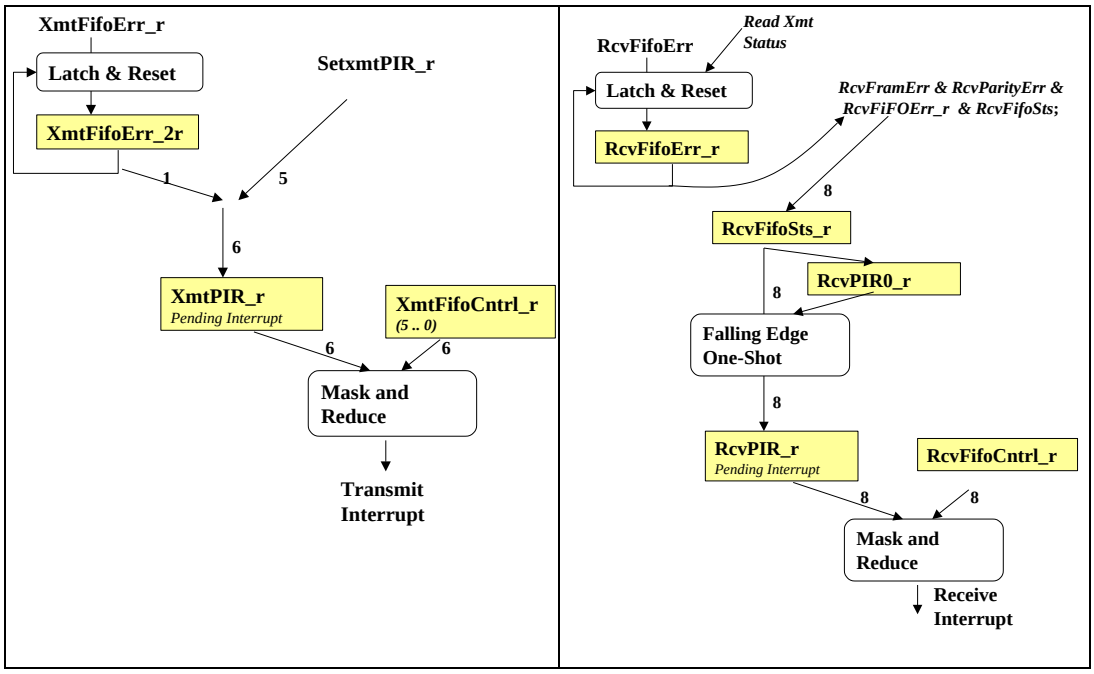


Figure 5.1.1-4 High Level View of Logic Generating the Interrupts

The CpuIF VHDL code (on CD in file vhdl/cpuif.vhd) is provided below on pages 83 through 89.

Insert 7 pages of cpuif.pdf
P1 cpuif.pdf

P2 cpuif.pdf

P3 cpuif.pdf

P4 cpuif.pdf

P5 P1 cpuif.pdf

P6
P1 cpuif.pdf

P7 P1 cpuif.pdf

5.1.2 Clock Control

The clock control logic serves several purposes:

1. For asynchronous transmission, it synchronizes the sixteen times clock (*Clk16X*) to the system clock (*Clk).*
2. For asynchronous transmission, it generates the Transmit-Clock-Enable (*TxEnb*) that represents a single pulse every sixteen synchronized sixteen times clock.
3. For asynchronous transmission, it generates the Receive-Clock-Enable *(RxEnb)* that represents a single pulse every sixteen synchronized sixteen times clock, synchronized to the START cycle (from RxD) when the receiver is in the IDLE state.
4. For synchronous transmission, it transfers the Transmit-Clock *(TC_Synch)* and Receive-Clock *(RC_Synch)* to the *TxEnb* and *RxEnb* ports.
5. It reclocks the *RxD* signal to the system clock to reduce meta-stability effects.

Figure 5.1.2-1 represents the interfaces of the Clock Control subblock. Figure 5.1.2-2 demonstrates the reclocking registers of the sixteen times clock and the *RxD* signal. Figure 5.1.2-3 demonstrates the timing of the clock control logic in asynchronous mode. The VHDL code for the clock control is file *clkcntrl*.vhd, and is enclosed in page 92.

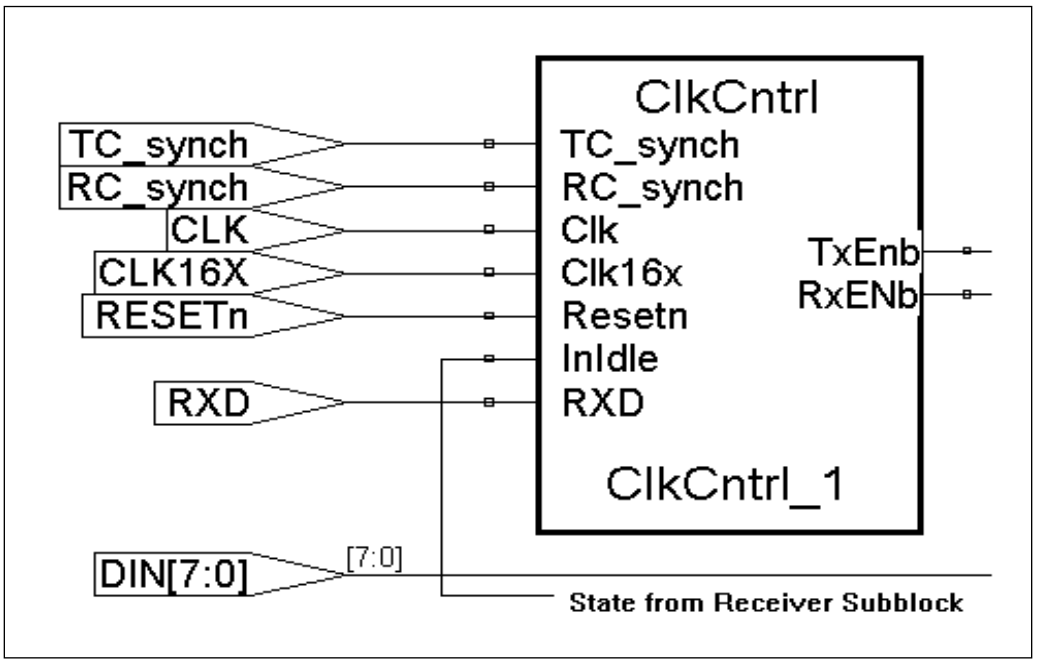


Figure 5.1.2-1 Clock Control Subblock (from Synplicity's *Synplify*)

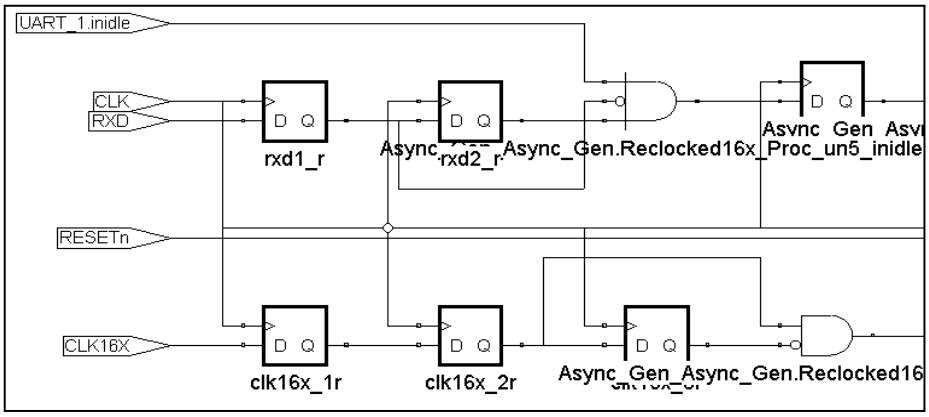


Figure 5.1.2-2 Reclocking Registers of the Sixteen Times Clock and the *RxD* signal (from Synplicity's *Synplify*)

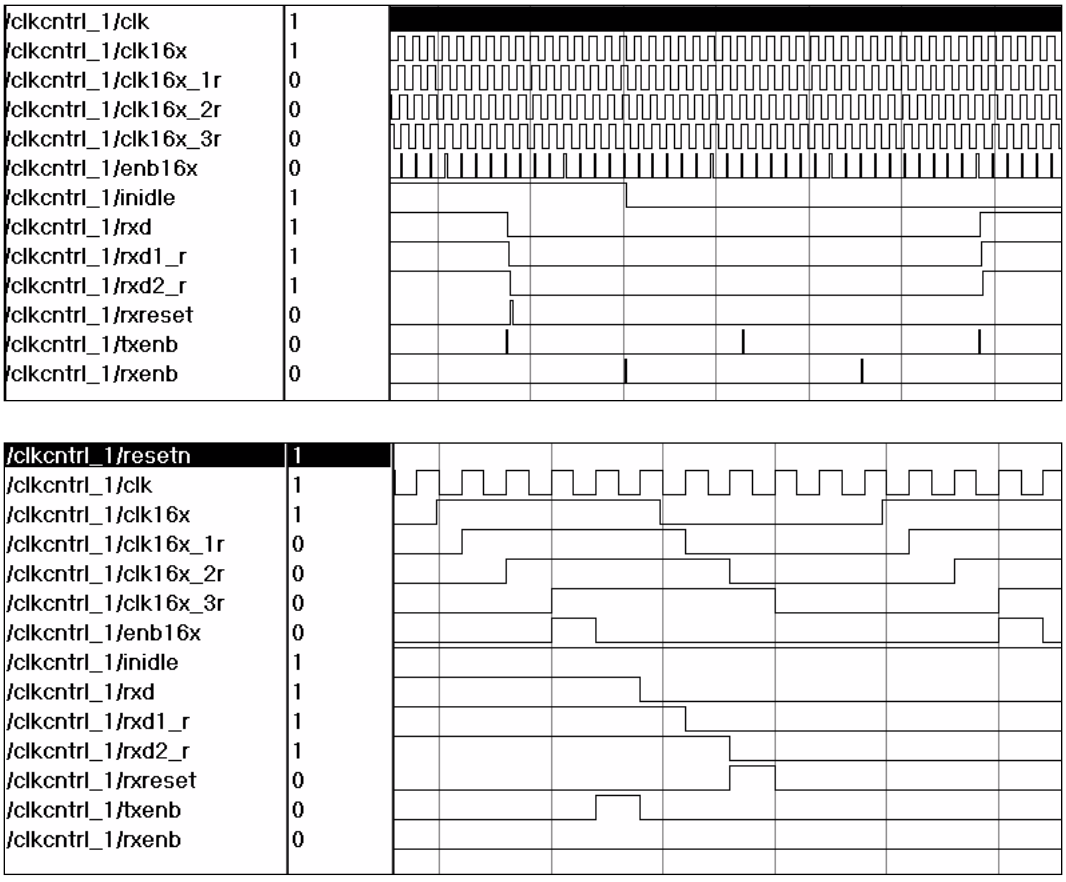


Figure 5.1.2-3 Clock Control Timing (generated with *ModelSim 5.4b*)
Note the Reclocking of the Clk16x and RxD, and the generation of the TxEnb and RxEnb in Asynchronous Mode.

Clkcntrl 1/2
Code

ClkCntrl code
Page 2/2

5.1.3 Receiver Subblock *(rcvsublk)*

The receiver subblock *(rcvsublk)* is responsible for the following tasks:

1. De-serializes of serial data *(RxD)* to a word
2. Stores the received word into an internal FIFO
3. Performs error checks
    i. Parity error
    ii. Overflow error
    iii. Framing error
4. Transfers received data to the CPU interface
5. Maintains FIFO pointers for WRITES and READS

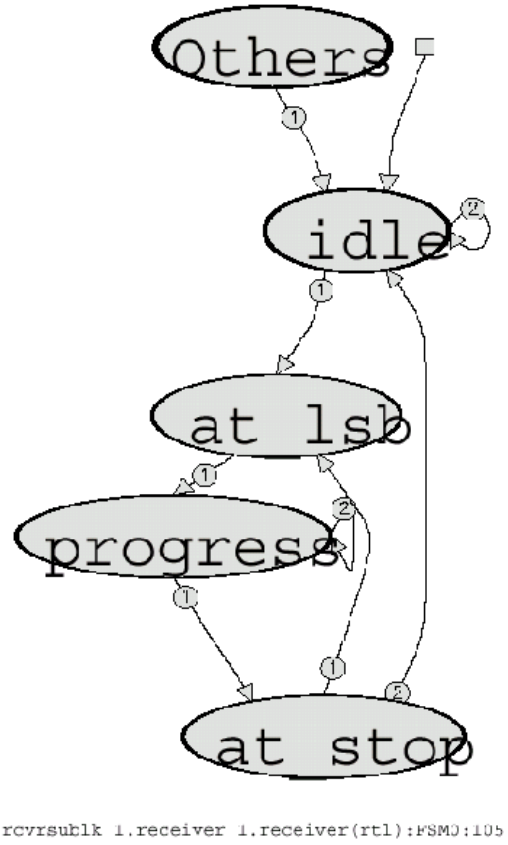


Figure 5.1.3-1 Receiver FSM (automatically generated from RTL by Novas' *Debussy* debugging tools)

The receiver subblock consists of two other hierarchical subblocks as shown in Figure 5.1.3-2.

1. Receiver component
2. FIFO component

Figure 5.1.3-1 represents a view of the receiver finite state machine (FSM).

The VHDL code for the *rcvsublk*, *receiver*, and *FIFO* are in files *rtl/rcvsublk.vhd*, *rtl/receiver.vhd*, and *fifo.vhd*. The code for these designs is shown on pages 97, 100, 103.

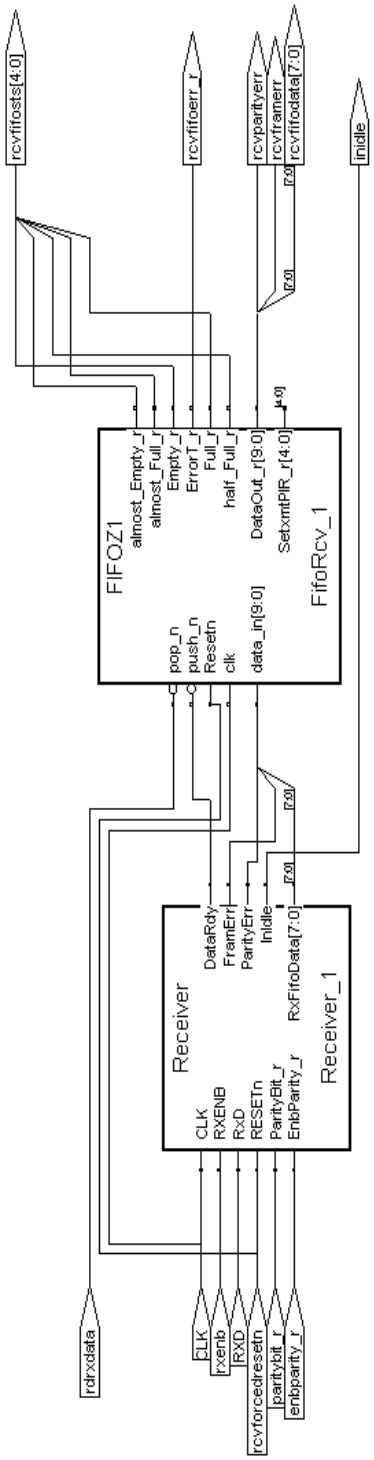


Figure 5.1.3-2 Receiver Subblock Hierarchy (from Synplicity's *Synplify*)

The receiver component is responsible for the following tasks:

1. It de-serializes serial data *(RxD)* to a word for transmission to the FIFO
2. It transfers serial data state to the control logic *(InIdle* state).
3. It performs error checks for storage into the FIFO
   i. Parity error
   ii. Overflow error
   iii. Framing error
4. It provides the PUSH and POP Controls for FIFO control.

Figure 5.1.3-3 is view of the processes within the Receiver component that consists of three major processes:

1. *Rcv_Proc* for de-serialization process
2. *Counter_Proc* for the count of received bauds, based on parity and number of bits per word.
3. *Frame_Proc* for the detection of framing error, based on parity and number of bits per word.

Figure 5.1.3-4 is a view of the processes within the FIFO component. The FIFO component is responsible for the following tasks:

1. It Stores de-serialized data and error status information into the FIFO
2. It supplies stored data to CPU
3. It maintains FIFO pointers for WRITES and READS

The FIFO component is reused in the transmit logic. However, the logic requires a different pipeline delay for the FIFO output between the FIFO used in the receiver logic versus the FIFO used in the transmit logic. This difference in behavior is handled within the code with the "generate" feature of VHDL.

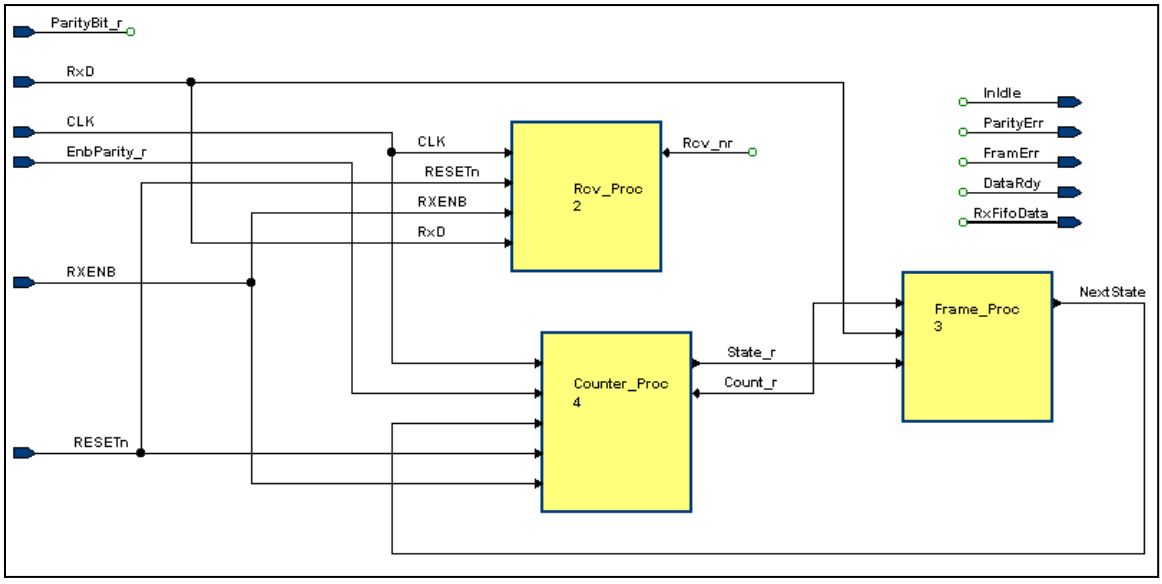


Figure 5.1.3-3 Receiver Component processes (*from Renoir's BD view)*

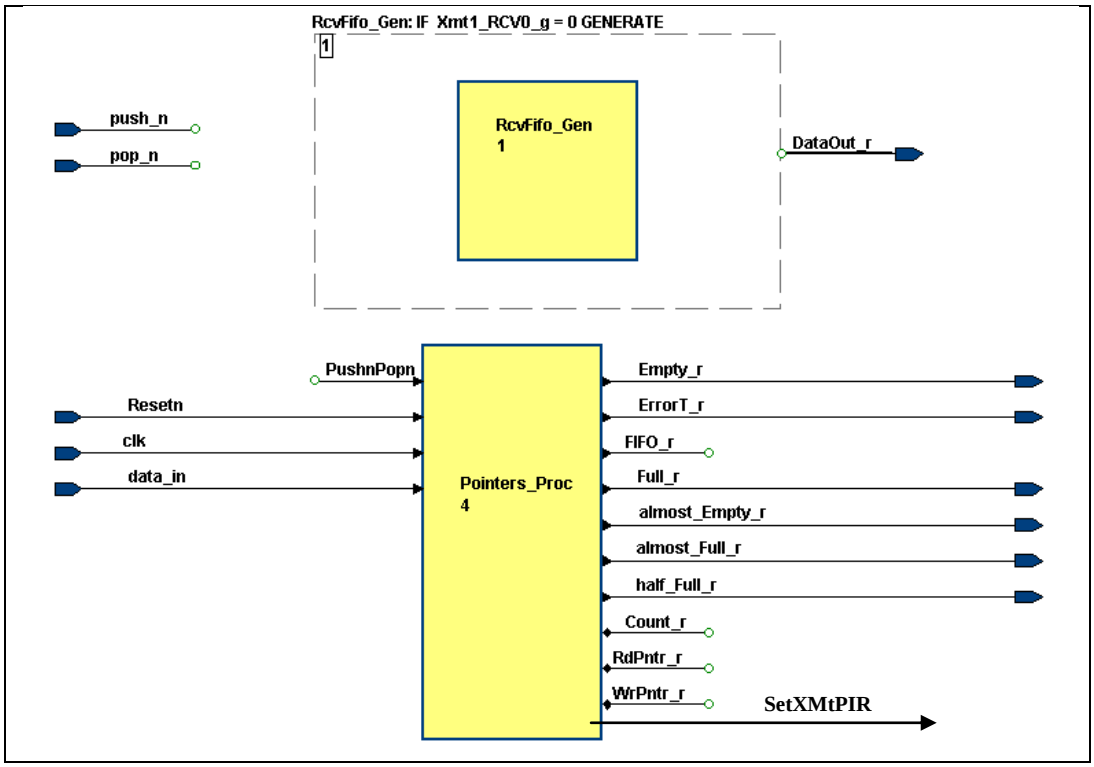


Figure 5.1.3-4 FIFO Component processes (*from Renoir's BD view)*

Rcvsublk
1/3

Rcvsublk
2/3

Rcvsublk
3/3

Receiver 1/3

Receiver 2/3

Receiver 3/3

FIFO 1/3

FIFO 2/3

FIFO 3/3

5.1.4 Transmit Subblock *(xmitsublk)*

The transmit subblock *(xmitsublk)* is responsible for the following tasks:

1. It stores CPU transmit data into a local FIFO
2. It extracts data from the transmit FIFO
3. It serializes data to be transmitted to *TxD* port per required format
4. It performs overflow error check (CPU write to a full FIFO)
5. It maintains FIFO pointers for WRITES and READS

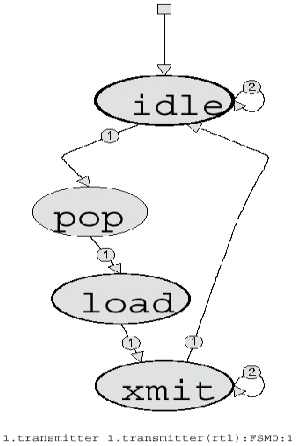


Figure 5.1.4-1 Transmitter FSM (automatically generated from RTL by Novas' *Debussy* debugging tools)

The transmit subblock consists of two other hierarchical subblocks as shown in Figure 5.1.4-3.

1. Transmitter component
2. FIFO component

Figure 5.1.4-1 represents a view of the transmitter finite state machine (FSM).

The VHDL code for the *xmtsublk and transmitter* are in files *rtl/xmtsublk.vhd* and *rtl/transmiter.vhd.* The code for these designs is shown on pages 109, 112.

Figure 5.1.4-2 demostrates the transmit path initiated from the FIFO to the transmit output *TxD.*

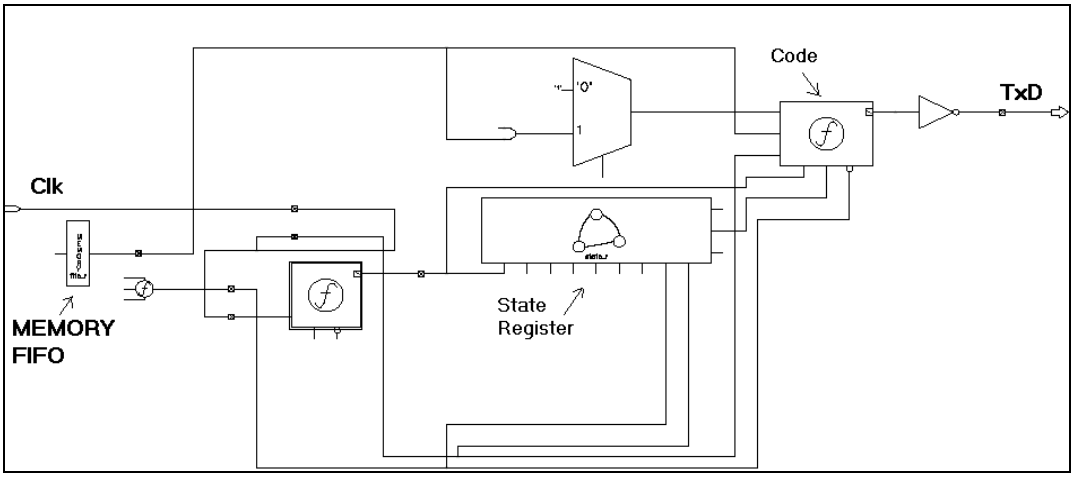


Figure 5.1.4-2 Transmit Path initiated from the FIFO to the transmit output *TxD* (automatically generated from RTL by Novas' *Debussy* debugging tools)

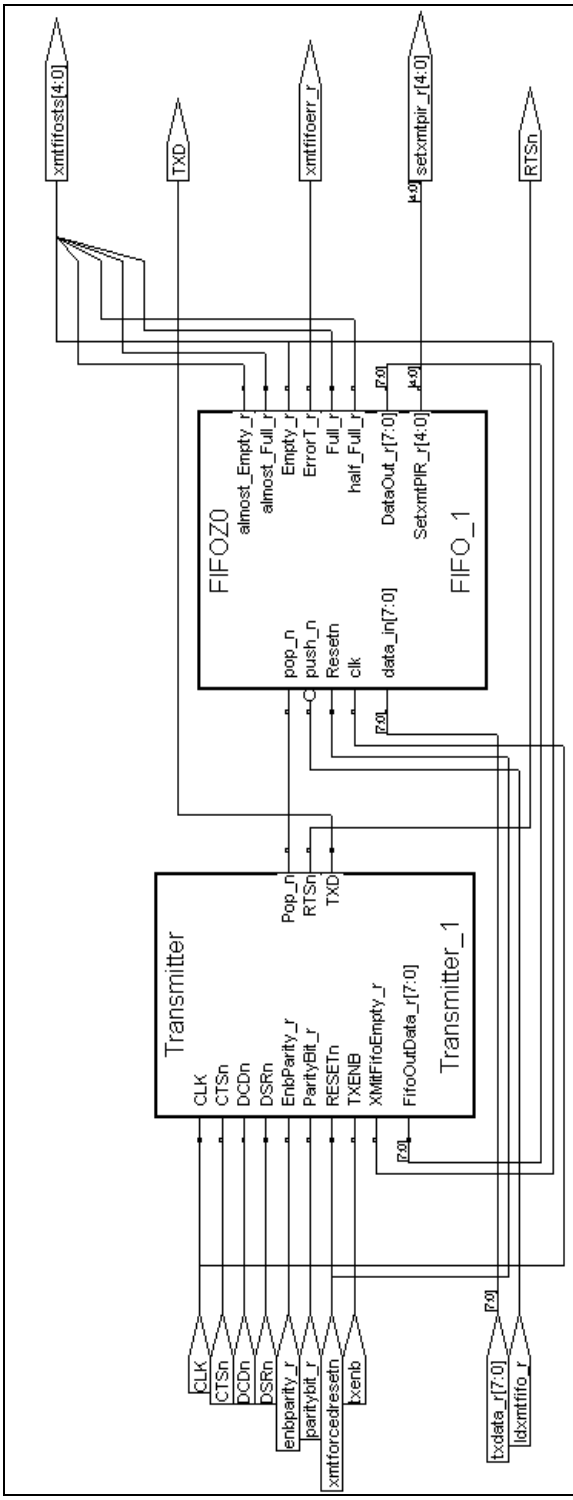


Figure 5.1.4-3 Transmit Subblock Hierarchy (from Synplicity's *Synplify*)

The Transmitter component is responsible for the following tasks:

1. It serializes FIFO data into the T*xD* port per required format.
2. It performs Overflow error
3. It provides to a FIFO the PUSH and POP controls. Data from the CPU is pushed into the FIFO. Data to be transmitted is POPPED from the FIFO.

Figure 5.1.4-3 is view of the processes within the Transmitter component that consists of three major processes:

1. *Xmit_Proc* for serialization process
2. *Counter_Proc* for the count of Transmit bauds, based on parity and number of bits per word.
3. *FSM_Proc* for the definition of the transmit state machine.
4. *Statereg_Proc* for the storage of the FSM
5. *XmitFifoEmpty* for the detection of the FIFO empty logic

The FIFO component is the same as described in the receive section. However, the logic required a different pipeline delay for the FIFO output between the FIFO used in the receiver logic versus the FIFO used in the transmit logic. This difference in behavior is handled within the code with the "generate" feature of VHDL.

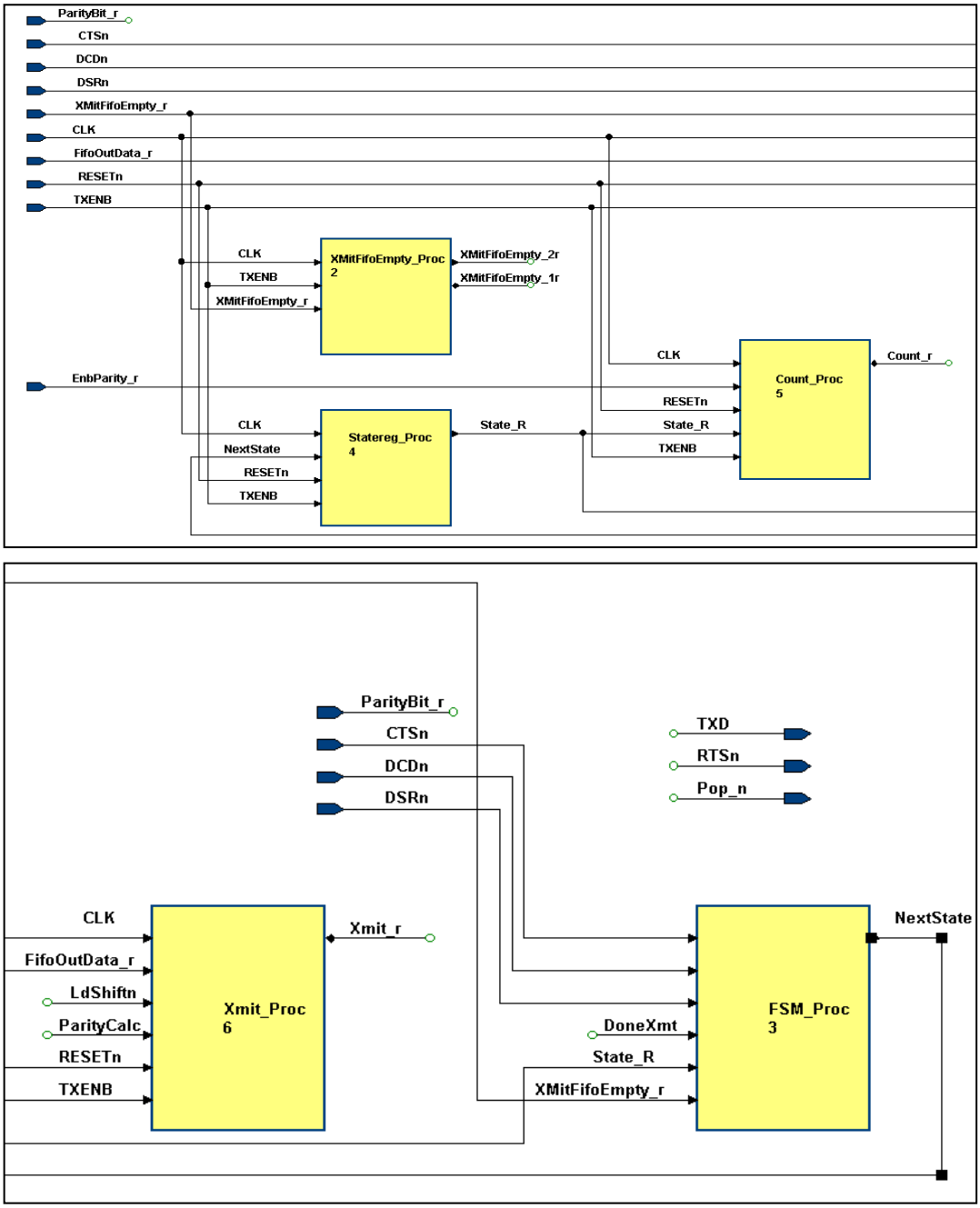


Figure 5.1.4-3 Transmit Component processes (*from Renoir's BD view)*

5.1.5 UART Model

Figure 5.1 UART demonstrates the UART hierarchy and its subblocks. The UART VHDL Code is in file *rtl/uart.vhd*, and is included on page 115.

Xmitsublk 1/3

Xmitsublk 2/3

Xmitsublk 3/3/

Transmitter 1/3

Transmitter 2/3

Transmitter 3/3

Uart.vhd1/5

Uart top 2/5

Uart top 3/5

Uart top 4/5

Uart top 5/5

5.1.6 Compilation

Table 5.1.6 is a compilation script of the UART model for ModelSim compiler. It demonstrates the compilation order. Users of Cadence *Affirma NC VHDL* compiler/simulator can use the following command:

ncvhdl -v93 -log ncvhdl.log -messages path/vhdl_file_name.vhd

Note that the file *Size_Pkg.vhd* is only used for testbench, but includes global signals that are directed off through pragmas. The package is needed for normal compilation because some RTL models assign to those global signals through OFF directives. The code within the OFF directives is ignored by synthesis, but not by VHDL compilation.

Table 5.1.6 Compilation Order for UART

```
vcom -explicit -work work_lib -93 vhdl/tb/Size_Pkg.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/fifo.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/transmitter.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/xmitsublk.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/receiver.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/rcvsublk.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/cpuif.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/clkcntrl.vhd
vcom -explicit -work work_lib -93 vhdl/rtl/uart.vhd
```

5.1.7 Synthesis

Synplify from Synplicity was used for the synthesis and the generation of the EDIF netlist for the chosen Altera FPGA. The project file, created by *Synplify*, defines the compilation order, parameters, and device selection. It is shown in Figure 5.1.7-1

```
#-- Synplicity, Inc.
#-- Synplify version 5.3.1
#-- Project file C:\path\VHDL\RTL\uart.prj
#-- Written on Wed Aug 09 13:46:45 2000
#device options
set_option -technology FLEX10K
set_option -part EPF10K10
set_option -package LC84
set_option -speed_grade –3

#add_file options
add_file -vhdl -lib work "c:/path/vhdl/tb/size_pkg.vhd"
add_file -vhdl -lib work "clkcntrl.vhd"
add_file -vhdl -lib work "fifo.vhd"
add_file -vhdl -lib work "transmitter.vhd"
```

```
add_file -vhdl -lib work "xmitsublk.vhd"
add_file -vhdl -lib work "receiver.vhd"
add_file -vhdl -lib work "rcvsublk.vhd"
add_file -vhdl -lib work "cpuif.vhd"
add_file -vhdl -lib work "uart.vhd"
add_file -constraint "uart.sdc"

#compilation/mapping options
set_option -default_enum_encoding onehot
set_option -symbolic_fsm_compiler false
set_option -resource_sharing true

#map options
set_option -frequency 25.000
set_option -map_logic true
set_option -cliquing true

#simulation options
set_option -write_verilog false
set_option -write_vhdl true

#automatic place and route (vendor) options
set_option -write_apr_constraint true

#MTI Cross Probe options
set_option -mti_root ""

#set result format/file last
project -result_file "uart.edf"
```

Figure 5.1.7-1 Synplify Project File

The following information was extracted from the *Synplify* report log

1. **Warnings**

UNUSED inputs

@W:"c:\path\vhdl\rtl\cpuif.vhd":80:4:80:13|Input **xmtfifosts** is unused
@W:"c:\path\vhdl\rtl\clkcntrl.vhd":39:4:39:11|Input **tc_synch** is unused
@W:"c:\path\vhdl\rtl\clkcntrl.vhd":40:4:40:11|Input **rc_synch** is unused
@W:"c:\path\vhdl\rtl\transmitter.vhd":87:9:87:14|
All reachable assignments to **xmit_r(10)** assign '0', register removed by optimization
@W:"c:\path\vhdl\rtl\fifo.vhd":57:4:57:9|Removing sequential instance **UART_1.xmitsublk_1.FIFO_1.Full_r** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\ path \vhdl\rtl\fifo.vhd":52:4:52:17|Removing sequential instance **UART_1.xmitsublk_1.FIFO_1.almost_Empty_r** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\ path \vhdl\rtl\fifo.vhd":58:4:58:14|Removing sequential instance **UART_1.xmitsublk_1.FIFO_1.half_Full_r** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\ path \vhdl\rtl\fifo.vhd":53:4:53:16|Removing sequential instance **UART_1.xmitsublk_1.FIFO_1.almost_Full_r** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\ path \vhdl\rtl\fifo.vhd":59:4:59:14|Removing sequential instance **UART_1.rcvrsublk_1.FifoRcv_1.SetxmtPIR_r[0]** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\path\vhdl\rtl\fifo.vhd":59:4:59:14|Removing sequential instance **UART_1.rcvrsublk_1.FifoRcv_1.SetxmtPIR_r[1]** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\path\vhdl\rtl\fifo.vhd":59:4:59:14|Removing sequential instance **UART_1.rcvrsublk_1.FifoRcv_1.SetxmtPIR_r[2]** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs
@W:"c:\path\vhdl\rtl\fifo.vhd":59:4:59:14|Removing sequential instance **UART_1.rcvrsublk_1.FifoRcv_1.SetxmtPIR_r[3]** of view:ALTERA.S_DFF(PRIM) because there are no references to its outputs

The removal of these signals was expected because the design did not attempt to optimize for unused resources.

2. **Performance Summary**

Clock	Requested Frequency	Estimated Frequency	Requested Period	Estimated Period	Slack
System	25.0 MHz	69.4 MHz	40.0	14.4	25.6
CLK	25.0 MHz	52.9 MHz	40.0	18.9	21.1

3. **Interface Information** ***(****output information shown here)*

```
Port        Reference      User         Arrival Required
Name        Clock          Constraint   Time    Time       Slack
-------------------------------------------------------------
DO[0]       CLK [rising]   0.0          16.7    40.0       23.3
…
DO[7]       CLK [rising]   0.0          16.7    40.0       23.3
DTRn        CLK [rising]   0.0          3.9     40.0       36.1
INTRPT[0]   CLK [rising]   0.0          8.7     40.0       31.3
INTRPT[1]   CLK [rising]   0.0          8.8     40.0       31.2
OutEnb      System         0.0          4.9     40.0       35.1
RTSn        CLK [rising]   0.0          1.0     40.0       39.0
TXD         CLK [rising]   0.0          1.0     40.0       39.0
=============================================================
```

4. Critical Path with worst case slack = 21.1 ns:

The start and the end point of this path are clocked by the CLK [rising]

```
Instance/Net                                                Pin      Pin     Arrival     Delta     Fan
Name                                                Type    Name     Dir     Time        Delay     Out
-------------------------------------------------------------------------------------------------------
rcvrsublk_1.FifoRcv_1.count_r[2]                    S_DFF   Q        Out     3.0         3.0
rcvrsublk_1.FifoRcv_1.count_r[2]                    Net                                  6
rcvrsublk_1.FifoRcv_1.G_159                         S_LUT   I0       In      3.0
rcvrsublk_1.FifoRcv_1.G_159                         S_LUT   OUT      Out     7.1         4.1
rcvrsublk_1.FifoRcv_1.G_159                         Net                                  4
rcvrsublk_1.FifoRcv_1.un1_un1_un5_count_r_i_or2     S_LUT   I0       In      7.1
rcvrsublk_1.FifoRcv_1.un1_un1_un5_count_r_i_or2     S_LUT   OUT      Out     11.2        4.1
rcvrsublk_1.FifoRcv_1.un1_un1_un5_count_r_i_or2     Net                                  4
rcvrsublk_1.FifoRcv_1.un1_count_r_1_add1            S_CAR   I0       In      11.2
rcvrsublk_1.FifoRcv_1.un1_count_r_1_add1            S_CAR   COUT     Out     12.4        1.2
rcvrsublk_1.FifoRcv_1.un1_count_r_1_carry_1         Net                                  1
rcvrsublk_1.FifoRcv_1.un1_count_r_1_add2            S_CAR   CIN      In      12.4
rcvrsublk_1.FifoRcv_1.un1_count_r_1_add2            S_CAR   OUT      Out     14.6        2.2
rcvrsublk_1.FifoRcv_1.un1_count_r_1_add2            Net                                  1
rcvrsublk_1.FifoRcv_1.Pointers_Proc_count_r_8_0_and2[2]  S_LUT  I1     In      14.6
rcvrsublk_1.FifoRcv_1.Pointers_Proc_count_r_8_0_and2[2]  S_LUT  OUT    Out     16.7
2.1
rcvrsublk_1.FifoRcv_1.Pointers_Proc_count_r_8_0_and2[2]  Net                              1
rcvrsublk_1.FifoRcv_1.count_r[2]                    S_DFF   D        In      16.7
=======================================================================================
```

Setup requirement on this path is 2.2 ns.

This path was generated from *Synplify* and is shown in Figure 5.1.7-2

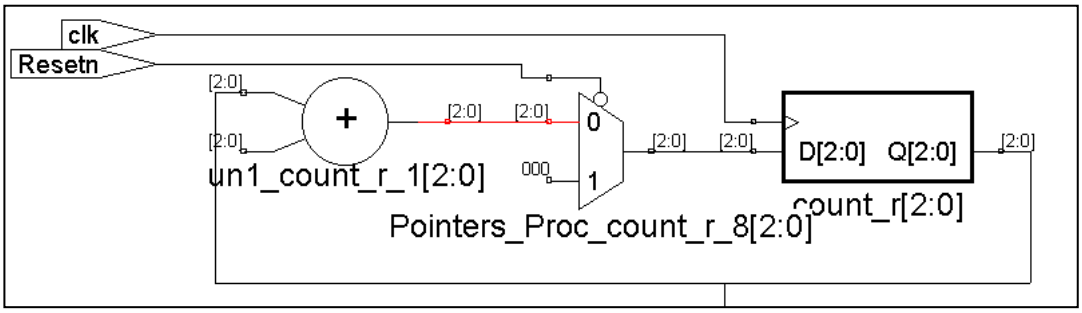


Figure 5.1.7-2 Worst Case Path as Identified, and viewed with *Synplify*

5.1.8 Layout

The EDIF output of *Synplify* was used with the Altera *MaxPlus* layout and timing analysis tool. After layout, the following paths were reported as worst case.

Info: Delay path from '|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_0_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.D': 20.6ns (Clock period: 22.8ns)
Info: Delay path from '|xmitsublk:xmitsublk_1|transmitter:Transmitter_1|state_r_0_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.D': 20.3ns (Clock period: 22.5ns)
Info: Delay path from '|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.D': 19.9ns (Clock period: 22.1ns)
Info: Delay path from '|xmitsublk:xmitsublk_1|transmitter:Transmitter_1|state_r_i_1_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.D': 19.8ns (Clock period: 22.0ns)
Info: Delay path from '|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_1_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_2_.D': 19.4ns (Clock period: 21.6ns)
Info: Delay path from '|clkcntrl:ClkCntrl_1|RxENb1.Q' to
'|rcvrsublk:rcvrsublk_1|fifoz1:FifoRcv_1|count_r_2_.D': 19.4ns (Clock period: 21.6ns)
Info: Delay path from '|clkcntrl:ClkCntrl_1|RxENb1.Q' to
'|rcvrsublk:rcvrsublk_1|fifoz1:FifoRcv_1|count_r_1_.D': 19.4ns (Clock period: 21.6ns)
Info: Delay path from '|rcvrsublk:rcvrsublk_1|receiver:Receiver_1|state_r_0_.Q' to
'|rcvrsublk:rcvrsublk_1|fifoz1:FifoRcv_1|count_r_2_.D': 19.2ns (Clock period: 21.4ns)
Info: Delay path from '|rcvrsublk:rcvrsublk_1|receiver:Receiver_1|state_r_0_.Q' to
'|rcvrsublk:rcvrsublk_1|fifoz1:FifoRcv_1|count_r_1_.D': 19.2ns (Clock period: 21.4ns)
Info: Delay path from '|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|count_r_0_.Q' to
'|xmitsublk:xmitsublk_1|fifoz0:FIFO_1|fifo_r_3_6_.D': 18.6ns (Clock period: 20.8ns)

With *Synplify* RTL view and the "extract net", "filter schematic, and "expand to register/port" features of the tool, the path from the counter to the FIFO is demonstrated in Figure 5.1.7-2. The most critical path was identified by *Synplify* and is shown in Figure 5.1.8-1.

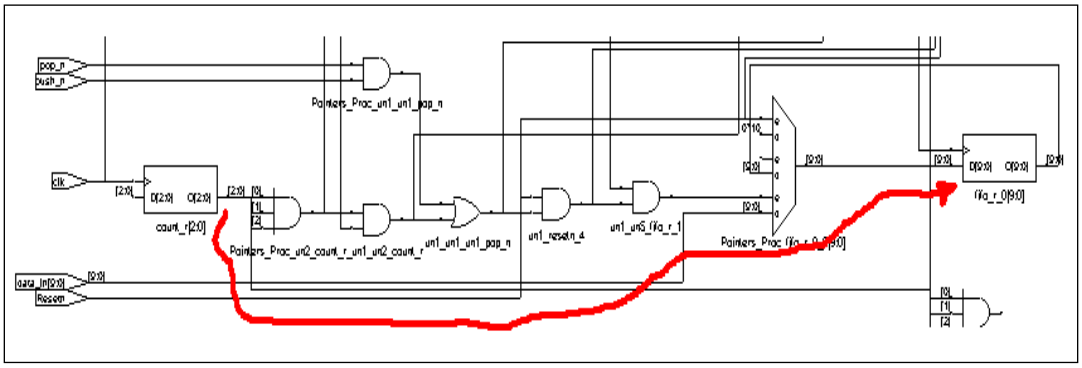


Figure 5.1.8-1 Worst-Case Path as Defined by *MaxPlus* and Drawn by *Synplify*

The *MaxPlus* timing report and expected operating frequency is shown is Figure 5.1.8-2 and 5.1.8-3.

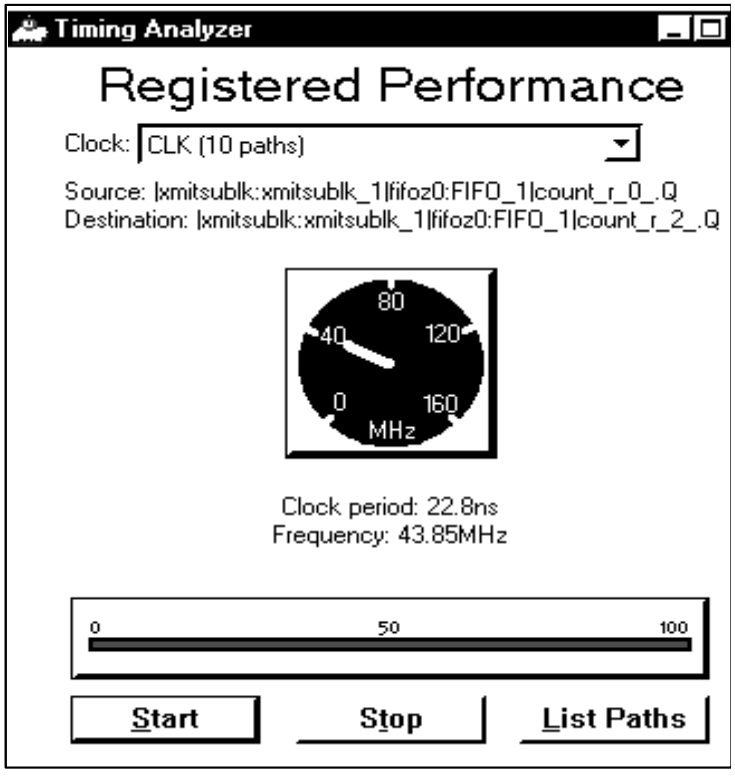
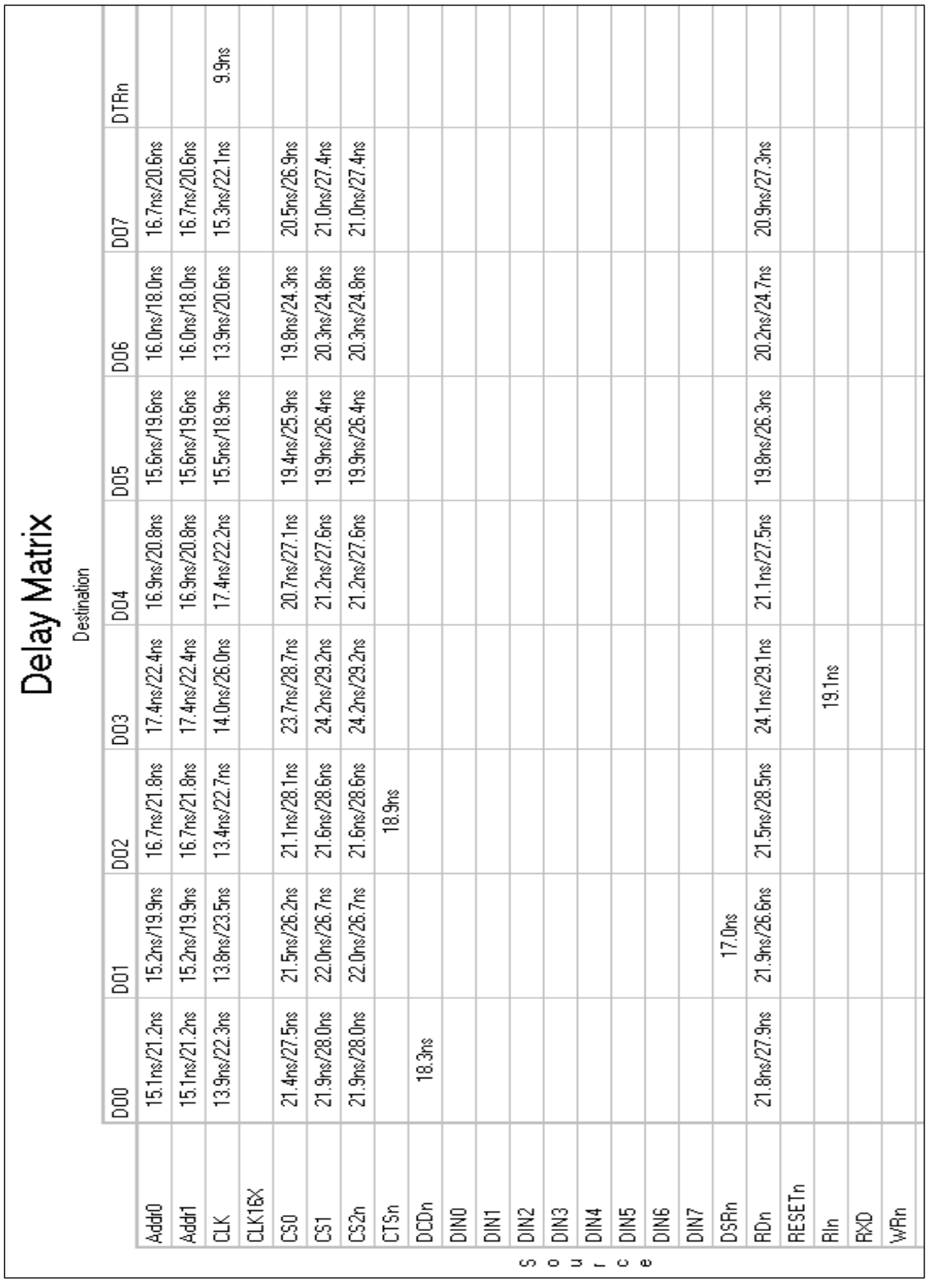


Figure 5.1.8-2 *MaxPlus* Layout Timing report



Delay Matrix

Destination

Source	DO0	DO1	DO2	DO3	DO4	DO5	DO6	DO7	DTRn
Addr0	15.1ns/21.2ns	15.2ns/19.9ns	16.7ns/21.8ns	17.4ns/22.4ns	16.9ns/20.8ns	15.6ns/19.6ns	16.0ns/18.0ns	16.7ns/20.6ns	
Addr1	15.1ns/21.2ns	15.2ns/19.9ns	16.7ns/21.8ns	17.4ns/22.4ns	16.9ns/20.8ns	15.6ns/19.6ns	16.0ns/18.0ns	16.7ns/20.6ns	
CLK	13.9ns/22.3ns	13.8ns/23.5ns	13.4ns/22.7ns	14.0ns/26.0ns	17.4ns/22.2ns	15.5ns/18.9ns	13.9ns/20.6ns	15.3ns/22.1ns	9.9ns
CLK16X									
CS0	21.4ns/27.5ns	21.5ns/26.2ns	21.1ns/28.1ns	23.7ns/28.7ns	20.7ns/27.1ns	19.4ns/25.9ns	19.8ns/24.3ns	20.5ns/26.9ns	
CS1	21.9ns/28.0ns	22.0ns/26.7ns	21.6ns/28.6ns	24.2ns/29.2ns	21.2ns/27.6ns	19.9ns/26.4ns	20.3ns/24.8ns	21.0ns/27.4ns	
CS2n	21.9ns/28.0ns	22.0ns/26.7ns	21.6ns/28.6ns	24.2ns/29.2ns	21.2ns/27.6ns	19.9ns/26.4ns	20.3ns/24.8ns	21.0ns/27.4ns	
CTSn			18.9ns						
DCDn	18.3ns								
DIN0									
DIN1									
DIN2									
DIN3									
DIN4									
DIN5									
DIN6									
DIN7									
DSRn		17.0ns							
RDn	21.8ns/27.9ns	21.9ns/26.6ns	21.5ns/28.5ns	24.1ns/29.1ns	21.1ns/27.5ns	19.8ns/26.3ns	20.2ns/24.7ns	20.9ns/27.3ns	
RESETn									
RIn				19.1ns					
RXD									
WrRn									

Figure 5.1.8-3 *MaxPlus* Layout Timing report

5.1.9 Area Statistics

Table 5.1.9 is a summary of the resources used by the UART, as reported by Altera's *MaxPlus II* layout tool. The complete report is available on disk in file *altera/uart.rpt.*

Table 5.1.9 MaxPlus II Resource Report

```
Embedded              Column       Row
Array     Embedded    Interconnect Interconnect         Read/
External
Block     Cells       Driven       Driven       Clocks  Write
Interconnect


Total dedicated input pins used:                 6/6      (100%)
Total I/O pins used:                            31/53     ( 58%)
Total logic cells used:                        419/576    ( 72%)
Total embedded cells used:                       0/24     (  0%)
Total EABs used:                                 0/3      (  0%)
Average fan-in:                               3.00/4      ( 75%)
Total fan-in:                                1258/2304    ( 54%)

Total input pins required:                      23
Total input I/O cell registers required:         0
Total output pins required:                     14
Total output I/O cell registers required:        0
Total buried I/O cell registers required:        0
Total bidirectional pins required:               0
Total reserved pins required                     0
Total logic cells required:                    419
Total flipflops required:                      219
Total packed registers required:                 0
Total logic cells in carry chains:               0
Total number of carry chains:                    0
Total logic cells in cascade chains:             0
Total number of cascade chains:                  0
Total single-pin Clock Enables required:         0
Total single-pin Output Enables required:        0

Synthesized logic cells:                        78/ 576   ( 13%)
```

6 DESIGN VERIFICATION

This section provides the verification models and test results of the UART that was specified in the requirement document. The verification models follow the directives defined in the verification plan. A description of the testbench models is first provided and is then followed by the test verification control files for the generation of directed test scenarios, and the test results.

6.1 OVERVIEW

Functional verification of the UART design is performed through simulation of the RTL code, and regression testing of the gate level model generated by the layout tool. The UART testbench represents the test environment described in the verification plan, and is shown graphically in Figure 6.1-1. Table 6.1 displays the testbench elements.

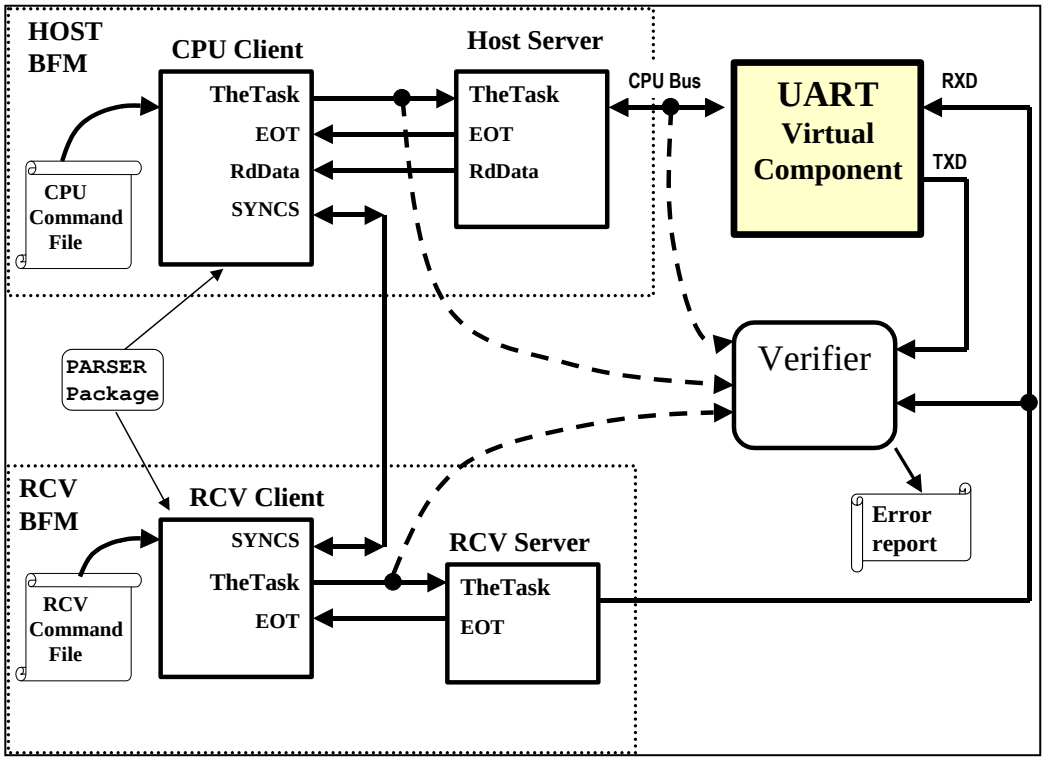


Figure 6.1-1 Testbench Architecture Overview

The testbench provides stimulus waveforms to the unit under test, and can provide means for automatic verification of the model for compliance to requirements. In the generation of the stimulus vectors, the process starts with the client models responsible for the creation of tasks or jobs to be expanded into waveforms by the server models. The definition of the sequence of tasks can be expressed in text files, and /or VHDL code. The verifier makes use of the tasks to be transacted and the interface signals of the UUT to validate the operations of the UART.

6.2 PARSER PACKAGE

One of the key design units for this testbench design is the parser package that provides the following functions:

1. It defines data types for the transactions and tasks.

2. It defines the subprograms necessary to parse instructions defined in text files.
3. It defines the subprograms to execute the process of reading files and generating the tasks.

Figure 6.2-1 demonstrates an example of a task command for a CPU READ instruction, the execution of the task, and the transmission of the end-of-task (EOT) back to the client to enable the generation of another task. The code for the parser package is in file *tb/parser_tb.vhd* and is included on page 133.

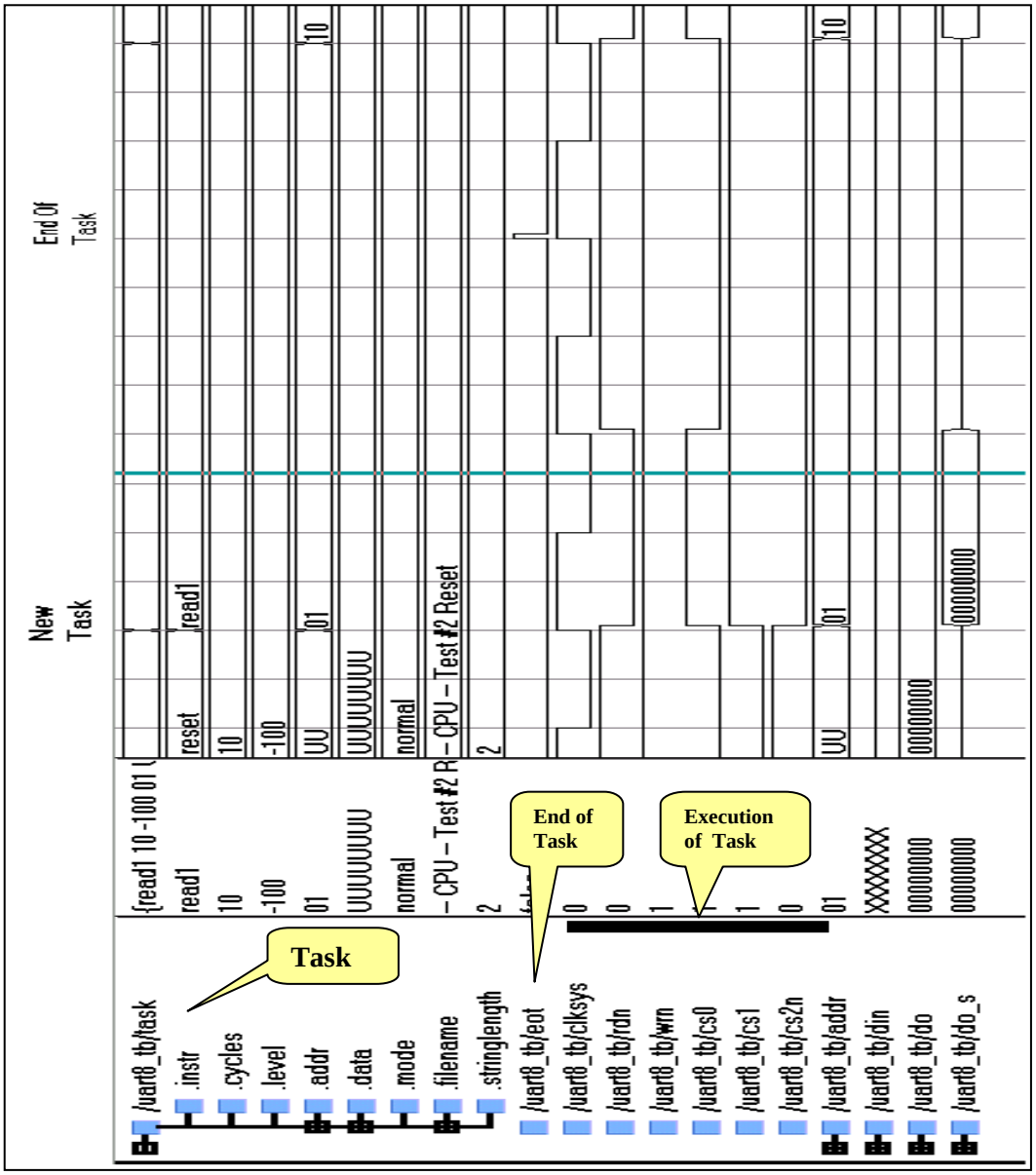
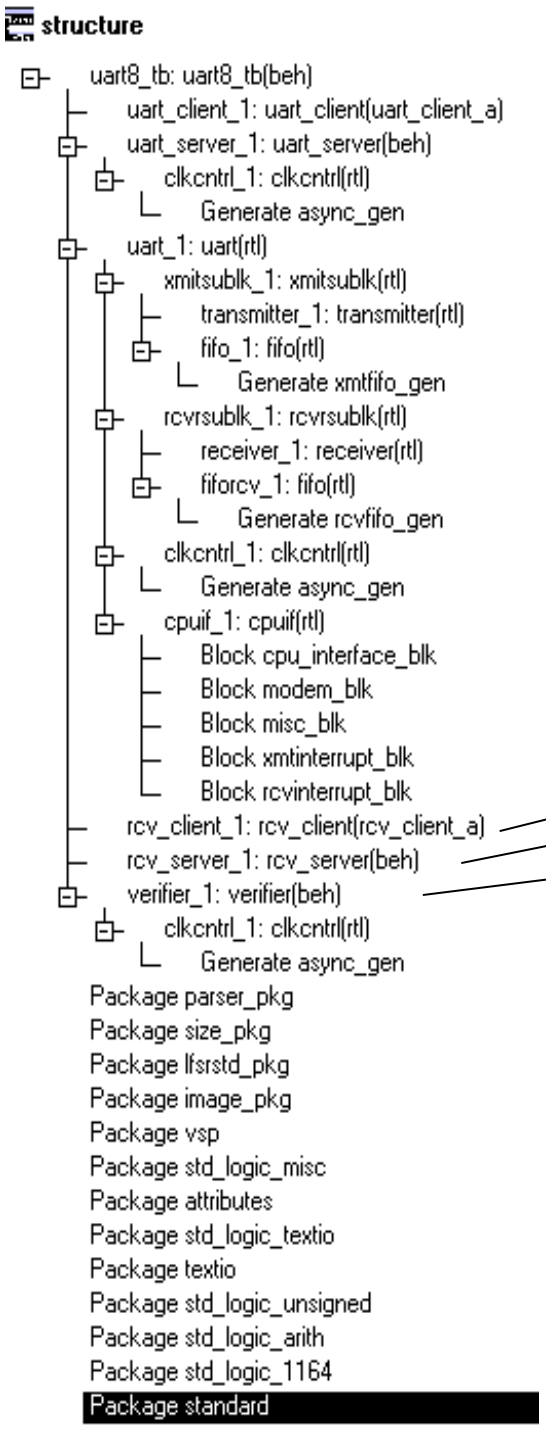


Figure 6.2-1 Task Command and Execution Example (*ModelSim EE 5.4b)*

Table 6.1 Testbench Elements (*ModelSim EE 5.4b)*

TESTBENCH ELEMENTS	DESCRIPTION
	- Top level testbench - CPU Client for definition of transactions - CPU Server for implementation of transactions - UART model, representing the unit under test - Receiver Client for receive transactions - Receive Server for receive interface - Verifier monitor for automatic verification and logging of transactions and errors. - Package supporting instruction parsing - Package defining width of word for TB - Package for pseudo-random numbers - Package for conversion to strings - Package for conversion functions - Package for Reduce operator - Referred package - Package for fileIO of Std_Logic objects - Package for textIO - Package for addition operators - Package for conversion functions - Package for definition of Std_Logic - Default package

Parser_pb.vhd 1/9

Parser_pb.vhd 2/9

Parser_pb.vhd 3/9

Parser_pb.vhd 4/9

Parser_pb.vhd 5/9

Parser_pb.vhd 6/9

Parser_pb.vhd 7/9

Parser_pb.vhd 8/9

Parser_pb.vhd 9/9

6.3 CLIENT MODEL

The client model may fetch and parse the sequence of instructions from files, as described in the verification plan. The parser package includes all the necessary supporting types and subprograms to achieve the parsing function of the text file into the task elements. The user needs only to call the *ExecuteControlFile* subprogram as shown below:

```
ExecuteControlFile (
      FileName_c   => ControlFile_g, --string: file path & name or generic
      Task         => Task, -- signal:  output of client
      ServerData   => ServerData, --signal: Data back to client from server
      EOT          => EOT, -- signal: end of task from server
      Clk          => ClkSys, -- signal: System clock
      Task_v       => Task_v,  -- Local variable: used for subroutine call
      LfsrData_v   => LfsrData_v); – Local variable: Pseudo-random data
```

The UART client model is also capable of generating the tasks directly from VHDL, without the parsing of files. This is used in the UART verification to generate a set of pseudo-random tasks, and to then wait for interrupts prior to generating other tasks based on the source of the interrupts. This emulates the operations of a CPU. Sample code demonstrating this concept is shown below.

```
      -- Send random data, check for interrupts
      if SendDATA then – Write "n" data words to UART
        for I in 0 to conv_integer(LfsrData_v(1 downto 0)) loop
          Task_v.Instr := WRITE1;
          Task_v.Addr := "11";
          Task_v.Data :=  LfsrData_v;
          LfsrData_v := LFSR(LfsrData_v);
          wait until ClkSys = '1';
          Task <= Task_v;
          wait until EOT;
          wait for conv_integer(LfsrDelay_v(1 downto 0)) * ClkPeriod_c;
        end loop;  -- I
     end if;

      SendDATA <= False; -- wait till MT interrupt
      wait on INTRPT; -- wait for an interrupt
      wait until ClkSys = '1';
      if INTRPT(0) = '1' then   -- Check which interrupt is active
        -- Receive interrupt, Read data
        Task_v.Instr := READ1;   -- read receive PIR
        Task_v.Addr  := "01";         -- receive status
        Task <= Task_v;
        wait until EOT;
```

Specifiying pseudo-random number of WRITE tasks with pseudo-random data at pseudo-random times (delays)

Wait for interrupts. Determine which interrupt to handle

Instructions from the client model are processed by the server model. Figure 6.3-1 is a block diagram view of the CPU client/server interface, and Figure 6.3-2 is a block diagram view of the receive side of the client /server interface (i.e., the side that generates the *Rxd* signal).

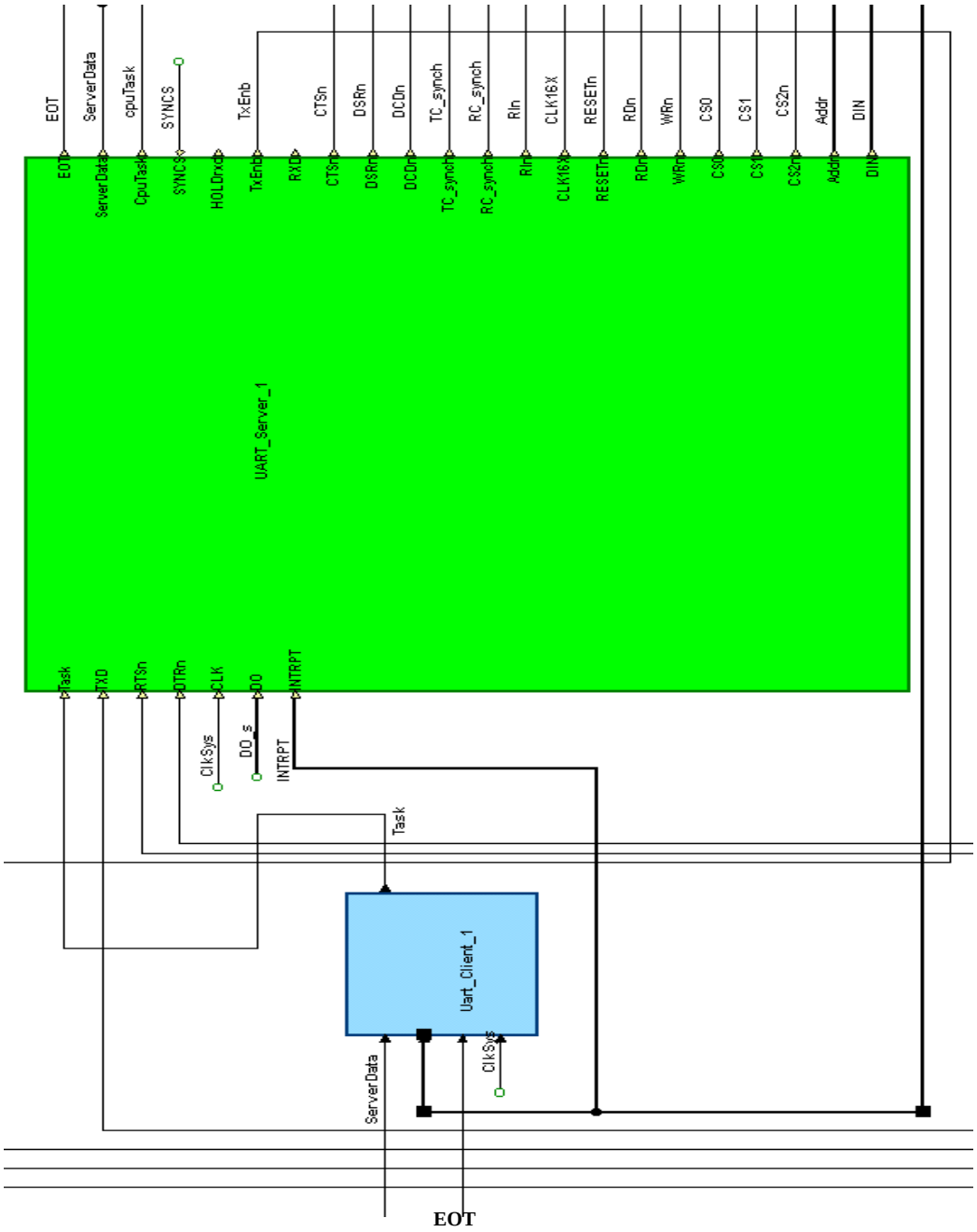


Figure 6.3-1 CPU Client/Server Interface in Testbench (*from Renoir's BD view)*

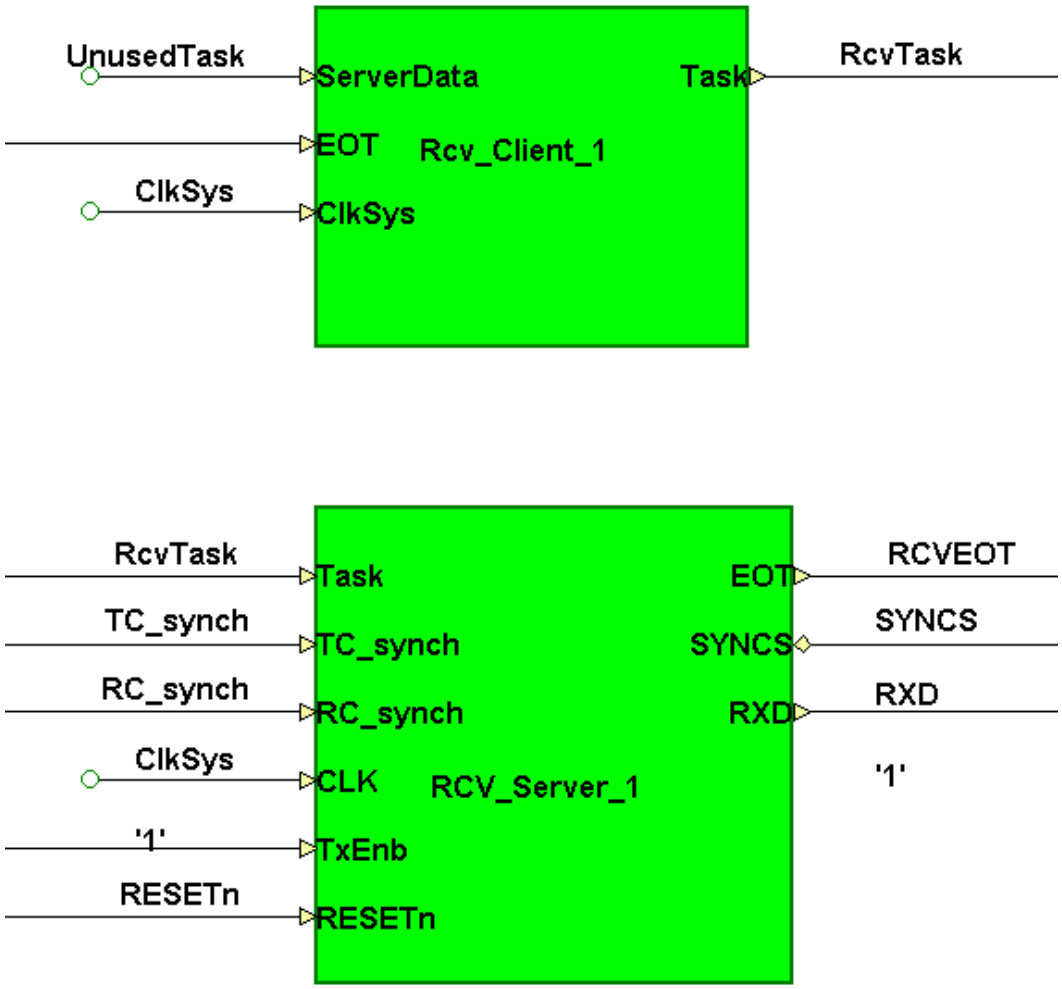


Figure 6.3-2 Receive Client/Server Interface in Testbench
(*from Renoir's BD view)*

The unused task is a signal that was originally intended for receipt of collected data to the receive client. However, it was not used, and remained as spare.

The UART client model is in file *tb/uart_clientrndm.vhd* and is shown on page 145. The receiver client model is in file *tb/rcv_client.vhd* and is shown on page 150.

Uart_clientrndm 1/5

Uart_client 2/5

Uart_client 3/5

Uart_client 4/5

Uart_client 5/5

Rcv_client ½

Rcv client 2/2

6.4 SERVER

The server provides the following functions:

1. It detects a transaction (or new assignment) from the client, thus alerting it of a new job request.
2. It decodes the kind of transaction requested (e.g., READ, WRITE, IDLE).
3. It executes the requested transaction per interface protocol.
4. It collects any data received over the bus interface, and transfers this data back to the client.
5. It sends an End-Of-Transaction (EOT) signal back to the client to inform it of completion of the task.

In this design, the CPU client was used with different servers to verify the functionality of the subblocks. Table 6.4 summarizes the list of server components using the same client. The FIFO server and testbench is included to demonstrate how the same client can be used to verify different subblocks.

Table 6.4 List of Server Components using the same Client

Component Name	File Name	Function
Uart_server	*/tb/Uart_server.vhd* page 153	Server for the UART component
Rcv_server	*/tb/rcv_server.vhd* page 157	Server for the Receiver subblock
Fifo_server	*/tb/Fifo_server.vhd* page 160	Server to verify the FIFO
Fifo_Tb	*/tb/Fifo_tb.vhd* page 162	Testbench for FIFO

To emulate pseudo-random data and delays, the linear feedback shift register (LFSR) package[1] is used. To enable the conversion of various data types to strings, the *Image* package[2] is used. The code for these packages is included on the CD in the testbench (TB) subdirectory for the user's convenience.

[1] Public domain, available at http://www.vhdlcohen.com

[2] Public domain, available at http://www.vhdlcohen.com

uart_server.vhd ¼

Uart_server.vhd 2 of 4

Uart_server.vhd 3 of 4

Uart_server.vhd 4/4/

rcv_server.vhd 1/3

rcv_server.vhd 2/3

rcv_server.vhd 3/3

fifo_server 1/2

fifo_server 2/2

fifo_tb 1/3

fifo_tb 2/3

fifo_tb 3/3

6.5 VERIFIER

6.5.1 ISSUES

The verifier serves several functions:

1. It detects and reports operating and requirements errors.
2. It logs UUT's transactions into a report file(s) to allow for off-line debugging or analysis.

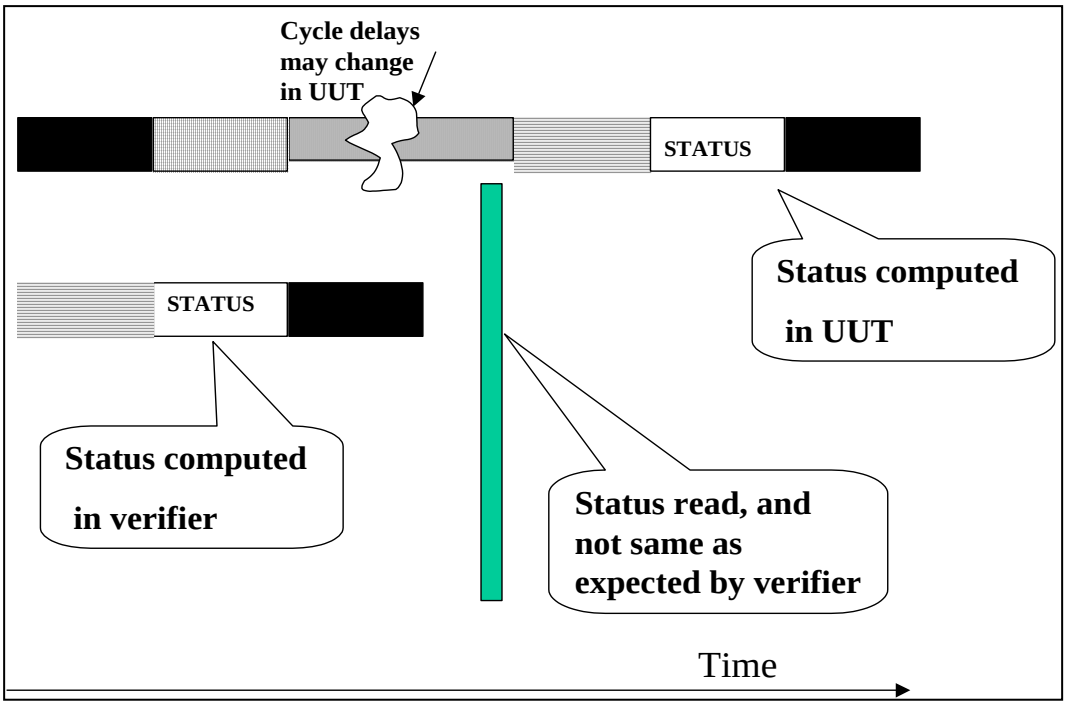
There are several issues to consider in the design of a verifier, including:

1. What is the definition of "expected results"?
2. To what level of timing accuracy is this "expected" data correct? This accuracy can be data accuracy, and/or cycle timing accuracy (i.e., accurate to the bit timing level).
3. When is the design considered verified?

For this design, "expected results" for the transmitted and received data means a bit match to the commanded data, as defined by the tasks within the client model. Thus, if the client instructs the UART to send the string "10110001", then that string formatted in the appropriate serial format must appear on the *TxD* port, provided all the conditions for transmission are enabled. Correspondingly, if the client instructs the UART to receive a serial string, as formatted by the server, then that data will be available at the CPU interface during a READ of the received data.

The "expected results" for the control, status, and computed results is more difficult to define because it is linked to the cycle level of accuracy. If the verifier keeps track of these results at the functional level, but not at the cycle level, then reports of the UUT's results (e.g., READ of PIR) may not match the verifier's computation of those results at the time the READs are performed. The match may eventually occur if enough settling time between the expected computed results and observed results is provided. For example, if the UUT's PIR is updated with a pipeline delay of three cycles, and the computed PIR is updated with zero or one pipeline delay, then a READ of that status register may be misconstrued as an error or mismatch if that READ occurs within this pipeline delay region. This concept is demonstrated in Figure 6.5-1.

There are other issues involved if the verifier does not maintain "cycle accuracy" with the UUT's timing. One such issue is the **loss of state synchronization** between the verifier and the UUT. For example, if the UUT receives a PUSH into a full FIFO (e.g., CPU WRITE of transmit data) while the UUT's internal logic is performing in that same cycle a POP of that full FIFO (to transfer data onto the



serial output), then this IS NOT an error because the transaction is a simultaneous PUSH and POP. Thus, the UUT reports no errors. However, if the verifier does not consider this low level cycle performance of the UUT and is unaware of the simultaneous POP, then the verifier may believe that the PUSH into a full FIFO is an error, and would put it's internal copy of the state (e.g., scoreboard) of the UUT in the error state, rather than the normal state.

Figure 6.5-1 Mismatch between Verifier and UUT because of Lack of Cycle Synchronization

This concept of cycle accuracy is very critical because it would guide the verification approach. This concept is irrelevant of the language (e.g., *VHDL, Verilog, Vera, Specman*) or tools used in the verification model. Maintaining cycle accuracy can be achieved by several methods, including the following:

1. **Design a verifier that is cycle accurate with the UUT**. This approach is complex, and problematic because it requires a low level (RTL) like modeling approach, with internal design information about the UUT, including pipeline delays. The UUT pipeline timing may change during the course of the design due to several factors, including timing margins, or design changes for better interfaces to other subblocks. However, the advantage of this modeling approach is that the UUT is treated as a black box, and the same verifier can be used for recursive tests for all levels of the UUT's implementation (e.g., RTL, gate).
2. **Design a verifier that is synchronized to internal critical signals within the UUT**. This approach enables the design of a verifier at a higher level of abstraction, yet is in synchronism with the UUT. Synchronization is still maintained even if the cycle timing of the UUT changes. The disadvantage of this approach is that the UUT is treated as a gray box, and the same verifier cannot be directly used for recursive tests for all levels of the UUT's implementation (e.g., RTL, gate). The testbench requires modifications to the gate level code, or to the peeking path definition, for the peeking of the critical synchronization signals. Access to the UUT's internal signals can be achieved in VHDL either with assignments to global signals, or with PLI interface calls. Verification languages provide access to internal signals of a design.

Based on the above discussion, the "expected results" for the control, status, and computed results should be synchronized to the UUT's cycle timing to achieve a more accurate verification.

When is the design considered verified? This is a difficult answer to provide. The best answer is probably *when the level of confidence for the correctness of the design meets a comfortable level*. Other definitions would include *when it works in the system (real or emulated), when it meets the specified requirements, when all the code was covered by a functional test pattern*, or *never*. Code coverage is one of the methods that can guide in the determination of when a design is considered verified; but that, by itself is not enough. For this model, the set of compliance tests, as defined in the testplan, and a set of pseudo-random tests will be demonstrated. This will achieve the purpose of demonstrating the verification methodology. A statement code coverage report generated by the built-in coverage tool of *ModelSim* will be provided[3].

6.5.2 Verifier Design Approach

For this design, the verifier will use a high-level, scoreboarding approach for the

[3] For an excellent discussion on code coverage, refer to *Verification Methodology Manual for Code Coverage in HDL Designs*, Michael Stuart & David Dempster, Teamwork International , 2000, ISBN 0-9538-4820-5

posting of transactions (e.g., READ, WRITE) from the client model. Two versions of the verifier will be built. The first version is a GRAY BOX approach synchronized with two critical signals of the UART, the *PUSH* into the transmit FIFO and the *POP* from the transmit FIFO. The second version is a BLACK BOX method that makes a prediction on the *PUSH* timing, and determines that a *POP* occurred when the verifier detects a *START* transaction on the *TxD* data. This is a poor, but simplistic, prediction because the actual *POP* occurs a few cycles before the *START* transaction. The actual cycle timing of the *POP* can vary because it is a function of the state of the transmit state. The RTL design includes a look-ahead prefetch of the FIFO data when there is data in the transmit FIFO and the current serial transmission is about to be completed.

Figure 6.5.2-1 represents a high level view of the verification interfaces and approach. Key features and operations of this design for the transmit verification section include:

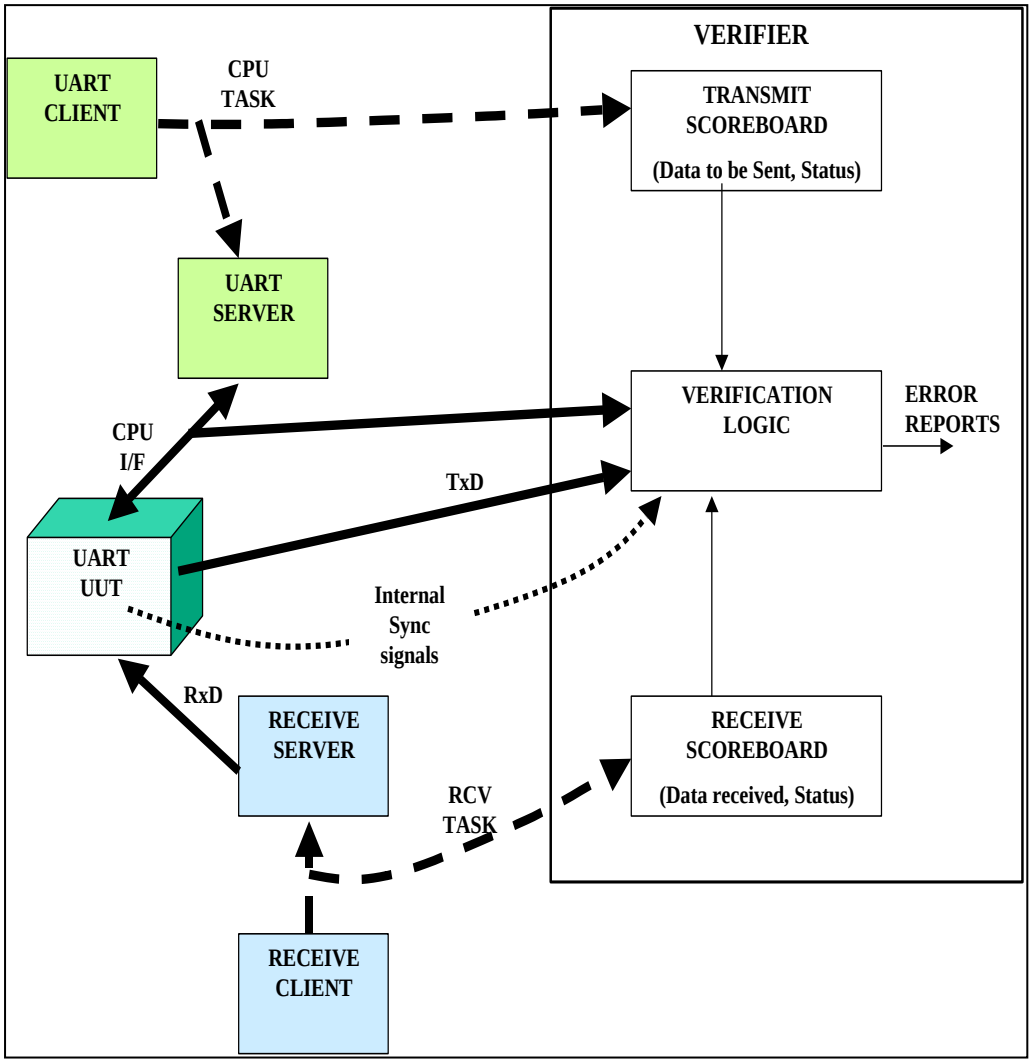


Figure 6.5.2-1 High Level View of the Verification Interfaces and Approach

1. **CPU Interface client** passes the tasks or transactions to both the server and the verifier.
2. **The server** is responsible for generating the interface protocol with the UUT.
3. **The verifier** logs the task into a scoreboard structure. The scoreboard uses a record structure to maintain the information, as shown below:

```
DTRn       : std_logic;  -- Data terminal ready
ParityEnb  : std_logic;  -- parity enable
ParityBit  : std_logic;  -- parity bit
MODEM      : std_logic_vector(3 downto 0); --  4 bits:RIn, CTSn, DSRn, DCn
FIFO       : Fiforeg_Typ;  -- XMt FIFO registers
Reset      : std_logic;  -- software reset
IntrptEnb  : std_logic_vector(7 downto 0);  --ERR, MT, AMT, AF, HF, FL
PIR        : std_logic_vector(7 downto 0);
Count      : integer;  --  range 0 to Depth_g;  -- counts fullness of fifo
WrPntr     : integer;  -- range 0 to Depth_g - 1;  -- Write pointer
RdPntr     : integer;  -- range 0 to Depth_g - 1;  -- Read pointer
Mode       : Mode_Enum;  --  NORMAL, FRAME_ERR, PARITY_ERR
```

4. **The verifier** maintains the status of the scoreboard based either on internal signals of the UUT (e.g., transmit *PUSH* and transmit *POP*) or on derived versions of these signals.
5. **The verifier** de-serializes the transmitted serial data from *TxD* into its constituents (data word and parity), and compares it against the value stored in the scoreboard. The verifier checks for proper protocol observance.
6. **The verifier** monitors the UART CPU interface, and verifies that data deposited by the UUT because of a *READ* matches the expected data maintained in the scoreboard.

Key features and operations of this design for the receive verification section include:

1. **Receive client** passes the tasks or transactions to both the server and the verifier.
2. **The server** is responsible for generating *RxD* receive serial data.
3. **The verifier** logs the task into a scoreboard structure. The scoreboard uses the same record structure defined above.
4. **The verifier** maintains the status of the scoreboard based on the termination of the *Rxd* message (e.g., end of word at *STOP* cycle).

5. **The verifier** monitors the UART CPU interface, and verifies that data deposited by the UUT because of a READ matches the expected data maintained in the scoreboard.

Another task of the verifier (not shown in the figure) is the logging of the tasks and the bus transactions for documentation and debugging. The logging is performed at both the transaction level (as issued by the clients), and at the bus level (as observed on the UUT's ports). Examples of logging transactions are shown in Figure 6.5.2-2. The tasked command is prefixed with the word "Tasked", and is originated from the task. The bus transactions are prefixed with a "**" and provide useful information. Events generated by the UART, such as interrupts, are also displayed. DISPLAY messages inserted in the text command file are also displayed in the log file.

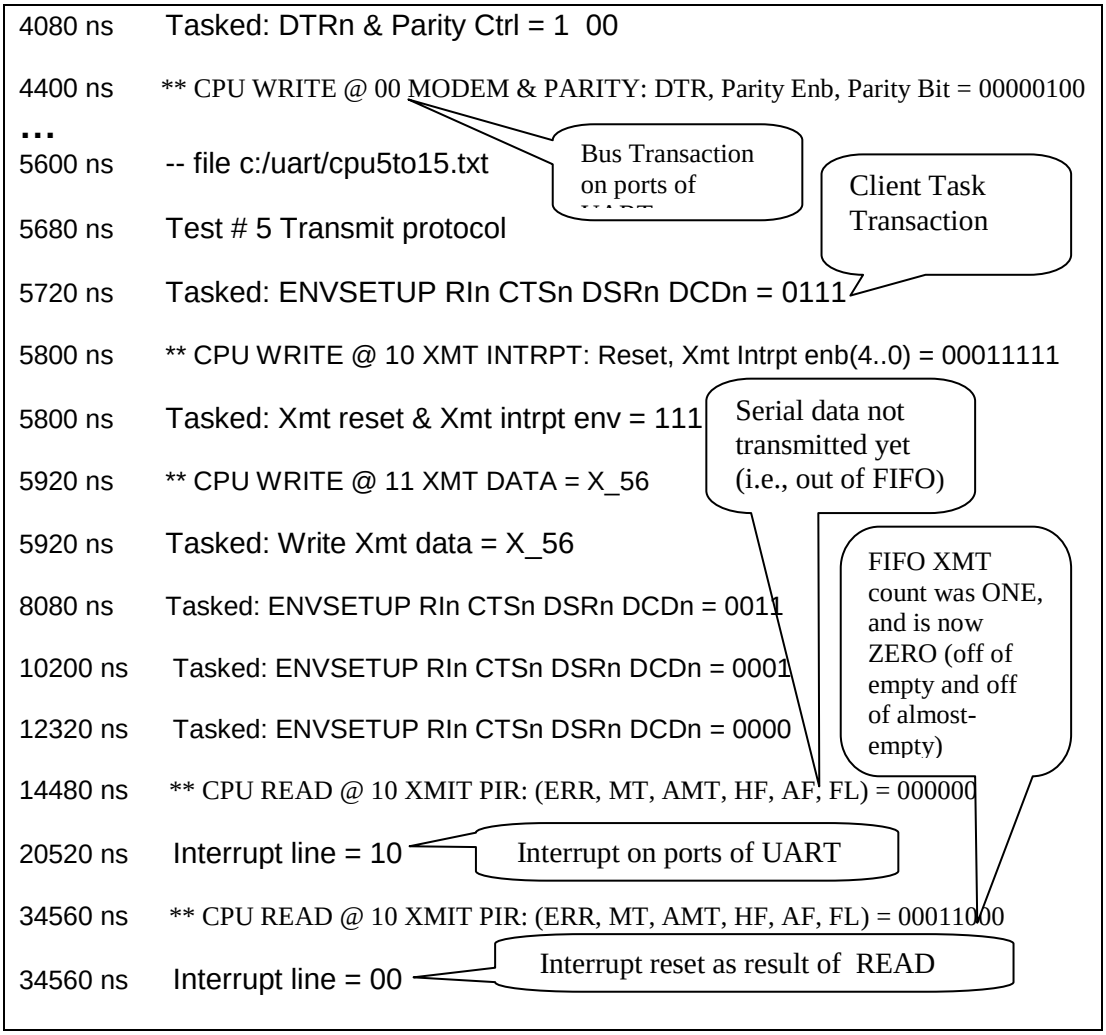


Figure 6.5.2-2 Example Log Messages of Tasked and Bus Transactions Generated by Verifier

6.5.3 Verifier Design

Table 6.5.3 summarizes the processes within the verifier.

Table 6.5.3 Processes within Verifier (Generated by *Renoir)*

PROCESS	FUNCTION
VerifierPeek.bd	
14 - CountEndOfRscMsg_Proc	Verifies and counts and identifies end of *RxD*
13 - DisplaySyncs_Proc	Logs the SYNC signal
12 - CheckRcvInterrupt_Proc	Checks the receive interrupt
11 - CheckXmtInterrupt_Proc	Checks the transmit interrupt
10 - InterruptCheck_Proc	Checks interrupts and directs timing for check
9 - RCVLog_Proc	Updates receive scoreboard
8 - XmtStart_Test_Proc	Checks lack of *TxD* message when one is expected
7 - XmtCheck_Proc	Verifies that *TxD* data is correct as expected
6 - ControlReg_Proc	Logs CPU Write data into UART
5 - RcvFifo_Proc	Logs CPU read data on DO
4 - XmitBit_Proc	Generation of TC_synch and RC_Synch
3 - CheckDTR_Proc	Checks accuracy of commanded DTRn
2 - Driver_Proc	Logs scoreboards
1 - eb1	Concurrent signal assignments
ClkCntrl_1 - work_lib/ClkCntrl/RT	Clock control for accurate timing

Figures 6.5.3a, b, c and d represent block level views of the processes within the verifier model.

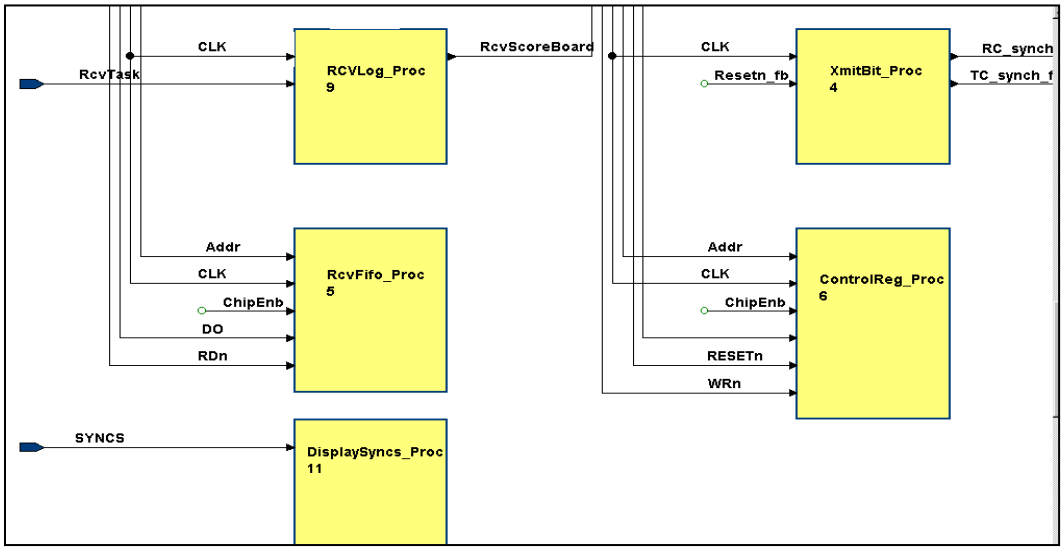


Figure 6.5.3-a Block Level View of the Processes within the Verifier Model (generated by *Renoir*)

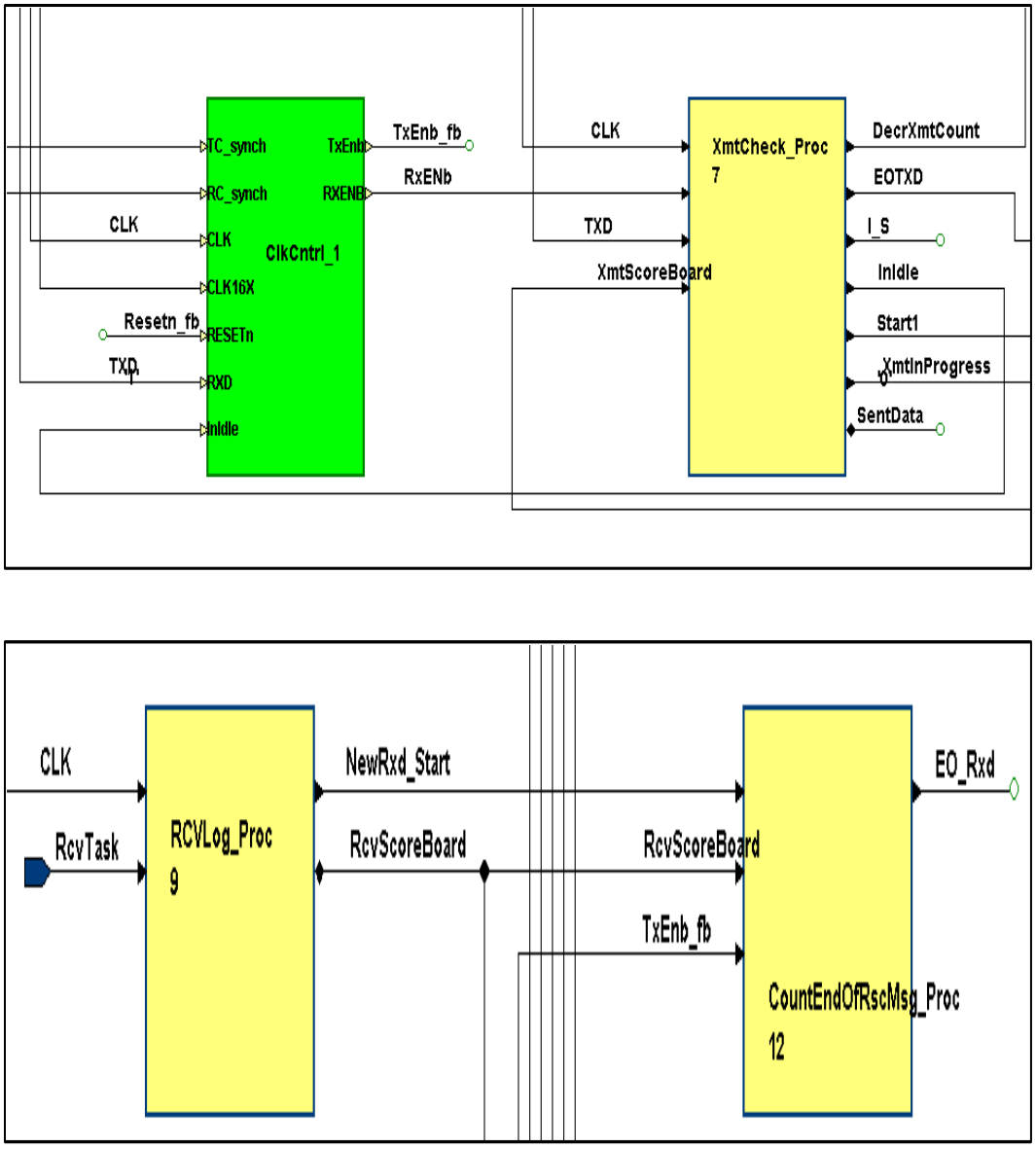


Figure 6.5.3-b Block Level View of the Processes within the Verifier Model (generated by *Renoir*)

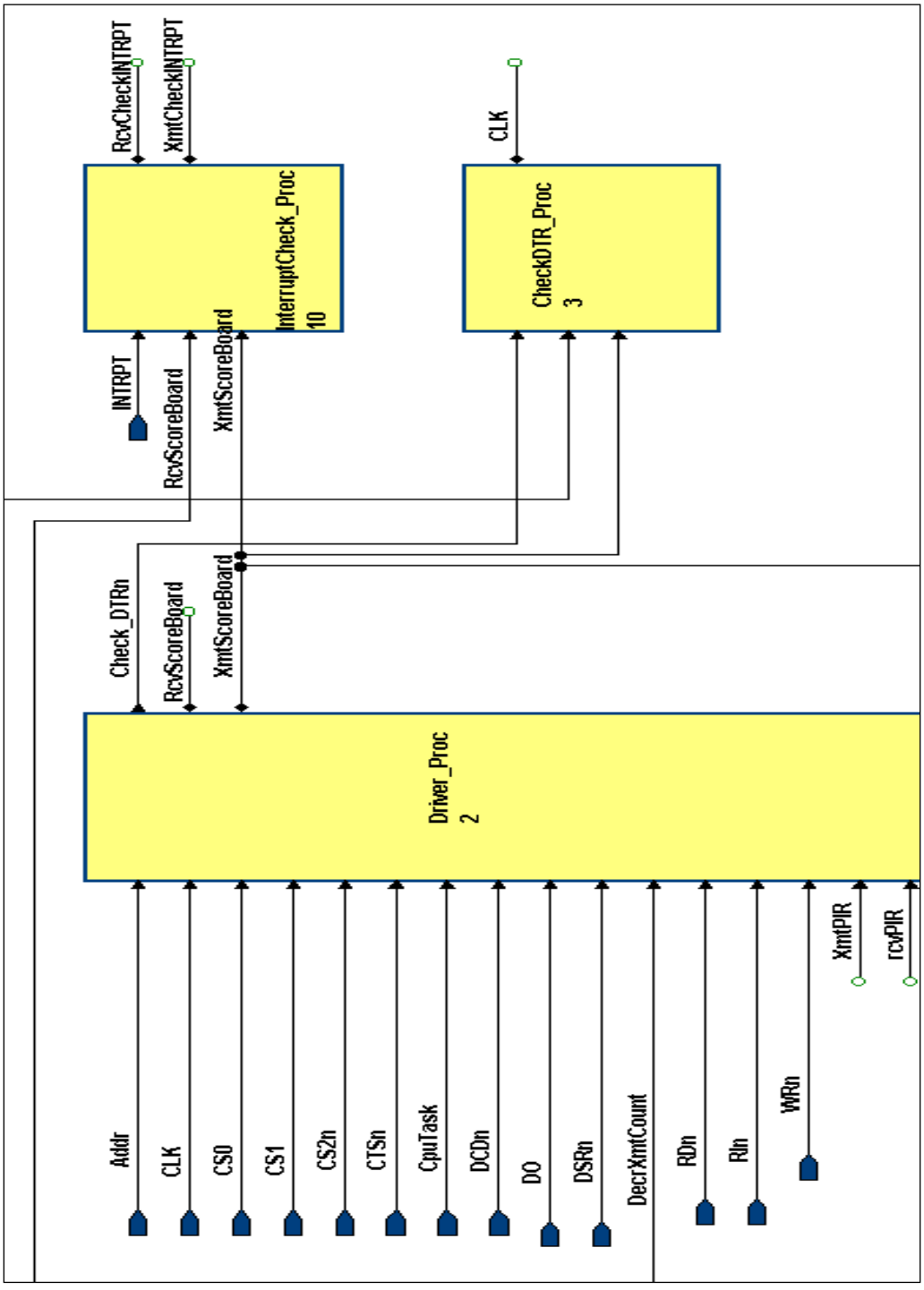


Figure 6.5.3-c Block Level View of the Processes Within the Verifier Model (generated by *Renoir*)

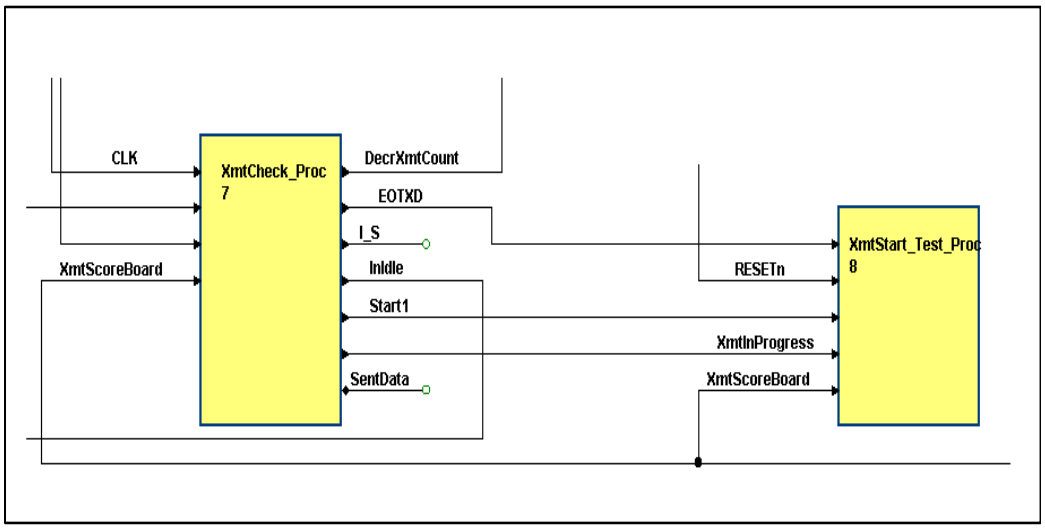


Figure 6.5.3-d Block Level View of the Processes Within the Verifier Model (generated by *Renoir*)

The code for the verifier that makes use of the UUT's internal critical timing signals (i.e., peek into the UART model) is in file *tb/verifierpeek.vhd* and is shown on page 176.. The architecture for the black box verifier is in file *tb/verifierblkbox.vhd* on CD. The peek verifier model requires the *size* package (in file *tb/size_pkg.vhd)* for the access of global signals. This package is shown in Figure 6.5.3-2.

```
-- file size_pkg.vhd
library ieee;
  use ieee.std_logic_1164.all;
package Size_Pkg is
  -- pragma translate_off
  constant WordWidth_c : integer := 8;
  signal   pop_n       : std_logic;
  signal   push_n      : std_logic;
  -- pragma translate_on
end package Size_Pkg;
```

Figure 6.5.3-2 Size Package (tb/size_pkg.vhd)

Some style features of the verifier model include the following:

1. **Constant array for the display of error messages.** This provides not only a collocation of all errors detected by the verifier, but also a means to easily convert an enumerated error type into a string. Example:

```
constant ErrorArray_c : ErrorArray_Typ :=
  (NONE                => "----------------------------------------------------",
   PARITY_FAIL_DETECT  => "`` UART Fails to detect parity error            ",
   PARITY_ERR_WHEN_NONE => "`` UART detects Parity error when none         ",
```

2. **Application of the *image* package for text IO.** This package converts various types (such as integer, std_logic_vectors, time) into strings. Example:

```
Write(L_v, Image(now) &
      "Tasked: DTRn & Parity Ctrl = " &
      Image(not CpuTask.Data(2)) & Image(CpuTask.Data(1 downto 0)));
 Writeline(LogFile_f, L_v);
```

3. **Application of pointers for writing messages into two files.** Once a message string is defined into a variable of type *line*, there is a need to write that message into more than one output, such as the log file and the error file. However, the *WriteLine* procedure deallocates the pointer, and the variable will point to *null*, thus losing the content of the string. The easiest method to maintain the information is to create a copy of the original data pointed by the variable onto another variable, also of type *Line*. For example:

```
Lfault_v := new string'(L_v.all);  -- make copy
Writeline(LogFile_f, L_v);   -- write to log
Writeline(ErrFile_f, Lfault_v); -- write to error file
```

6.5.4 Top level Testbench

The top level testbench is in file *tb/uart8_tb.vhd* and is shown on page 192. It instantiates the UART model, CPU client, UART server, Receiver client, receiver server, and the verifier.

6.5.5 Configuration

The configuration files for the various models is in file *uart_c.vhd* and is shown on page 198. The values for the UART generics and scenario control file paths can be defined with configuration declarations.

Verifpeek 1/16

Vpk 2/16

Vpk 3/16

Vpk 4/16

Vpk 5/16

Vpk 6/16

Vpk 7/16

Vpk 8/16

Vpk 9/16

Vpk 10/16

Vpk 11/16

Vpk 12/16

Vpk 13/16

Vpk 14/16

Vpk 15/16

Vpk 16/16

Uart8tb 1/6

Uart8tb 2/6

Uart8tb 3/6

Uart8tb 4/6

Uart8tb 5/6

Uart8tb 6/6

Config
uart_c 1/4

Uart_c 2/4

Uart_c ¾

Uart 4/4

6.5.6 Definition of Scenarios (test cases)

The transaction tests are defined in the testplan. Two methods are used to generate those transactions:

1. **Command files** as called by the client model with the *ExecuteControlFile* procedure. For example:

```
ExecuteControlFile (
       FileName_c   => "path/commandfile.txt",
       Task         => Task,
       ServerData   => ServerData,
       EOT          => EOT,
       Clk          => ClkSys,
       Task_v       => Task_v,
       LfsrData_v   => LfsrData_v);
```

2. **VHDL code** from within the same client model. This code may precede or follow the command file procedure call. The code may also call multiple command files anywhere within the sequence.

The commanded sequence is defined in files for the sequential tests, and in VHDL code in the client models for the pseudo-random tests. Two files are used for the sequential tests, one for the CPU client, and one for the receive side of the model (i.e., *Rxd)*. Each of those file refer to subroutine files for the command of tests setup under different environments (e.g., parity, no parity). This technique promotes reuse. A *SYNC* instruction in conjunction with *SYNCS* signal of a resolved integer type is used to synchronize the CPU and Receive model. This method was explained in section 3.2.3. The scenario command files are listed in Table 6.5.6.

Table 6.5.6 Scenario Command Files

File	Figure #	Page	Function
instr1.txt	**6.5.6-1**	**202**	**CPU Client main instruction stream**
cpu5to15.txt	**6.5.6-2**	**207**	**Subroutine for CPU client, tests 5 to 15**
sw_reset.txt	**6.5.6-3**	**211**	**Subroutine for CPU client, software reset**
rcvinstr.txt	**6.5.6-4**	**212**	**Receiver client main instruction stream**
rcv11to15.txt	**6.5.6-5**	**214**	**Receiver client subroutine for tests 11 to 15**

```
--    (WRITE1,  -- Write a single word @ address with data
--     RNDM_DATA,  -- Write a single word @ address with ran
--     READ1,   -- READ a single word @address
--     IDLE,    -- Stay in IDLE (no load mode) "IDLE 5  -- 5 cycles"
--     RESET,   -- hardware reset for "n" cycles
--     DISP,    -- Displays a message
--     MODE,    -- Uart TB mode
```

```
--      RDUNTIL,    -- wait for n clocks and Read until a '1' condition is true,
--      ENVSETUP,    -- Sets the hold signal to true/false
--      CALL,       -- jump to subroutine
--      SYNC       -- Sets sync level
--      WT4INTRPT   -- wait 4 interrupt
--      STOP       -- stop sim
-- Address  Read  Write  Function          Comments
-- 00       X     -     Modem status       4 bits: RIN, CTSn, DSRn, DCDn
-- 00       -     X     Modem Control       DTR, Parity Enb, Parity Bit,
--
-- 01       X     -     Rcv PIR           FE, PE, OVE, NMT, AMT, AF, HF, FL
-- 01       -     X     Rcv Fifo control    6 bits: Reset, Rcv Intrpt enb(4..0)
--                        not empty, almost empty, almost full, half-full, full
-- 10       X     -     Xmt PIR           ER, NMT, AMT, AF, HF, FL
-- 10       -     X     Xmt Fifo control    6 bits: Reset, xmt Intrpt enb(4..0)
--                        empty, almost empty, almost full, half-full, full
--
-- 11       -     X     Write Xmt Data      transmit data
-- 11       X     -     Read  Rcv data      Receive data
-- file c:/uart/instr1.txt
-- only the first 4 characters of the instruction are important
--
--=================================================
-- Test # 1 Setup @elaboration, no tests
DISPLAY -- CPU --  file c:/uart/instr1.txt
-- Fixed Parameterization- Word size- Buffer Depth- Buffer Almost Empty
-- Buffer Almost Full- Synchronous/    Asynchronous Mode
-- Instantiation transmit function- Instantiation receive function
--=================================================
IDLE 10
DISPLAY
================================================================
DISPLAY  -- CPU -- Test #2 Reset
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
SYNC -100 -- low number, below the receiver driver number
RESET  10  -- reset for 10 cycles

READ 01 -- RCV Buffer, DO(7..0) = 00000000
READ 10 -- XMT  Buffer, DO(5..0) =  000000
IDLE 10
DISPLAY
================================================================
DISPLAY -- CPU -- Test #3 Modem Status
-- I/O BFM to Toggle modem status bits: RINn CTSn DSRn DCDn CPU to read data
-- Set RINn CTSn DSRn DCDn  =  0000
```

```
ENVSETUP  00
READ 00  -- Modem status
-- Set RINn CTSn DSRn  DCDn  =  0001
ENVSETUP  01
READ 00 -- Modem status

--Set RINn CTSn DSRn  DCDn  =  0010
ENVSETUP  02
READ 00 -- Modem status

-- Set RINn CTSn DSRn  DCDn  =  0100
ENVSETUP  04
READ 00 -- Modem status

--Set RINn CTSn DSRn  DCDn  =  1000
ENVSETUP  08
READ 00 -- Modem status

--Set RINn CTSn DSRn  DCDn  =  1111
ENVSETUP  0F
READ 00 -- Modem status
IDLE 10
DISPLAY
================================================================
DISPLAY -- CPU -- Test #4 Modem ControlCPU to Toggle DTRn Set no parity mode
WRITE 00 00 -- DTR = 0
IDLE 4
WRITE 00 04 -- NO parity DTRn = 1
IDLE 4
WRITE 00 00 -- DTR = 0
IDLE 10
DISPLAY -- CPU --  **** ********** CPU 5 to 15 **************************
CALL c:/uart/cpu5to15.txt  -- IN @ set SYNC, then 0 to 9, out @ -100
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU --  Test #16 Parity bit, ODD
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
IDLE 10
DISPLAY -- CPU --  **** ********** CPU 5 to 15 **************************
CALL c:/uart/cpu5to15.txt -- IN @ set SYNC, then 0 to 9, out @ -100
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU --  Test 17 Parity bit, EVEN
WRITE 00 02 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
IDLE 10
```

```
DISPLAY -- CPU --  **** ********** CPU 5 to 15 **************************
CALL c:/uart/cpu5to15.txt -- IN @ set SYNC, then 0 to 9, out @ -100
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU --  TEST 18 Receive framing error, Even Parity
IDLE 100
SYNC 9  -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 100
SYNC 10 -- wait till end of RCV BFM
READ 01 -- rcv fifo status
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
IDLE 100
DISPLAY  -- CPU -- --> end of test 18
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU -- TEST 19 Receive parity error, Even Parity
WRITE 00 02 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
SYNC 11  -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 100
SYNC 12 -- wait till end of RCV BFM
READ 01 -- rcv fifo status
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
IDLE 100
DISPLAY  -- CPU -- --> end of test 19
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU -- TEST 20 Receive buffer overrun error, even Parity
WRITE 00 02 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
SYNC 13 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 100
SYNC 14 -- wait till end of RCV BFM
READ 01 -- rcv fifo status
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
IDLE 100
DISPLAY  -- CPU -- --> end of test 19
CALL c:/uart/sw_reset.txt
```

```
DISPLAY
================================================================
DISPLAY -- CPU -- TEST 21, Transmit buffer overrun error, even Parity
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 100
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 100
READ 01 -- rcv fifo status
DISPLAY -- CPU -- End of test 20
CALL c:/uart/sw_reset.txt
IDLE 4
DISPLAY
================================================================
DISPLAY -- CPU -- Test 22 Receive framing error, Odd  Parity
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
SYNC 15 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 100
SYNC 16 -- wait till end of RCV BFM
READ 01 -- rcv fifo status
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
IDLE 100
DISPLAY  -- CPU -- --> end of test 19
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU -- Test 23 Receive parity error, Odd  Parity
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
SYNC 17  -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 100
SYNC 18 -- wait till end of RCV BFM
READ 01 -- rcv fifo status
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
IDLE 100
DISPLAY  -- CPU -- --> end of test 19
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU -- Test 24 Receive buffer overrun error, Odd  Parity
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
```

```
SYNC 19
IDLE 100
SYNC 20
READ 01 -- rcv fifo status
DISPLAY -- CPU -- end of test 24
CALL c:/uart/sw_reset.txt
DISPLAY
================================================================
DISPLAY -- CPU -- Test 25 Transmit buffer overrun error, Odd  Parity
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 100
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 100
READ 10 -- xmt fifo status
DISPLAY -- CPU -- End of test 24
CALL c:/uart/sw_reset.txt
SYNC 21
DISPLAY -- CPU -- End of tests
-- STOP
```

Figure 6.5.6-1 CPU Client Main Instruction Stream (file instr1.txt)

```
DISPLAY -- file c:/uart/cpu5to15.txt
DISPLAY Test # 5 Transmit protocol
-- CPU writes 1 word into buffer,
-- Set RINn CTSn DSRn DCDn  =  0111
ENVSETUP 07
-- Xmt Fifo control    6 bits: Reset, MT, AE, HF, AF, FL
WRITE 10 1F -- Xmt buffer setup
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles

ENVSETUP 03 -- Set RINn CTSn DSRn DCDn  =  0011
IDLE 51 -- Wait for 51 cycles
ENVSETUP 01 -- Set RINn CTSn DSRn DCDn  =  0001
IDLE 51 -- Wait for 51 cycles
ENVSETUP 00 -- Set RINn CTSn DSRn DCDn  =  0000
-- Wait for interrupt within serial   transmission time  for one message
-- Read @ADDR (in bit)  MASK (in Hex) Interval (in natural) until the received masked
data has a ONE.
IDLE 51 -- wait till start of send
RDUNTIL 10 10 500 -- Read data from address 10, MASK = X"10, if false, wait for
                -- 500 cycles
```

```
DISPLAY    --> Test 5, XMT PIR EMpty reached
IDLE 8000 -- 10 bits * 16 16x_clk/bit * ~10 sysclk/16x_clk  + spare -- end of serial message
DISPLAY
============================================================
DISPLAY Test # 6 Transmit protocol CPU writes "n" words into buffer, interrupt on empty (MT)
-- Read 10
-- read/clr xmt buffer status
WRITE 10 10 -- interrupt when MT
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 5 -- Wait for 5 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 1 -- Wait for 51 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles
WT4INTRPT 10 5000  -- wait 4 interrupt
RDUNTIL 10 10 200 -- Read data from address 01, MASK = X"10, if false, wait for
            -- 20 cycles
DISPLAY    --> Test 6, XMT PIR Emptied reached
IDLE 9000 -- end of serial message

DISPLAY
============================================================
DISPLAY Test #7 Transmit protocol CPU writes "n" words into buffer,interrupt on Almost empty
WRITE 10 08  -- interrupt when MT
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 5 -- Wait for 5 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 1 -- Wait for 51 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles
WT4INTRPT 10 5000
RDUNTIL 10 08 500 -- Read data from address 01, MASK = X"10, if false, wait for
            -- 20 cycles
DISPLAY    --> Test 7, XMT PIR Almost Emptied reached
IDLE 9000 -- 10 bits * 16 clk/bit * ~10 clk/16x_clk -- end of serial message

DISPLAY
============================================================
DISPLAY Test #8 Transmit protocol CPU writes "n" words into buffer,interrupt on half-full
WRITE 10 04  -- interrupt when MT
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 5 -- Wait for 5 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
```

```
IDLE 1 -- Wait for 51 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles
WT4INTRPT 10 5000
RDUNTIL 10 04 500 -- Read data from address 01, MASK = X"10, if false, wait for
            -- 500 cycles
DISPLAY    --> Test 8, XMT PIR Almost Emptied reached
IDLE 9000 -- 10 bits * 16 clk/bit * ~10 clk/16x_clk -- end of serial message

DISPLAY
================================================================
DISPLAY Test #9 Transmit protocol CPU writes "n" words into buffer,interrupt on almost-full
WRITE 10 02  -- interrupt when MT
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 5 -- Wait for 5 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 1 -- Wait for 51 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles
WT4INTRPT 10 5000
RDUNTIL 10 02 20 -- Read data from address 01, MASK = X"10, if false, wait for
            -- 20 cycles
DISPLAY    --> Test 9, XMT PIR Almost Emptied reached
IDLE 8000 -- 10 bits * 16 clk/bit * ~10 clk/16x_clk -- end of serial message

DISPLAY
================================================================
DISPLAY Test #10 Transmit protocol CPU writes "n" words into buffer,interrupt on full
WRITE 10 01  -- interrupt when MT
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 5 -- Wait for 5 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 1 -- Wait for 51 cycles
RNDM_DATA 11 --WRITE 11 RandomData, fill xmt buffer
IDLE 51 -- Wait for 51 cycles
RDUNTIL 10 01 500 -- Read data from address 01, MASK = X"10, if false, wait for
            -- 500 cycles
DISPLAY    --> Test 10, XMT PIR Almost Emptied reached
IDLE 10000 -- 10 bits * 16 16x_clk/bit * ~10 clk/16x_clk -- end of serial message

-- TEST 11 Receive protocol  interrupt on not empty
-- setup control registers, and then enable te receiver to fire
DISPLAY
================================================================
DISPLAY Test 11, Receive protocol  interrupt on not empty
WRITE 01 10 -- interrupt on recvr not MT
```

```
SYNC 0 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 1
SYNC 1 -- wait till receiver tests are done --> Wait till RCV BFM DONE
READ 01 -- RCV Buffer,
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data

DISPLAY
================================================================
DISPLAY Test 12, Receive protocol  interrupt on almost-empty
WRITE 01 08 -- interrupt on recvr  AMT
SYNC 2 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 1
SYNC 3 -- wait till receiver tests are done--> Wait till RCV BFM DONE
READ 01 -- RCV Buffer,
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data

IDLE 2000
DISPLAY
================================================================
DISPLAY Test 13, Receive protocol  interrupt on half-full
WRITE 01 04 -- interrupt on recvr  AMT
SYNC 4 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 1
SYNC 5 -- wait till receiver tests are done--> Wait till RCV BFM DONE
READ 01 -- RCV Buffer,
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data

DISPLAY
================================================================
DISPLAY Test 14, Receive protocol  interrupt on almost-full
WRITE 01 02 -- interrupt on recvr  AMT
SYNC 6 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 1
SYNC 7 -- wait till receiver tests are done--> Wait till RCV BFM DONE
READ 01 -- RCV Buffer,
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
```

```
DISPLAY
================================================================
DISPLAY Test 15, Receive protocol  interrupt on full
WRITE 01 01 -- interrupt on recvr  AMT
SYNC 8 -- enable rcv bfm to send RXD data --> GO RCV BFM
IDLE 1
SYNC 9 -- wait till receiver tests are done--> Wait till RCV BFM DONE
READ 01 -- RCV Buffer,
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data
READ 11 -- Read received FIFO data

SYNC -100 -- take control of syncs
DISPLAY --> CPU driver reached end of tests
DISPLAY
================================================================
DISPLAY
================================================================
```

Figure 6.5.5-2 Subroutine for CPU client, tests 5 to 15 (file cpu5to15.txt)

```
-- File sw_reset.txt
-- SYNC -100 -- low number, below the receiver driver number
-- WRITE 10 40 -- reset XMT portion
-- IDLE 4
-- WRITE 01 40 -- reset receiver portion
-- IDLE 4
-- WRITE 10 0F -- Enable xmt interrupts
-- IDLE 4
-- WRITE 01 0F -- Enable rcv interrupts
IDLE 4
```

Figure 6.5.5-3 Subroutine for CPU Client, Software Reset (file sw_reset.txt) (Instructions commented out, and not used in tests, demonstrates possibilities)

```
-- file c:/uart/rcvinstr.txt
-- only the first 4 characters of the instruction are important
-----------------------------------------------------
-- file c:/uart/rcvinstr.txt
--=================================================
DISPLAY -- RCV --  RCV for test 5 thru 15
WRITE 00 00 -- DTRn = 0
DISPLAY    --    RCV    --        --    <<<<<<<<<<<        RCV    11    to    15
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
CALL  c:/uart/rcv11to15.txt --  In @ SYNC 0, exits @ SYNC 9
--=================================================
IDLE 200
DISPLAY -- RCV --  -- Test 16 Parity bit, ODD
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
IDLE 4
DISPLAY    --    RCV    --        --    <<<<<<<<<<<        RCV    11    to    15
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
CALL  c:/uart/rcv11to15.txt --  In @ SYNC 0, exits @ SYNC 9
--=================================================
DISPLAY -- RCV --  -- Test 17 Parity bit, EVEN
WRITE 00 02 -- DTRn = 0  DTR, Parity Enb, Parity Bit,
IDLE 10
DISPLAY    --    RCV    --        --    <<<<<<<<<<<        RCV    11    to    15
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
CALL  c:/uart/rcv11to15.txt --  In @ SYNC 0, exits @ SYNC 9
--=================================================
DISPLAY -- RCV --  -- TEST 18 Receive framing error, Even Parity
IDLE 100
MODE FRAME_ERR  -- framing error mode
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 10 -- Control back to CPU bfm
IDLE 10
SYNC 11 -- wait enable from cpu bfm
DISPLAY -- RCV --  --End of test 18
--=================================================
DISPLAY -- RCV --   -- TEST 19 Receive parity error, Even Parity
MODE NORMAL
IDLE 1
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,, wrong parity bit
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
```

```
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 12 -- Control back to CPU bfm
DISPLAY -- RCV --  --End of test 19
--=================================================
DISPLAY -- RCV --    TEST 20 Receive buffer overrun error, even Parity
MODE NORMAL
IDLE 1
SYNC 13 -- wait enable from cpu bfm
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,, wrong parity bit
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here -- overrun error
IDLE 10000
SYNC 14 -- Control back to CPU bfm
DISPLAY -- RCV --  --End of test 19
--=================================================
SYNC 15 -- RCV -- Test 22  Receive framing error, Odd  Parity
WRITE 00 01 -- DTRn = 0  DTR, Parity Enb, Parity Bit,       --
IDLE 100
MODE FRAME_ERR  -- framing error mode
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 16 -- Control back to CPU bfm
DISPLAY -- RCV -- end of test 22
--=================================================
DISPLAY -- RCV -- Test 23 Receive parity error, Odd  Parity
MODE NORMAL
IDLE 1
SYNC 17 -- wait enable from cpu bfm
WRITE 00 02 -- DTRn = 0  DTR, Parity Enb, Parity Bit,, wrong parity bit
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 18 -- Control back to CPU bfm
DISPLAY -- RCV --  --End of test 23
```

```
--=================================================
DISPLAY -- RCV --    TEST 24 Receive buffer overrun error, Odd Parity
MODE NORMAL
IDLE 1
SYNC 19 -- wait enable from cpu bfm
WRITE 00 03 -- DTRn = 0  DTR, Parity Enb, Parity Bit,,
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here -- overrun error
IDLE 10000
SYNC 20 -- Control back to CPU bfm
DISPLAY -- RCV --  --End of test 24
--=================================================
SYNC 21
DISPLAY -- RCV --  --End of tests
```

Figure 6.5.5-4 Receiver Client Main Instruction Stream (file rcvinstr.txt)

```
-------------------------------------------------
-- file c:/uart/rcv11to15instr.txt
DISPLAY
================================================================
DISPLAY
================================================================
DISPLAY  TEST 11 Receive protocol  interrupt on not empty
IDLE 10
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
SYNC 0 -- --> wait till GO RCV BFM
DISPLAY
================================================================
DISPLAY Test #11 RCV BFM, send 4 data word to RXD
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 1  -- end of test 11  --> RCV BFM DONE
IDLE 1  --

SYNC 2  -- let CPU setup for test 12, --> wait till GO RCV BFM
DISPLAY
================================================================
```

```
DISPLAY  TEST 12 Receive protocol  interrupt on AMT
IDLE 1000
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
DISPLAY Test #12 RCV BFM, send 4 data word to RXD
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 3  -- end of test 12 --> RCV BFM DONE
IDLE 1  --
SYNC 4  -- let CPU setup for test 12, --> wait till GO RCV BFM
DISPLAY
================================================================
DISPLAY  TEST 13 Receive protocol  interrupt on half-full
IDLE 1000
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
DISPLAY Test #13 RCV BFM, send 4 data word to RXD
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 5  -- end of test 13 --> RCV BFM DONE
IDLE 1  --
SYNC 6  -- let CPU setup for test 12, --> wait till GO RCV BFM
DISPLAY
================================================================
DISPLAY  TEST 14 Receive protocol  interrupt on almost-full
IDLE 1000
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
DISPLAY Test #14 RCV BFM, send 4 data word to RXD
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 7  -- end of test 14 --> RCV BFM DONE
```

```
IDLE 1  --

SYNC 8  -- let CPU setup for test 12, --> wait till GO RCV BFM
DISPLAY
================================================================
DISPLAY  TEST 15 Receive protocol  interrupt on full
IDLE 1000
-- RESETVC to be in idle state, all software visible registers to be reset,
-- no interrupt outputs
MODE NORMAL -- NORMAL, FRAME_ERR, PARITY_ERRx
DISPLAY Test #15 RCV BFM, send 4 data word to RXD
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
RNDM_DATA 11 -- 11 is not used here
IDLE 100
RNDM_DATA 11 -- 11 is not used here
IDLE 10000
SYNC 9  -- end of test 15 --> RCV BFM DONE
IDLE 1  --
--================================================
--SYNC 4 -- Let cpu setup for test 13
DISPLAY --> end of RCV rcv11to15 subroutine
DISPLAY
================================================================
DISPLAY
================================================================
```

Figure 6.5.5-5 Receiver Client Subroutine for Tests 11 to 15 (file rcv11to15.txt)

6.5.7 Compilation Scripts

The compilation scripts for the complete design and verification models is shown in Table 6.5.7-1, as defined in file *scripts*/*compile.do.*

Table 6.5.7 compilation scripts, *scripts*/*compile.do*

SCRPIT	FUNCTION
rm -r work_lib **vlib work_lib**	**Remove old library** **Create library**
# RTL Design **vcom -explicit -work work_lib -93 vhdl/tb/Size_Pkg.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/fifo.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/transmitter.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/xmitsublk.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/receiver.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/rcvsublk.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/cpuif.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/clkcntrl.vhd** **vcom -explicit -work work_lib -93 vhdL/rtl/uart.vhd**	 **Size package** **FIFO** **Transmitter** **Transmit subblock** **Receiver** **Receiver suibblock** **CPU interface** **Clock controller** **UART top level**
# TESTBENCH **vcom -explicit -work work_lib -93 vhdl/tb/image_pb.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/vsp.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/lfsrstd.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/parser_pb.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/uart_clientrndm.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/uart_client_bad.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/uart_server.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/rcv_client.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/rcv_server.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/fifo_server.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/fifo_tb.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/verifierpeek.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/verifierblkbox.vhd** **vcom -explicit -work work_lib -93 vhdl/tb/uart8_tb.vhd** **vcom -explicit -work work_lib -93 vhdl/gates/uart.vho** **vcom -explicit -work work_lib -93 vhdl/tb/uart_c.vhd** **vcom -explicit -work work_lib -93 vhdl/rtl/uart_level2.vhd**	 **Image package** **VSP package** **LFSR package** **Parser package** **UART client, works OK** **UART Client, overrun errors** **UART server** **Receive client (for *RxD*)** **Receive server** **Fifo Server for FIFO only tests** **Fifo testbench for FIFO only** **Verifier with peek (gray box)** **Verifier black box** **UART testbench, set for 8 bits** **Gate level model by Altera** **Configuration file** **UART into a subsystem**

Four configurations are defined in the *uart_c.vhd* configuration file as shown in Table 6.5.7-2.

Table 6.5.7-2 Configurations

Configuration Name	Purpose
UART_RTL_Rndm_Grey_cfg	UART @RTL Gray Box verification model. Random tests limited to CPU WRITE of transmit data from ONE to THREE data words, randomly selected. CPU waits until completion of serial transmission before repeating sequence.
UART_RTL_BlackBox_cfg	UART @RTL Black Box verification model. Random tests limited to ONE CPU WRITE of transmit data. CPU waits until completion of serial transmission before repeating sequence.
UART_Gates_cfg	UART @ Gate level, VHDL produced by Altera. Black Box verification model. Random tests limited to ONE CPU WRITE of transmit data. CPU waits until completion of serial transmission before repeating sequence.
UART_RTL_Grey_Bad_cfg	UART @RTL. Gray Box verification model. Random tests with transmission-overrun errors.

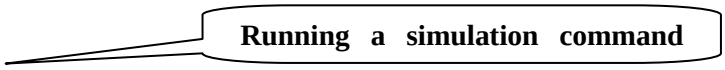
6.5.8 Simulation Results

The UART model was simulated with ModelSim 5.4b with the statement coverage option. The results of this simulation effort are presented and discussed. Users of other simulation tools would use the relevant commands for those tools. However, this presentation will provide a basis for the discussion of results.

6.5.8.1 Running the Simulator

The *ModelSim* simulator can be initiated with the *vsim* command. Since configuration declarations were defined, the firing of *uart_rtl_grey_cfg* configuration and start of simulation is shown in Figure 6.5.8.1. Only a portion of the transcript window is included here. The complete set of simulation result files are included in subdirectory *SimRuns.* The simulator was run for a simulation period of forty milliseconds.

```
cd c:/design_path
# reading modelsim.ini
do run.do
# vsim -coverage work_lib.uart_rtl_grey_rndm_cfg
# Loading C:/MODELTECH_5.4B/WIN32/../std.standard
# Loading C:/MODELTECH_5.4B/WIN32/../ieee.std_logic_1164(body)
# Loading C:/MODELTECH_5.4B/WIN32/../ieee.std_logic_arith(body)
# Loading C:/MODELTECH_5.4B/WIN32/../ieee.std_logic_unsigned(body)
# Loading C:/MODELTECH_5.4B/WIN32/../std.textio(body)
```



```
# Loading C:/MODELTECH_5.4B/WIN32/../ieee.std_logic_textio(body)
# Loading C:/MODELTECH_5.4B/WIN32/../synopsys.attributes
# Loading C:/MODELTECH_5.4B/WIN32/../ieee.std_logic_misc(body)
# Loading work_lib.vsp(body)
# Loading work_lib.image_pkg(body)
# Loading work_lib.lfsrstd_pkg(body)
# Loading work_lib.size_pkg
# Loading work_lib.parser_pkg(body)
# Loading work_lib.uart_rtl_grey_cfg
# Loading work_lib.uart8_tb(beh)
# Loading work_lib.uart_client(uart_clientrndm)
# Loading work_lib.uart_server(beh)
# Loading work_lib.clkcntrl(rtl)
# Loading work_lib.uart(rtl)
# Loading work_lib.xmitsublk(rtl)
# Loading work_lib.transmitter(rtl)
# Loading work_lib.fifo(rtl)
# Loading work_lib.rcvrsublk(rtl)
# Loading work_lib.receiver(rtl)
# Loading work_lib.cpuif(rtl)
# Loading work_lib.rcv_client(rcv_client_a)
# Loading work_lib.rcv_server(beh)
# Loading work_lib.verifier(verifierpeek)
view signals
# .signals
add wave -r /*
run 40 ms
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /uart8_tb/uart_1/xmitsublk_1/transmitter_1
# ** Note: New control file: c:/uart/rcvinstr.txt
#    Time: 40 ns  Iteration: 0  Instance: /uart8_tb/rcv_client_1
# ** Note: DISP:  -- RCV --  RCV for test 5 thru 15
#    Time: 40 ns  Iteration: 0  Instance: /uart8_tb/rcv_client_1
# ** Note: New control file: c:/uart/instr1.txt
#    Time: 40 ns  Iteration: 0  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- CPU --  file c:/uart/instr1.txt
#    Time: 40 ns  Iteration: 0  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- RCV --  -- <<<<<<<<<<<  RCV 11 to 15
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
#    Time: 200 ns  Iteration: 1  Instance: /uart8_tb/rcv_client_1
# ** Note: Subroutine to: c:/uart/rcv11to15.txt
#    Time: 280 ns  Iteration: 1  Instance: /uart8_tb/rcv_client_1
# ** Note: DISP:
============================================================
=
#    Time: 360 ns  Iteration: 0  Instance: /uart8_tb/rcv_client_1
# ** Note: DISP:
```

Simulator loading the design units

Simulator running and displaying results

```
================================================================
=
#    Time: 440 ns  Iteration: 1  Instance: /uart8_tb/rcv_client_1
# ** Note: DISP:  TEST 11 Receive protocol  interrupt on not empty
#    Time: 520 ns  Iteration: 1  Instance: /uart8_tb/rcv_client_1
# ** Note: DISP:
================================================================
=
#    Time: 560 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- CPU -- Test #2 Reset
#    Time: 640 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 2280 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- CPU -- Test #3 Modem Status
#    Time: 2360 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 3840 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- CPU -- Test #4 Modem ControlCPU to Toggle DTRn Set no parity
mode
#    Time: 3920 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- CPU --  **** ********** CPU 5 to 15 **************************
#    Time: 5200 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: Subroutine to: c:/uart/cpu5to15.txt
#    Time: 5280 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 5360 ns  Iteration: 0  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 5440 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  -- file c:/uart/cpu5to15.txt
#    Time: 5520 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  Test # 5 Transmit protocol
#    Time: 5600 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  --> Test 5, XMT PIR EMpty reached
#    Time: 34600 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 354720 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  Test # 6 Transmit protocol CPU writes "n" words into buffer, interrupt on empty (MT)
```

```
#    Time: 354800 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  --> Test 6, XMT PIR Emptied reached
#    Time: 555560 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
#    Time: 915680 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  Test #7 Transmit protocol CPU writes "n" words into buffer,interrupt on Almost empty
#    Time: 915760 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:  --> Test 7, XMT PIR Almost Emptied reached
#    Time: 1115920 ns  Iteration: 1  Instance: /uart8_tb/uart_client_1
# ** Note: DISP:
================================================================
=
```

Figure 6.5.8.1 Running the Simulation (only the first 1.1 ms is shown)

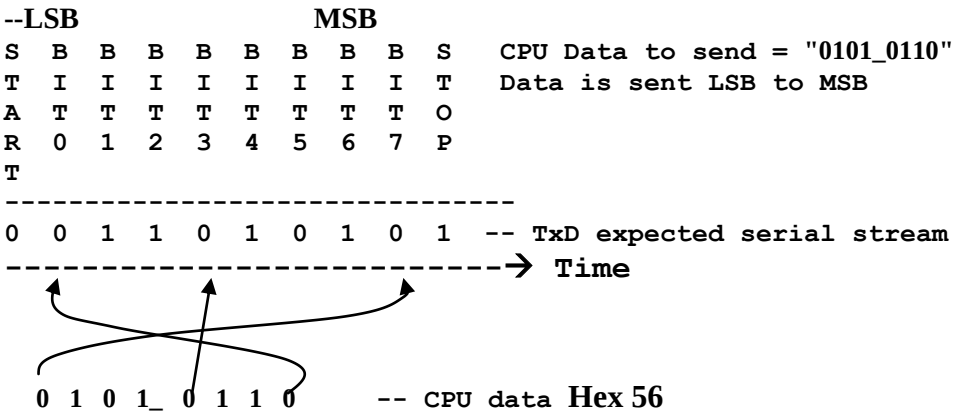
6.5.8.2 Evaluating Results

The waveform viewer is usually the best tool to debug a design, but the process is various tedious because transactions must be extracted from the multitude of signals (e.g., *Rdn, CS0, CS1, CS2n, Addr*) . In addition, the users are subjected to fatigue and error. In this design, the verifier extracts the transactions and asserts the tasked transactions and user defined messages to files and to the transcript display to ease the debugging process. Note that the generation of this report log is somewhat loading the simulator because of the use of TextIO. However, this reporting feature could be turned off with a generic.

Figure 6.5.8.2-1 is a report log of a CPU tasking the UART to send DATA (value = Hex 56) over the *TxD* port. All Interrupts are enabled. The generics were set as follows:

1. Bus width = 8
2. Transmit FIFO depth = 4
3. Full = 4, Almost full = 3, Half-full = 2, Almost empty = 1, Empty = 0

Figure 6.5.8.2-2 is a waveform view for the same timing period as the report view generated by the verifier. Note that the transmitted data to be sent is Hex 56, or binary "0101_0110" with parity disabled. Thus, the transmitted stream on the *TxD* signal will be:



This output, and the generation of the *Interrupt* signal is demonstrated in these two figures. The pending interrupt register (*PIR*) has a value of 00011000, meaning that the transmit model got to the empty state (from a FIFO having a count value of ONE and then to ZERO). In addition, the transmit model got OFF the almost-empty state (i.e., had reached the almost-empty, and is now OFF that level).

```
5520 ns   ====================================================
4720 ns   ** CPU WRITE @ 00 MODEM & PARITY: DTR, Parity Enb, Parity Bit =
00000000
4720 ns   Tasked: DTRn & Parity Ctrl = 1  00
5280 ns   -- CPU --  **** ********** CPU 5 to 15 **********
5360 ns   CPU Subroutine to: c:/uart/cpu5to15.txt
5440 ns   ====================================================
5520 ns   ====================================================
5600 ns   -- file c:/uart/cpu5to15.txt
5680 ns   Test # 5 Transmit protocol
5720 ns   Tasked: ENVSETUP RIn CTSn DSRn DCDn = 0111
5800 ns   ** CPU WRITE @ 10 XMT INTRPT: Reset, Xmt Intrpt enb(4..0) = 00011111
5800 ns   Tasked: Xmt reset & Xmt intrpt env = 111
5920 ns   ** CPU WRITE @ 11 XMT DATA = X_56
5920 ns   Tasked: Write Xmt data = X_56
8080 ns   Tasked: ENVSETUP RIn CTSn DSRn DCDn = 0011
10200 ns  Tasked: ENVSETUP RIn CTSn DSRn DCDn = 0001
12320 ns  Tasked: ENVSETUP RIn CTSn DSRn DCDn = 0000
14480 ns  ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00000000
20520 ns  Interrupt line = 10
34560 ns  ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00011000
34560 ns  Interrupt line = 00
34680 ns  --> Test 5, XMT PIR EMpty reached
```

Parity is Disabled

Enable Interrupts

CPU Write DATA for Xmit

Interrupt Occurred. Read PIR

Figure 6.5.8.2-1 Report Log of a CPU tasking the UART to send DATA (Generated by the Verifier Model)

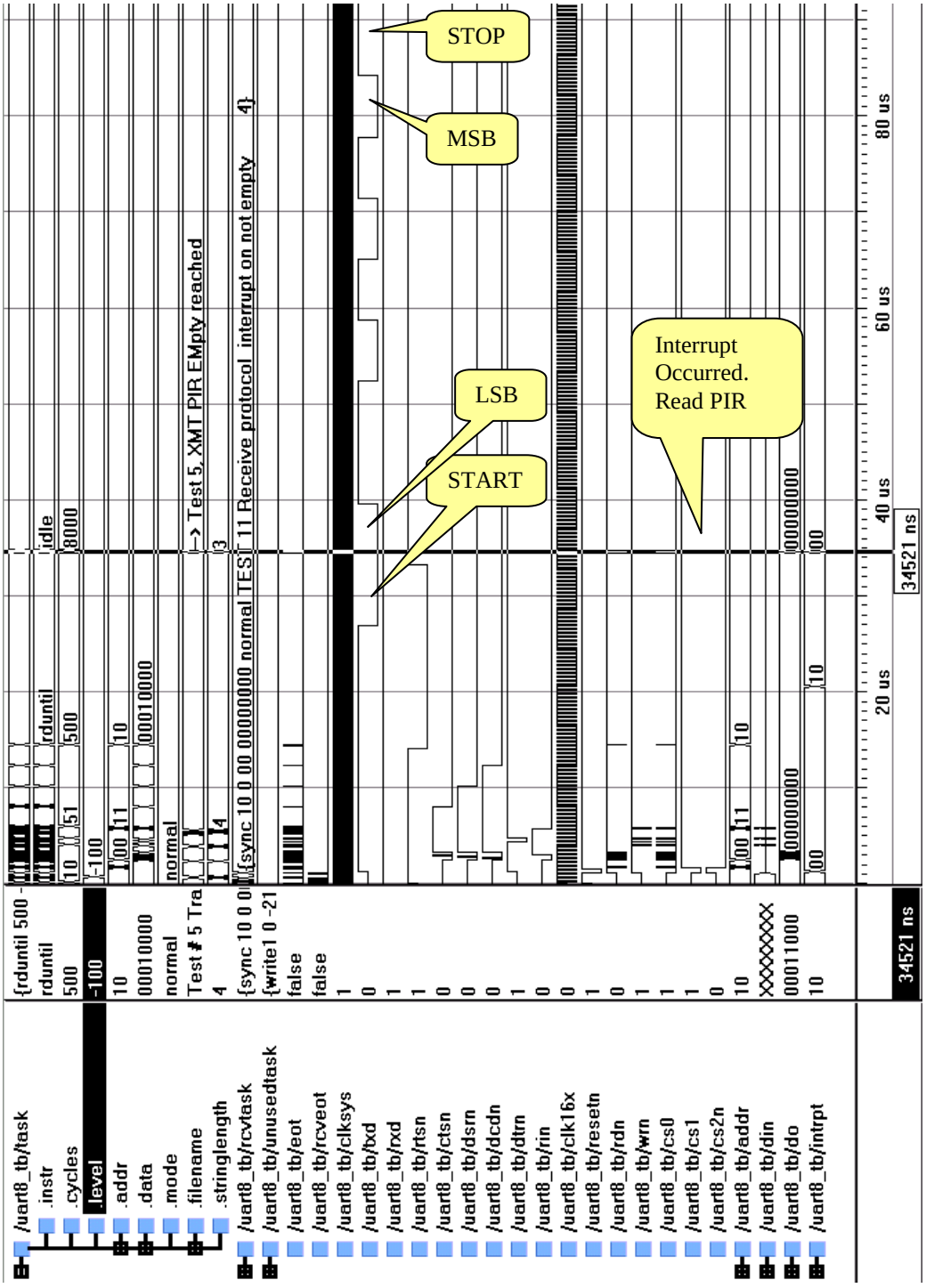
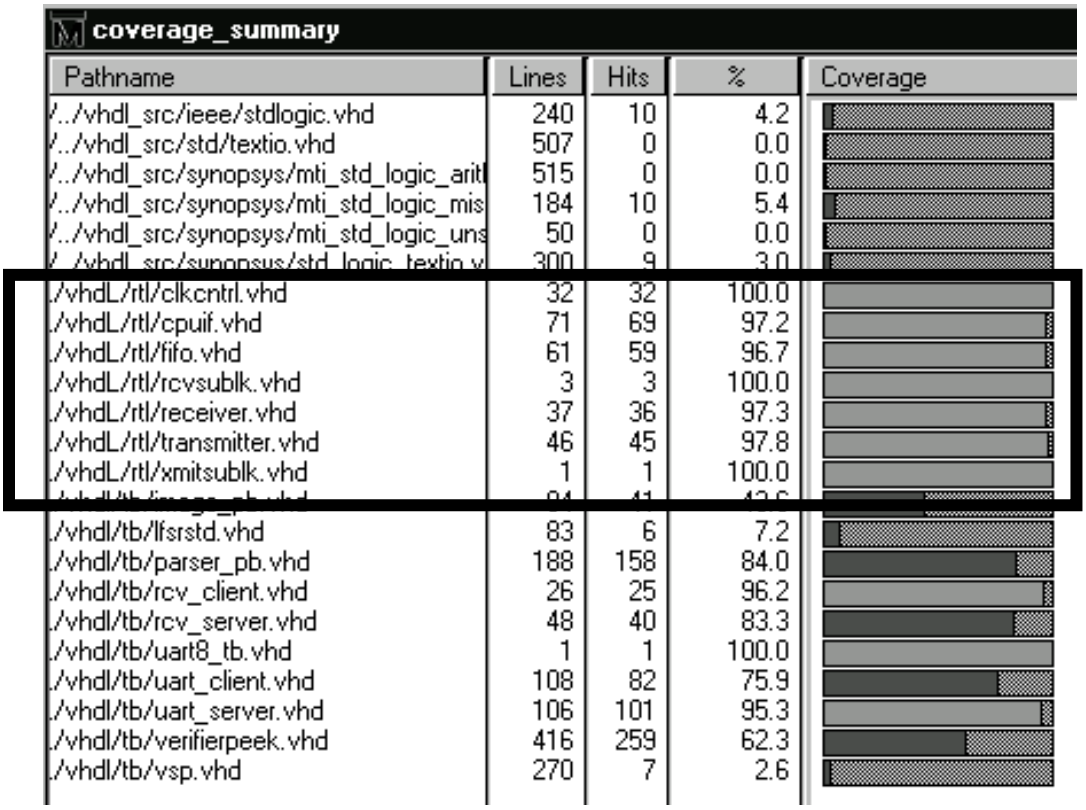


Figure 6.5.7.2-1 Waveform View of a CPU tasking the UART to send DATA (Generated by *ModelSim*)

ModelSim provides a built-in statement code coverage feature that provides graphical and report file feedback on how the source code is being executed. It allows line number execution statistics to be kept by the simulator. Note that statement code coverage is only one of the many coverage features that can be measured. For example, TransEDA *Verification Navigator*[4] provides the following additional coverage metrics, including branch, condition and expression, path, toggle, triggering, and signal-tracing coverages[5]. Figure 6.5.8.2-2 is a copy of the graphical view of the statement coverage. Figure 6.5.8.2-3 demonstrates areas of the FIFO code where statements were never visited during the simulation. That was the case of a simultaneous *push* and *pop* operation.



coverage_summary

Pathname	Lines	Hits	%	Coverage
/../vhdl_src/ieee/stdlogic.vhd	240	10	4.2	
/../vhdl_src/std/textio.vhd	507	0	0.0	
/../vhdl_src/synopsys/mti_std_logic_arit	515	0	0.0	
/../vhdl_src/synopsys/mti_std_logic_mis	184	10	5.4	
/../vhdl_src/synopsys/mti_std_logic_uns	50	0	0.0	
/../vhdl_src/synopsys/std_logic_textio.v	300	9	3.0	
./vhdL/rtl/clkcntrl.vhd	32	32	100.0	
./vhdL/rtl/cpuif.vhd	71	69	97.2	
./vhdL/rtl/fifo.vhd	61	59	96.7	
./vhdL/rtl/rcvsublk.vhd	3	3	100.0	
./vhdL/rtl/receiver.vhd	37	36	97.3	
./vhdL/rtl/transmitter.vhd	46	45	97.8	
./vhdL/rtl/xmitsublk.vhd	1	1	100.0	
./vhdl/tb/image_pb.vhd	84	41	48.8	
./vhdl/tb/lfsrstd.vhd	83	6	7.2	
./vhdl/tb/parser_pb.vhd	188	158	84.0	
./vhdl/tb/rcv_client.vhd	26	25	96.2	
./vhdl/tb/rcv_server.vhd	48	40	83.3	
./vhdl/tb/uart8_tb.vhd	1	1	100.0	
./vhdl/tb/uart_client.vhd	108	82	75.9	
./vhdl/tb/uart_server.vhd	106	101	95.3	
./vhdl/tb/verifierpeek.vhd	416	259	62.3	
./vhdl/tb/vsp.vhd	270	7	2.6	

Figure 6.5.8.2-2 Graphical View of the Statement Coverage for the UART Model Simulation

4 http://www.transeda.com

5 See *Verification Methodology Manual for Code Coverage in HDL Designs*, Michael Stuart & David Dempster, isbn 0-9538-4820-5, Teamwork International 2000

```
         .    110
2000000       111    case PushnPopn is
         .    112      when "00" =>
         .    113        -- no change to pointers or st
         0    114        FIFO_r(WrPntr_r) <= data_in;
         .    115
         .    116      when "01" =>
       554    117        if Count_r = Depth_g then
        10    118          ErrorT_r <= '1';
         .    119        else
       544    120          FIFO_r(WrPntr_r) <= data_in;
       544    121          WrPntr_r         <= (WrPntr_
         .    122          -- right argument must evalu
       544    123          if Count_r /= Depth_g then
       544    124            Count_r <= Count_r +  ;
         .    125          end if;
         .    126        end if;
         .    127
         .    128      when "10" =>
       538    129        if Count_r =   then
         0    130          ErrorT_r <= '1';
         .    131        else
       538    132          RdPntr_r <= (RdPntr_r +  ) r
       538           Count_r  <= (Count_r -  );
```

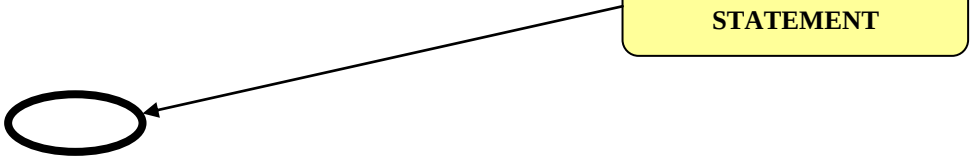


Figure 6.5.7.2-3 Statements of FIFO never visited during the simulation

6.5.8.2.1 Regression Tests, Gray Box RTL versus Black Box RTL

The gray box verification model, with visibility into internal synchronization signals of the RTL model (e.g., the transmit FIFO push and pop), produced results with no reported errors. However, the black box verification model, with a timing estimate of the critical events of the RTL model, produced errors that after analysis were false. This is because the calculated state of the UUT as computed by the verifier is not synchronized (i.e., in error) to the UUT's state. Figure 6.5.8.2.1 represents an example of this false error condition:

Gray Box Log, RTL UART – Good Cycle Synchronization (UUT and Verifier)

354880 ns Test # 6 Transmit protocol CPU writes "n" words into bu
354960 ns ** CPU WRITE @ 10 XMT INTRPT: Reset, Xmt Intrpt enb(4..0) = 00010000
354960 ns Tasked: Xmt reset & Xmt intrpt env = 000
355080 ns ** CPU WRITE @ 11 XMT DATA = X_AB
355080 ns Tasked: Write Xmt data = X_AB
355440 ns ** CPU WRITE @ 11 XMT DATA = X_D5
355440 ns Tasked: Write Xmt data = X_D5
355560 ns ** CPU WRITE @ 11 XMT DATA = X_EA
355560 ns Tasked: Write Xmt data = X_EA
355760 ns ** CPU WRITE @ 11 XMT DATA = X_75
355760 ns Tasked: Write Xmt data = X_75
357920 ns Wait for Interrupt 10
555440 ns Interrupt line = 10
555520 ns ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00011111
555520 ns Interrupt line = 00
555640 ns --> Test 6, XMT PIR Emptied reached

Black Box Log, RTL – Poor Cycle Synchronization (UUT and Verifier)

354880 ns Test # 6 Transmit protocol CPU writes "n" words into bu
354960 ns ** CPU WRITE @ 10 XMT INTRPT: Reset, Xmt Intrpt enb(4..0) = 00010000
354960 ns Tasked: Xmt reset & Xmt intrpt env = 000
355080 ns ** CPU WRITE @ 11 XMT DATA = X_AB
355080 ns Tasked: Write Xmt data = X_AB
355440 ns ** CPU WRITE @ 11 XMT DATA = X_D5
355440 ns Tasked: Write Xmt data = X_D5
355560 ns ** CPU WRITE @ 11 XMT DATA = X_EA
355560 ns Tasked: Write Xmt data = X_EA
355760 ns ** CPU WRITE @ 11 XMT DATA = X_75
355760 ns Tasked: Write Xmt data = X_75
357920 ns Wait for Interrupt 10
555440 ns Interrupt line = 10
555520 ns ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00011111
555520 ns `` XMT PIR Empty Error Observed PIR = 00011111 Expected PIR = 00000111
555520 ns `` XMT PIR almost Empty Error Observed PIR = 00011111 Expected PIR = 00000111
555520 ns Interrupt line = 00
555640 ns --> Test 6, XMT PIR Emptied reached

False error reporting by verifier

Figure 6.5.8.2.1 False Error Reporting because of Lack of Cycle Synchronization between UUT and Verifier

6.5.8.2.2 Regression Tests, Black Box RTL versus Black Box Gate

The gate level model was generated by *Altera* from the *Synplify* generated *EDIF* file. The gate level model matched the performance of the RTL model, with the exception of gate delays. Below is an example of the log reports.

Back Box Gate

```
14480 ns    ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00000000
20541 ns    Interrupt line = 10
34560 ns    ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00011000
34582 ns    Interrupt line = 00
```

Black Box RTL

```
14480 ns    ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00000000
20520 ns    Interrupt line = 10
34560 ns    ** CPU READ @ 10 XMIT PIR: (ERR, MT, AMT, HF, AF, FL) = 00011000
34560 ns    Interrupt line = 00
```

The Gray box gate level could not be easily simulated without code changes to access the internal *push* and *pop* signals of the transmit logic within the UART.

6.5.8.2.3 Client with Generation Framing Error

The UART_CLIENT (*file uart_client_bad.vhd)* generates framing error because it can generate more CPU transmission requests than the buffer within the UART can handle. It is included here to show the potentials of transactions that can be created. An example of reported errors is shown below:

```
24943680 ns   Uart_client detected error in XMT Err bitObserved PIR = 100011Resetting XMT
side
24943920 ns   Uart_client detected error in MT Err bit.  Observed PIR = 100011Sending new data
26375840 ns   Uart_client detected error in XMT Err bitObserved PIR = 111111Resetting XMT
side
26757920 ns   Uart_client detected error in XMT Err bitObserved PIR = 100011Resetting XMT
side
26758160 ns   Uart_client detected error in MT Err bit.  Observed PIR = 100011Sending new data
26807480 ns   `` XMT Data error                    Parity ON  Observed TXD data = 11101100
Expected TXD data = 11110111
26896640 ns   `` XMT Data error                    Parity ON  Observed TXD data = 11101101
Expected TXD data = 11111011
```

6.5.9 Reading Text File into a Linked List

The parser package (*tb/parser_tb.vhd)* includes a procedure called *GetData* that collects a file of data in Hex notation into a linked list. This can be used in a client (or in a special parser instruction) to collect the data from a file, and then use that data from the linked list. The data types and procedures are shown below. Figure 6.5.9 is a sample code that demonstrates a trivial application of this procedure.

```
 -- Pointers for use in loading data from a file of undeterminate length
--   type DataRec_Typ; -- incomplete type
--   type DATAPNTR_TYP is access DataRec_Typ;
--   type DataRec_Typ is record
--     Data  : std_logic_vector(WordWidth_c - 1 downto 0); -- 7 .. 0
--     NextP : DATAPNTR_TYP;
--   procedure GetData -- in Parser_Pkg  --- DATA is in HEX in the file
--     (constant FileName_c    : in    string;
--      variable DataPntr_v    : inout DATAPNTR_TYP);
-- Compile:  vcom -explicit -work work_lib -93 vhdl/tb/filedata.vhd
architecture RTL of filedata is                    -- file tb/filedata.vhd
  signal Data : std_logic_vector(7 downto 0);
begin  -- architecture RTL

  GetDataFromFile_Proc : process is
    variable Head_v : DATAPNTR_TYP;
    variable TempPtr_v : DATAPNTR_TYP;
    variable L : Line;
  begin  -- process GetDataFromFile_Proc
    wait for 100 ns;
    GetData (FileName_c => "c:/uart/data8.txt",
             DataPntr_v => Head_v);
    report "Send File data from: c:/uart/data8.txt";
    while Head_v /= null loop
      Data <= Head_v.Data;       -- copy data
      HWrite(L, Head_v.Data);
      Writeline(Output, L);
      TempPtr_v   := Head_v;
      Head_v      := Head_v.NextP;      -- advance pointer
      Deallocate(TempPtr_v);  -- remove original item since it was consumed
      wait for 100 ns;
    end loop;
    report "Done data, end of sim";
    wait;
  end process GetDataFromFile_Proc;
end architecture RTL;
```

```
Simulation Transcript
run 1 ms
restart
run 1000 ns
# ** Note: Send File data from:
c:/uart/data8.txt
#    Time: 100 ns  Iteration: 0  Instance:
/filedata
# B8
# A0
# 76
# 01
# FE
# FF
# ** Note: Done data, end of sim
```

Figure 6.5.9 Sample Code Demonstrating the Application of the GetData Procedure (*tb/filedata.vhd)*

7 DOCUMENTATION AND DELIVERY

The documentation format for the UART component follows the Motorola *Semiconductor Reuse Standard SRS06 Doc, Version 2.0, 10 DEC 1999*[1]. This document is included on the CD. It is copyright of Motorola, and is used by permission.

The purpose of this chapter is to demonstrate the various entries of a component document. Many of the entries in this documentation chapter are discussed in other chapters. Thus, to avoid repetition, only references to the chapters are provided whenever applicable.

This chapter also includes the spreadsheet calculations as defined by OpenMORE Assessment Program[2], the Industry Reference for IP Measure of Reuse Excellence. The CD includes a copy of the downloaded Excel spreadsheet. *The numbering system for the documention starts at 2.0 because it is intended to represent a stand-alone document based on the Motorola's Semiconductor Reuse Standard. Therefore, it does not follow the chapter numbering system.*

---

[1] http://www.mot-sps.com/technology/srs/index.html
IP/VC Block Deliverables Semiconductor Reuse Standard

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

Section 2 IP/VC Block Deliverables

2.1 Introduction

This section describes the deliverable requirements for IP/VC blocks. Virtual Component (VC) is defined by the Virtual Socket Interface Alliance (VSIA). The IP/VC Deliverables are the set of views in specified formats that must be submitted by the IP/VC Creator to allow for easy reuse and integration by SoC developers. These deliverables are stored in the IP Repository and are available to the Motorola design community through the Web. The SRS IP/VC Deliverable Standards are based on close collaboration with the Motorola SPS design community and are following industry standards such as the VSIA standards

.

This document describes the deliverable items for the UART model.

2.2 Reference Information

2.2.1 Documented References

See chapter 2, 3, 4 5, 6 of this book.

2.2.2 Terminology

See chapter 2 of this book.

2.3 Deliverable Overview

The IP/VC Block deliverable requirements for the SRS are listed in Table 2-1. Each deliverable has its own unique identification listed in the SRS ID column. The release phases are explained in 2.3.2 Phased Release of Deliverables and listed in the Release Phase column. The standard data formats selected for each deliverable are included in the Format Standard column. Entries in the Soft, Hard and Analog columns indicate whether the deliverable is a mandatory (M) or recommended (R) deliverable for soft, hard or analog IP/VC blocks. Finally, the VGS column indicates whether the deliverable is created by the View Generation System, a design environment for the creation of IP/VC deliverables.

RULE 2.3.1 All mandatory IP/VC deliverables must be created and provided by the IP/VC creator.

GUIDELINE 2.3.2 It is recommended that all recommended IP/VC deliverables are created and provided by the IP/VC creator.

Pages 13 through 19 of the Semiconductor Reuse Standard are included herein.

Pg 13

Pg 14

Pg 15

2.4 Data Organization for the Packaging of Deliverables

Pg 16

Pg 17

Pg 18

Pg 19

2.5 Deliverables Descriptions

2.5.1 General Deliverables

2.5.1.1 Metadata

The Metadata information is an important part of the IP Repository infrastructure. It enables the search and query of the IP/VC database by the IP/VC users. Metadata is information about the IP/VC block that is searchable and contained in the IP Repository database. The Metadata can be divided into two categories: 1) information conveying IP/VC characteristics such as die size, name, target library and speed; and 2) usage information, including reuse count and history, products used and lists of users. This Metadata is necessary to facilitate the search and management of the IP/VC block data in the IP Repository.

Table 2.5.1.1 represents a summary of the UART Metadata

Table 2.5.1.1 UART Metadata

Metadata Name	Description
UART	Module name
Description	EIA standard serial data communication RS-232 UART.
Version	Rev 1.0
Status	Submitted, verified, but could use more verification tests
Keywords	UART, FIFO, RS232
DateLoaded	August 28, 2000
OwingOrganization	VhdlCohen Training and Consulting, http://www.vhdlcohen.com
ContactPerson	Ben Cohen, vhdlcohen@aol.com
PatentInformation	This software can freely be used in conjunction with the book *Component Design by Example*, authored by Ben Cohen.
IP/VCTypesList	UART, FIFO, CPU Interface
IP/VCLibrary	Digital IP
ChangeRequestList	none
NotificationList	None
IntegrationList	None
SRSVersion (Compliance)	Does not apply
Certification Target	Uncertified
CertificationGrade	Uncertified
CertifiedBy	Uncertified
ClassificationGrade	Does not apply
PhaseStatus	Phase 1 completed
DesignSystem	For *Component Design by Example* book
Metadata Name	**Description**
DesignStyle	VHDL code retargetable to any library

Parameters	Width_g -- Bits/word Depth_ -- fifo depth ae_level_g -- Almost empty level af_level_g -- Almost full threshold Asynch_g -- asynchronous, synchronous
BlockType	Soft
DocFormat	Hard bound book
BusType	RS232, generic CPU bus
EndianType	Little
RTLBlockCov	98
Comment	98% statement coverage
SimLang	VHDL
SimEnv	VHDL testbench with text command files
SimEnvVersion	1.0
ACTestCov	0
ACTestCoMethod	None
ScanMethod	None
TestStrategy	None
TesterType	None
DCAtpgTestCov	0
DCFaultCon FuncPatt	0
RegMemCov Method	None
DRCRunset	None
DRCRunsetVersion	None
LVSRunset	None
LVSRunsetVersion	None
SimEngine	ModelSim EE
SimEngineVersion	5.4B
RTLCovTool	ModelSim EE
RTLCov ToolVerison	5.4B
DFTtool	None
DFTtoolVersion	None
AtpgTool	None
AtpgToolVersion	None
STA Tool	Synplify and Altera Max+Plus II
STA ToolVersion	Synplify 5.3.1 and Altera Max+Plus II 9.4
FormalVerification Tool	None
Metadata Name	**Description**
FormalVerification Tool version	None
PowerEstimation tool	None

PowerEstimation tool version	None
PlaceRouteTool	Altera Max+Plus II
PlaceRouteTool Version	9.4
PhysicalVerTool	None
PhysicalVerTool version	None
AnalogSim Tool	None
AnalogSim ToolVersion	None
MixedSignalSim Tool	None
MixedSignalSim ToolVersion	None
Comment	Synthesis and timing optimization performed with Synplify 5.3.1, Layout and timing performed with Altera Max+Plus II 9.4
Technology Specific Metadata	
TECHNOLOGY	FPGA, ASIC
TargetLib	Altera EPF10K10LC84-3
ProcessVariations	NA
WaferFab	NA
PinsNumber	37 for 8-bit configuration
MAX_SPEED	43 MHz
MAX_Power	TBD
AV_Power	TBD
GATE_COUNT	1400
AREA	TBD
Width	TBD
Height	TBD
NumMetalLayers	NA
XdirPorosity	NA
YdirPorosity	NA
YdirPorosity	Code written in RTL, can be targeted into any technology. Checkout out with Altera EPF10K10LC84-3

2.5.1.2 Errata Information

Errata Information details identified bugs, possible workarounds or impact to a design utilizing the block. This is not a separate document but will be available from the IP Repository web. Notification will be provided to users of an IP as soon as Errata Information is available.

None. However, for proper operation, software must read the transmit pending interrupt register (PIR) before writing more data for transmission. In addition, software must read the receive pending interrupt register before reading received data. The READ of a PIR is an atomic read/clear operation.

2.5.1.3 Certification Documents

Certification Documents are intended to provide information regarding the results of the self-certification process by the IP Creator. Certification reports and checklists need to be provided along with the IP/VC. Details regarding these documents can be found in Section 3 Certification.

Does not apply for this book

2.5.2 Documentation Deliverables

2.5.2.1 One Pager

The One Pager is primarily a technical document that provides prospective users with details about this IP/VC block. Users will search the IP Repository for blocks that match their requirements based on the technology specific Metadata (see Table 2-4). When one is found, the One Pager can be downloaded and provides a brief description of the features and capabilities of that particular IP/VC. It should include the data that is necessary for an engineer to perform a quick evaluation of the IP/VC. It will also include information such as an overview, features, modes, functional description with its scope and characteristics, parameter options and logical and physical implementation attributes.

Please refer to chapter 2 of this book under *Requirement specification, section 4.0 Architectural overview*.

2.5.2.2 Core/Block User Guide

The Core/Block User Guide is intended to provide information to the end consumer of the SoC design in which the IP/VC block is incorporated. This guide should include the data that is necessary for an engineer to design a product using the SoC chip. This will include information such as the electrical specification, register definitions, software access requirements and details of the functionality of the IP/VC block.

Please refer to chapter 5 of this book

2.5.2.3 Analog Design Guide

Does not apply

2.5.2.4 Integration Guide

The Integration Guide is intended to provide information to the SoC designer who will incorporate this IP/VC block. This guide should include the data that is necessary for an engineer to design a SoC using this IP/VC block. This will include information such as gate count, power requirements, physical layout interface, test interface, functional verification strategy and any other data required to smoothly integrate this IP/VC block into a SoC design.

Please refer to chapter 8 of this book under *Application of VC into higher level designs*

2.5.2.5 Test Guide

The Test Guide describes the test strategy for the IP/VC block. This document will specify the structures and methodologies required to make the IP/VC block testable in a SoC. All design approaches required to accommodate high quality and manufacturable testing will be described.

This is beyond the scope of this book.

2.5.2.6 Verification Guide

The Verification Guide is a document that details the verification strategy and environment used for functional verification of an IP/VC block. This includes descriptions of bus monitors and models as well as the concepts of the testbenches.

Please refer to chapter 4, *Verification Plan* and chapter 6 *Design Verification* of this book.

2.5.2.7 Schematics

The Schematics are intended to provide information to both the analog and SoC designers to visually communicate the circuit design and topology of an analog block. These can describe the design either hierarchically with multiple pages or as a flat design with a single or multiple pages. Identification of critical components and/or parameters, functional block areas and special notes can be used to more clearly communicate design attributes.

Please refer to chapter 5 of this book, *Design and Synthesis.*

2.5.2.8 DFMEA

The Design Failure Mode Effects Analysis (DFMEA) document is required as part of QS9000 compliance. The DFMEA consists of a standard form, which is provided as a separate template.

This is beyond the scope of this book.

2.5.3 Creation Guide

The Creation Guide provides detailed information about internal attributes of a design and the environment in which it was created, so that designers can recreate the original environment and change the design. Information in the Creation Guide is critical for blocks that can be changed as a part of chip integration. The information is not as critical for blocks delivered as hard macros, or for core designs that cannot be modified, but is useful for reviewing design decisions and trade-offs.

The design models were written in VHDL code that is compliant to the style described in *IEEE P1076.6 Standard For VHDL Register Transfer Level Synthesis*, with the exception of the use of VHDL'93 syntax instead the VHDL'87 syntax. Problems are not envisioned with this design approach because most synthesis tool vendors conform to this standard, and with the '93 syntax. In addition, the testbench design in vendor independent because it uses VHDL with no reliance on vendor specific PLI.

The tools used for this design are described in section 7.5.1.1, and were proven reliable in industry. Therefore, there is little risk in recreating the design environment for this UART.

2.5.4 Logic Design Deliverables

2.5.4.1 Synthesizable RTL Source Code

The Synthesizable RTL Source Code is a soft representation of the IP/VC block that completely and accurately models the functionality of the IP/VC block in supported RTL simulators and can be fully implemented in gates by using a supported synthesis tool.

See chapter 5 of this book, design and synthesis.

2.5.4.2 Synthesis Scripts

The Synthesis Scripts will be provided as a file to be read by the supported synthesis tools. This file will contain all the instructions necessary to transform the RTL code for the IP/VC blocks into a gate-level implementation.

See chapter 5 of this book, design and synthesis

2.5.4.3 Synthesis Constraints

The Synthesis Constraints will be provided as a file to be read by the supported synthesis tools. This file will contain all the instructions necessary to transform the RTL code for the IP/VC blocks into a gate-level implementation. The constraints will include area and timing considerations. With these constraints, SoC designers should be able to retarget an IP/VC block to a new technology quickly and easily.

See chapter 5 of this book, design and synthesis. For this book, no constraints were imposed on this design.

2.5.4.4 Synthesis Model

The Synthesis Model is a tool-dependent representation of the IP/VC Block for a given IC design process. While soft IP/VC is independent of a particular library, a Synthesis Model deliverable is required for a hard IP/VC block. This Synthesis Model is used in conjunction with other synthesis libraries, such as standard cell libraries or other hard IP/VC, when synthesizing a SoC.

See chapter 5 of this book, design and synthesis

2.5.5 Physical Design Deliverables

This is beyond the scope of this book.

2.5.6 Design-for-Test and Manufacturing-Related Test Deliverables

This is beyond the scope of this book.

2.5.7 Functional Verification Deliverables

2.5.7.1 Testbench

The Testbench deliverable consists of the components required to exercise the IP/VC block via traditional simulation methods that identify differences between expected and actual behavior. The components are part of a structured verification approach that exercises the IP/VC block with transaction-based stimulus. A block behavior checker may also be provided with the Testbench. Block behavior checkers ensure the IP/VC block is performing the correct function. It works in conjunction with monitors that check for protocol violations. The block behavior check makes sure an operation should be happening. The Testbench consists of the following components:

Interface Drivers: *Modules that drive the IP/VC block interface(s). The driver converts a transaction command into the proper protocol that the IP/VC block understands.*

Interface Monitors: *Modules used to detect protocol errors and provide abstractions of activity on the interfaces of the IP/VC block.*

Stimulus*: Block and system-level test case for functional verification. The stimulus is a sequence of transactions executed by the IP/VC block and Testbench components. Expected results may be part of the stimulus. If expected results are not part of the stimulus, a model of this must be provided to prove correct operation (see Models below).*

Top Level Netlist: *Structure that instantiates the IP/VC block and necessary verification components to exercise the IP/VC block.*

Models*: All models necessary to simulate the device for functionality. This may include: internal and external memory models, clock models, pad models and IP/VC block behavior models that run as part of the simulation.*

Scripts*: Necessary programs to build, run and debug the Testbench.*

Configuration Files*: Files required for the stimulus, drivers, monitors and models. For example, there may be text files to regulate the mode of operation for the IP/VC block or interface drivers.*

See chapter 6 of this book, design verification.

2.5.7.2 Interface Model

An Interface Model is a component model that describes the operation of a component with respect to its surrounding environment.

This is beyond the scope of this book.

2.5.7.3 Instruction Set Accurate (ISA) Model

An Instruction Set Accurate (ISA) Model describes the function of the complete instruction set recognized by a given programmable processor, along with (and as operating on) the processor's externally known register set and memory/input-output (I/O) space.

This is beyond the scope of this book.

2.5.7.4 Behavioral Model, Full Functional Model

This model is also called detailed-behavioral model by the VSIA. The Behavioral Model, Full Functional Model (FFM) is used for simulation purposes. It is developed in C (PLI interface) or Verilog RTL. It is fully accurate for data and timing. It provides the detailed, cycle-by-cycle behavior of the module. The Full Functional Model represents the full functionality of the block / core.

This is beyond the scope of this book..

2.5.7.5 Gate-Level Model, Full Functional Model for ATPG resimulation

The gate level model for the EPF10K10LC84-3 device produced by Altera's MAX+plus II Compiler is included in file *gates/uart.vho*

2.5.7.6 Clock Cycle Accurate Model

The Clock Cycle Accurate Model provides accurate cycle counts for each instruction that is processed, in terms of a regular system clock. Events occurring during the processing of an instruction are indicated with exact precision as to which clock cycle they occur in.

This is beyond the scope of this book.

2.5.7.7 Stub Model

The Stub Model is a very simple model that includes only the module definition as well as a list of all inputs, outputs or bidirectional signals the IP/VC block may have. Specific output values can be assigned.

This is beyond the scope of this book.

2.5.7.8 Formal Runtime Constraints

In the case that formal verification was being used in the soft IP development process, the Formal Runtime Constraints can be reused for the formal verification of another synthesized representation of this soft IP.

This is beyond the scope of this book.

2.5.7.9 Formal Equivalency Scripts and Data

In the case that formal verification was being used in the soft IP development process, the Formal Equivalency Scripts and Data can be reused for the formal verification of another synthesized representation of this soft IP.

This is beyond the scope of this book.

2.5.8 Design Analysis Deliverables

2.5.8.1 Timing Model

The Timing Model provides the characterized timing of the IP/VC block. It is generated from an actual characterization of the IP/VC block using the target library characterization conditions.

This is beyond the scope of this book.

2.5.8.2 Power Model

The Power Model provides the characterized power of the IP/VC block. It is generated from an actual characterization of the IP/VC block using the target library characterization conditions. The Power Model can be used for power analysis of an SoC design.

This is beyond the scope of this book.

2.5.8.3 Autobond Class File

This file is required for Autobond to run. It is automatically generated by the VGS and contains basic information about the module name as well as input and output pins.

This is beyond the scope of this book.

2.6 Design Status and Recommendations

This section is not in the Motorola guidelines. However, it is important to document the status and maturity of the design, particularly if there are certain known problems, and/or certain tests were not performed due to schedule or cost issues.

2.6.1 Status

The UART design was tested with the following values for the generics:

- Width_g 8 Bits/word
- Depth_g 4 fifo depth
- ae_level_g 1 Almost empty level
- af_level_g 3 Almost full threshold
- Asynch_g 1 asynchronous

The statement code coverage revealed that not all cases of the RTL were traversed. For example, the condition where the transmit logic was storing data into the transmit FIFO (i.e., Push) with a simultaneous extraction from the FIFO (i.e., POP) was not verified. Only statement coverage was used in the design verification.

2.6.2 Suggested Work

Should this UART be used for inclusion in a subsystem, the following work is suggested:

1. Simulate the design with different values of generics.
2. Create testcases that would enhance the code coverage.
3. Use all coverage categories (toggle, triggering, trace) to further evaluate the accuracy of the design and the thoroughness of the test vectors.
4. In the verifier model, add the correlation between the error and the requirement item number.
5. Add required design constraints in the synthesis script or constraint file to achieve the desired performance (area/speed).

2.7 OpenMore

Below is the spreadsheet calculation as defined by OpenMORE Assessment Program[3], the Industry Reference for IP Measure of Reuse Excellence. This UART design scored 418 out of 618, or 63%. This is relatively a low score. However, the design was intended for education of the front-end processes, rather the fabrication of a full-pledged UART.

[3] http://www.openmore.com

Openmore 1/10

Openmore 2/10

Openmore 3/10

Openmore 4/10

Openmore 5/10

Openmore 6/10

Openmore 7/10

Openmore 8/10

Openmore 9/10

Openmore 10/10

8 INTEGRATION OF COMPONENTS INTO DESIGNS

This chapter addresses the integration of the UART component into higher-level designs. It also addresses the generic issue of integrating components into designs.

8.1 APPLICATION OF UART INTO HIGHER LEVEL DESIGN

Components, Intellectual Properties, or subblocks are typically inserted into higher levels of hierarchy through component instantiations. Figure 8.1 represent the UART model instantiated into a higher level of hierarchy with slightly different top-level interfaces. Specifically, the output is tri-stated and the read/write control signals are merged at the top level into a single signal. In addition, there is only one chip select control signal. The attached code demonstrates this model.

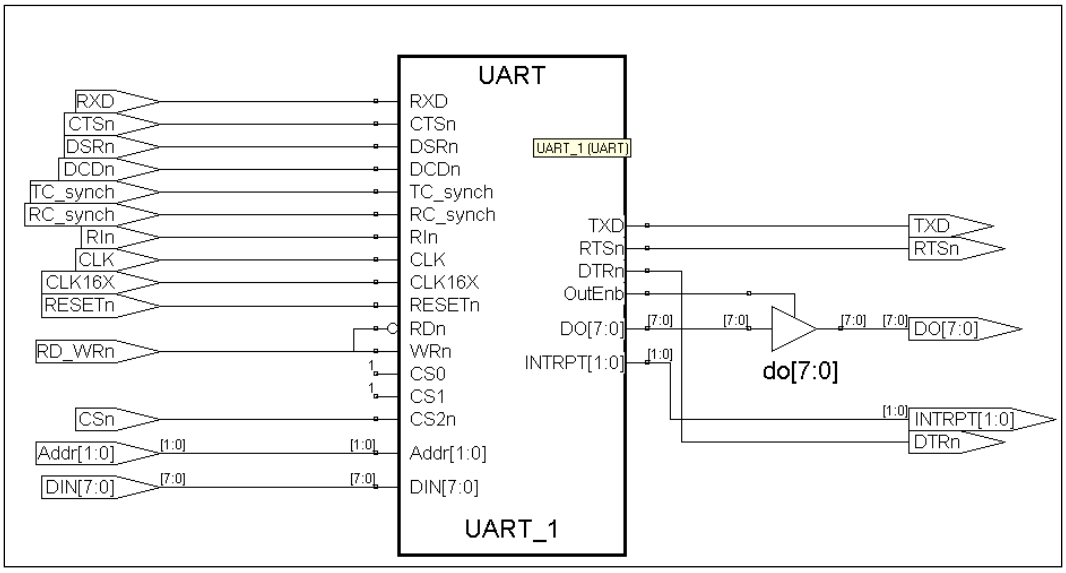


Figure 8.1 UART Instantiated into A Higher Level of Hierarchy (generated by *Synplify)*

Uart_level2 1 of 3

uartlevel2 2 /3

uart level 2 3 of 3

8.2 HIGHER LEVEL COMPONENT INCLUSION AND INTEGRATION [1]

8.2.1 Motivation for Change

Today's semiconductor technology allows designers to build chips with millions of transistors. Future manufacturing advancements promise to increase this size, thus allowing complex systems to be built on a single piece of silicon. At the same time, market competition is shortening product life cycles to less than the traditional product design time in some cases. Time to market is a critical issue, and is impacted by the definition and verification of the requirements and the resultant synthesized hardware. Furthermore, system companies must also cope with an increasing number of product derivatives and a decreasing number of available designers. Thus, the semiconductor industries as well as their customers at system companies are both faced with the problem of increasing complexity and diminishing design capability.

Increased IC density and shortening time to market are nothing new. Every decade or so, a methodological shift occurs in IC design. We relegate well-understood tasks to software applications and algorithms that increase productivity by freeing designers from low-level design details. Biasing transistors and interconnecting discrete three-terminal devices occupied the attention of thousands of electrical engineers doing system and IC design 30 years ago. The shift to gate libraries eliminated the need for that level of detail as IC transistor density increased and time to market shortened. The uses of schematic capture tools and place-and-route algorithms eliminated the need for designers to hand instantiate every small gate instance, again representing a paradigm shift in IC design methods and productivity. Roughly 10 years ago, logic synthesis became the widespread method for allowing designers to ignore gate level detail and focus on RTL based IC design. Now the same forces of IC transistor density and time-to-market pressure have set the stage for the adoption of another increase in design abstraction and an accompanying design methodology and optimized tools.

---

1 Extracted by permission from the Y Explorations, Inc. website http://www.yxi.com

The solution lies, as in the past, in increasing the level of design abstraction and thus reducing the number of objects a designer has to deal with during the design and verification of a System on a Chip (SoC). To achieve these objectives, raising the level of abstraction must be done in all aspects of the design process including (a) design specification and algorithms, (b) design descriptions and architectures, (c) design components and libraries, and (d) design testing and verification. In order to move to higher levels of abstraction the design community must also introduce standardization in languages, models, architectures, interfaces, protocols and other high-level concepts.

8.2.2 Related Industry Trends

Market pressure towards a shortened design cycle combined with worldwide IC designer shortages are globally impacting IC design process and business strategy. Systems companies have excellent expertise in their particular application domains and are interested in designing and manufacturing ICs quickly, but not necessarily in house. On the other hand, semiconductor companies are interested in attracting more customers, thus increasing the volume of their production. Therefore, both system companies and semiconductor vendors are now interested in using large megacells or cores, also commonly called semiconductor Intellectual Property (IP), in the design of their systems on silicon.

There are several hundred companies developing and marketing soft and hard IP cores. Soft cores are synthesizable high-level descriptions and are not process technology dependent. In contrast, hard cores are process technology dependent, and as a result can guarantee IP performance characteristics. Soft cores may not satisfy, the performance requirements in every possible process technology, but provide flexibility in moving from one technology or methodology to another.

In order to most effectively and efficiently compete in a global SoC market, to both shorten design cycles, and to improve designer productivity, companies must consider the following:

1. Maximize reuse of their own existing (or legacy) cores, and
2. Use of cores from other sources or IP providers, as depicted in Figure 8.2.2.

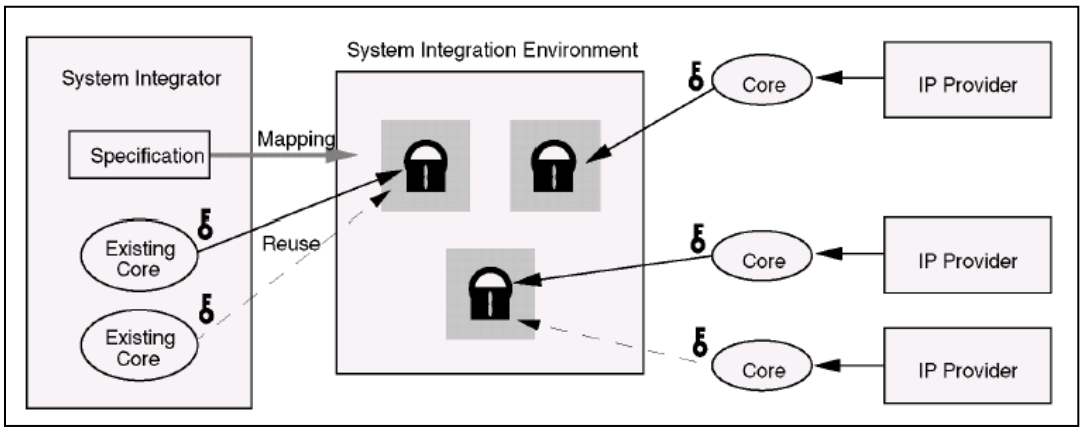


Figure 8.2.2 System Integration Using a Core-Based Design Methodology

The success of a core-based design methodology primarily depends on two key interdependent players: (1) core providers and (2) core integrators, as depicted in Figure 8.2.2. Core providers must develop, maintain, support and administer the distribution of cores. Core integrators require IP specification, exploration and assembly of complex IP in a fraction of the time currently spent in these areas.

On the IP provider side, the generation of cores requires the following issues be resolved:

1. Definition of suitable cores with high usability, such as standard processors, protocols, encoding/decoding standards, frequently used multimedia algorithms, and others.
2. Design of these cores with a proper set of parameters to cover a broad segment of the market.
3. Packaging these cores with proper interfaces and documentation so they can be easily used.
4. Developing methodologies for reuse of these components during standard design flow (IP plug-and-play).
5. Developing proper business and customer models for distribution and use of cores.

Support of cores requires capturing their functionality, electrical, mechanical, and timing parameters, as well as, the characterization, verification and packaging of cores for reuse by other EDA tools, designers and libraries. In other words, core providers must develop proper core models for easy insertion into system design for verification, simulation and testing. They also have to provide techniques for generating interfaces between cores and the rest of the system and tools for programming cores for specific use. Core maintenance deals with upgrades for functionality and features and porting cores to different technologies.

Administration requires marketing, accounting, and customer training in reuse techniques for different cores and different application domains. When a third party supplies cores or IPs, then administration must also include patenting, licensing and royalty collections from IP users.

On the system integrator side, the integration of cores requires the development of an infrastructure and a system-integration environment supporting design specification, exploration, reuse, and verification. The essential tasks include developing methodologies for (1) capturing design specification, (2) design space exploration, (3) evaluating cores provided by various IP vendors and in-house divisions, (4) design synthesis, and (5) design verification and testing. To create complex designs, the specification requires the use of higher-level models for design description. Exploration requires searching alternative algorithms, architectures, and components that satisfy the system functionality and constraints. This search is very difficult because of the variety of algorithms, architectures and components and the myriads of ways to use them. Core evaluation requires estimation of design quality metrics and measuring the suitability of previously designed cores or imported cores. System integrators must develop a design reuse strategy that can fully utilize existing cores for productivity improvement. They also have to develop a design synthesis and verification methodology that can easily and rapidly map the design specification into a set of interconnected cores for verification and testing.

8.2.3 Types of IP Cores

Components can be divided into five categories: combinatorial, sequential, storage, pipelined and cores. **Combinatorial components** such as adders, multipliers, and shifters do not have any storage elements. Most frequently, they perform arithmetic, logic and bit manipulation operations. They are characterized by input to output delay. **Sequential components** such as counters, registers and register files can be modeled by FSMs. In contrast to combinatorial components, they have states defined by the values in their storage elements such as flip-flops. Flip-Flops are controlled by clock signals. **Storage elements** such as RAMs, ROMs, FIFOs, and Stacks behave as combinatorial components with complex input/output protocols. Protocols define timing ranges between inputs and/or outputs. Control and data inputs must satisfy these timing constraints for storage elements to work properly. Storage elements may take several clock cycles to read or store data, and may have several states. They differ from combinatorial and sequential components by having asynchronous input and/or output protocols. **Pipelined components** generally are components from the previous four categories that are implemented such that the computation can be initiated with a new data set before terminating the previous instance of computation. Such an implementation increases the computation rate of the component over its non-pipelined implementation. **Megacells** are complex, concurrent super-state FSM

with Dataflow (SFSMD) that may contain combinatorial components, sequential components, storage components and other cores. Megacells interface via complex input/output protocols and typically have many states and non-deterministic execution time. The latter is because the execution time depends on the function being executed and the data being supplied to it.

In general, **controllers** are implemented with flip-flops and gates, data paths and processors with combinatorial, sequential, storage and pipelined components, while IP-centric processors and systems are implemented with all types of components. An IP can be any of the above types.
The main characteristic of a **hard IP** is that the IP itself has been previously designed, thus, its characteristics such as timing, performance, and area can be accurately measured or predicted. However, the implementation of a hard IP is considered proprietary information, fully accessible only to the IP provider. Methods used to regulate these restrictions include non-disclosure agreements between IP buyer and seller, or encrypting the information.

On the other hand, **a soft IP** is always provided with a description of its functionality that can be synthesized to fit any target technology. Hence, characteristics of a soft IP are generally difficult to measure and predict because it is dependent on many factors such as the optimization capability of the logic synthesis algorithms used, the target technology, etc.

The above defined components and IPs are necessary and sufficient for design of complex systems on silicon. Combinatorial, storage, pipelined and some sequential components can be automatically reused by many of the existing synthesis tools. However, up until now, there is no synthesis tool that allows all these complex IPs to be automatically reused. This is the primary motivation that drove the development of Reuse Automation tools from Y Explorations, Inc. (*YXI).*

8.2.4 Reuse Automation through High-Level Synthesis

A synthesis methodology is defined as a set of models and a set of transformations that refine the most abstract model, usually called the (executable) specification, into a lower-level structural model, usually called the implementation. A specification of a complete SoC is usually given with something similar to either a Super-state Finite State Machine with Dataflow (SFSMD) or a set of concurrent, hierarchical SFSMDs. The implementation is usually described by the block diagram or a netlist of components in the given library. Components in the library can be on system level, such as processors, memories and cores, or on RTL level such as combinatorial and sequential components, or on gate level such as gates and flip-flops.

Top-down methodologies start from a specification and refine it into a set of virtual components, which are further refined in subsequent steps until components in the library are found. This methodology, therefore, synthesizes a given specification from virtual components and then synthesizes each virtual component by repeating the process until each component is mapped to gates. The advantage of this methodology is that automatic synthesis tools that fit this approach are readily available. However, its main disadvantage is that the virtual components are not realistic, design quality metrics cannot be accurately estimated, and, therefore, many design iterations are needed to satisfy requirements, if convergence is possible at all. Furthermore, if the entire SoC is implemented in terms of gates, the resulting estimations, number of design iterations and the design itself could become unmanageable.

Bottom-up methodologies, in contrast, start by building simple components and then using them to build more complex structures. This process repeats until the design is completed. Applying this methodology to SoC designs would mean building cores first, and then integrating them later to obtain the required functionality. The advantage of this methodology is that component characteristics are known before being used (especially in the case of hard cores); therefore, design quality metrics can be accurately estimated. However, the component may not fit the required functionality well and may have non-matching protocols, which makes integration very difficult and costly because interfaces must be created manually. Currently, this methodology is most preferred by system designers for SoC designs. However, the approach requires manual interface and control logic design in which most of the design time and cost is in the interfacing and integration of IP cores.

The cost of SoC designs can be reduced by combining the best features from both top-down and bottom-up methodologies. These are: 1) automatic synthesis and 2) reuse of pre-designed cores or IPs. This combined design methodology is called an IP-centric methodology. IP-centric methodology assumes that there exist synthesis tools that can automatically reuse pre-designed cores or IPs. Using this approach, the system designer first describes the specification of the system using a high-level description language and design models described in previous sections. Then, during the design process, the designer will explore the possibility and cost of reusing components from the library that includes proprietary cores and third party IPs. When a desired design solution or design constraint is selected or determined, the synthesis tool is invoked to automatically produce interfaces that integrate components and IPs together so that they will perform the functionality given in the specification.

8.2.5 IP-Centric Synthesis Methodology[2]

This IP-centric synthesis methodology is intended to efficiently map higher-level models into architectures by fully utilizing components, including in-house and externally supplied IP cores. Such a design methodology is encapsulated in an environment provided by Y Explorations, Inc. (YXI), a company specializing in methodologies and tools for SoC design through IP Reuse Automation.
In contrast, to existing logic and behavioral synthesis EDA tools, YXI tools allow users to employ all five different models to describe the design, focusing particularly on SFSMD and concurrent SFSMD models. These models are given in a high-level description language, presently VHDL or Verilog. In contrast to other tools, the YXI tools also target IP-centric processor and system architectures. The tools generate RTL models that are synthesizable by commercially available logic synthesis tools. A further unique feature is YXI's IP database format that stores (in an encrypted format) all types of components including complex cores and IPs and their uses in a SoC design. In addition to providing automatic high-level synthesis from VHDL or Verilog, the YXI environment (as shown in Figure 8.2.5) supports a large degree of designer interaction by allowing the option to override recommendations made by these automated tools and fine-tune the resulting RTL design.

8.2.6 Summary and Recommendation

The traditional forces of shortening design cycles, combined with extraordinary chip complexity are forcing change in approaches to designing and manufacturing systems on silicon. The new SoC business model favors out-sourcing all but high value added functions and keeping only necessary application domain knowledge in-house. This business model favors new design methods in procuring and reusing IPs. This IP reuse requires new technology, methodology and tools. Tools like those supported by YXI, support this new paradigm based on hierarchical concurrent communicating SFSMDs, IP-centric architecture and IP-centric Reuse Automation methodology. Reducing everything to IC standard cells is not efficient and not profitable since it requires many iterations and large design teams that are difficult to manage. Design assembly from predefined cores is more economical. Furthermore, cores may contain knowledge not broadly available and allow flexibility of reprogramming and re-synthesis. This methodology and Reuse Automation tools allow designers to quickly assemble SoC designs from cores or IPs and automatically synthesize the functions not performed by previously designed and verified cores. The tools also support a unique IP database and tools for characterization and automatic integration of IPs into SoC designs.

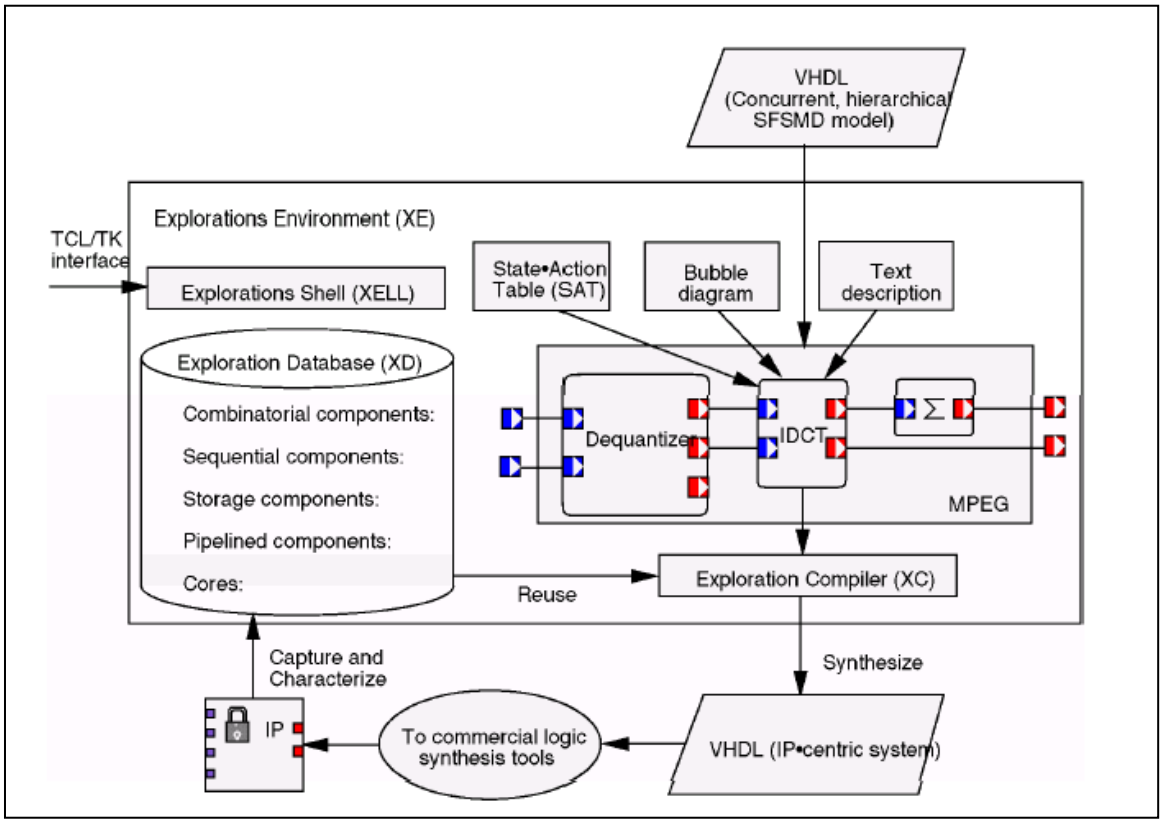


Figure 8.2.5. The YXI eXplorations Environment (XE)

9 REFLECTIONS

This design experience brought up several issues related to requirement definitions, RTL design, and verification. These issues and lessons learned are discussed herein.

9.1 REQUIREMENTS

9.1.1 Realities

Ideally, the requirement specifications are 100% firm before starting any implementation specifications or design effort. However, as one delves into the design and gets a deeper understanding of the performance and holes in the specifications, it often becomes necessary to update the requirement specification. Some of the issues that surfaced during the design effort that caused a change in the requirements are discussed in this chapter. Other requirement issues also surfaced during the definition of the verification plan because it forced the revisit of the requirements and the challenges of verifying correctness of the design. Poorly stated or ambiguous requirements became obvious in this exercise.

9.1.2 Costs

GUIDELINE: Evaluate cost implications implied by the requirements.

Engineers tend to be perfectionists and often easily add unnecessary or superfulous requirements for additional security and potential growth. However, it is important to consider that any additional requirement causes additional costs in all the phases of the design process, including:

1. Entry and review of the requirement document.
2. Entry and review of the architecture implementation document

3. Entry and review of the verification plan
4. Design and debugging of the code (RTL, Behavioral)
5. Design of the testbench and the testcases
6. Design of the verifier
7. Synthesis optimization and timing
8. Back-end processes (testability insertion, regression, release simulations)
9. Documentation, software implications, etc.

Figure 9.1.2 demonstrates the ripple effects caused by an additional requirement.

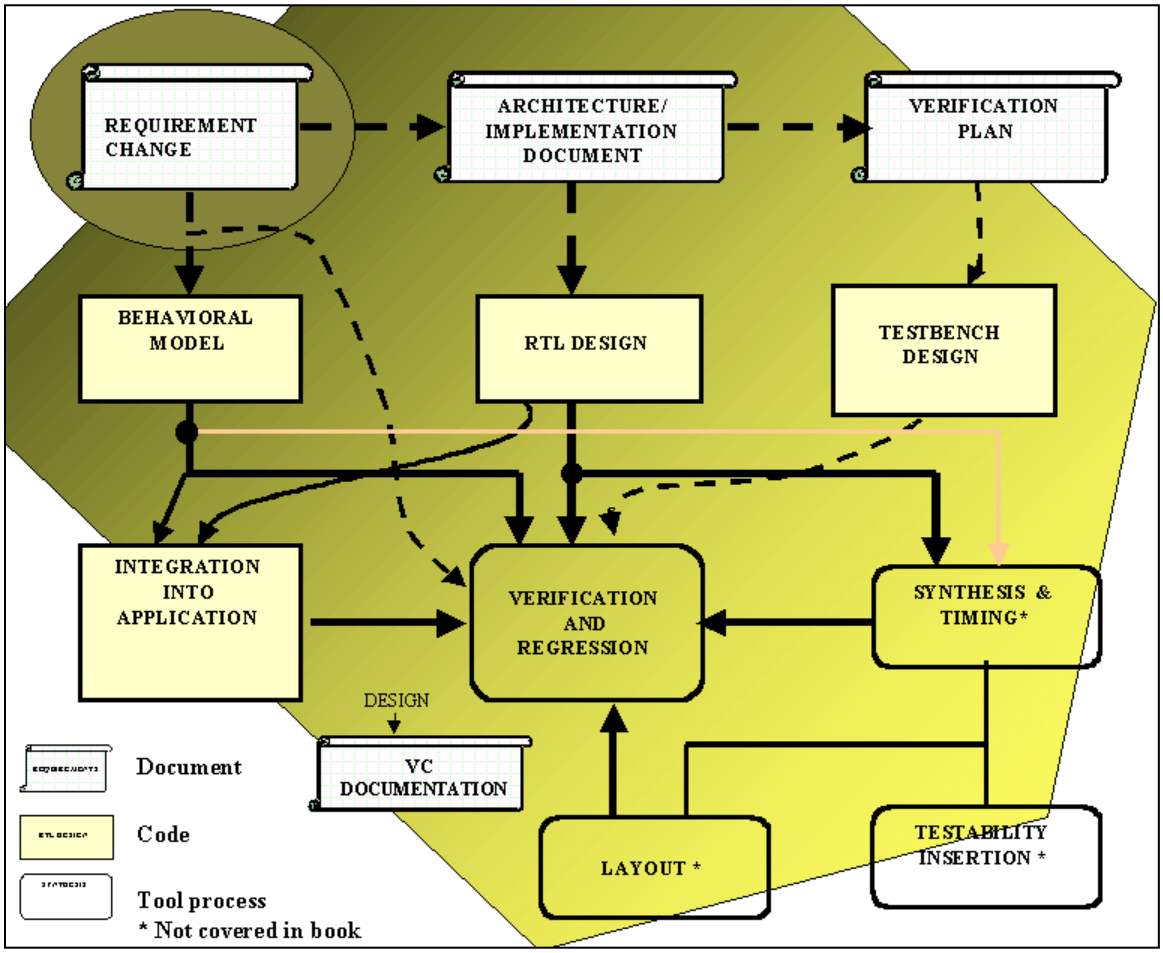


Figure 9.1.2 Ripple Effects Caused by an Additional Requirement

In this UART model, an example of a potential superfluous, and initially trivial addition to the specification, is the requirement for a second parity bit. This change could be justified by the need of a higher error detection level. However, this simple modification creates additional issues that are reflected into costs:

1. Where are the parity located in the serial format?
2. Are the parity bits organized as ODD and then EVEN, or EVEN and then ODD?
3. What happens to the serial format if there is no parity? Are spaces inserted instead of parity bits?
4. Should a one-bit parity be supported for compatability with the standard RS232 format?
5. How should this one or two-bit parity selection be controlled?

The addition of a second parity modifies the FSM machines for the receive and transmit logic. Additional tests modes and test cases must be supplied. The verifier must add more code to verify these options.

The more complex FSMs may impact timing and resources. That would require additional efforts in defining the constraints to achieve the desired performance.

Documentation must now include the effects of this additional parity bit.

The bottom line is that engineers should not be quick to the draw in adding requirements without evaluating the cost implications and benefits implied by those requirements.

9.1.3 System implications

GUIDELINE: Evaluate system implications implied by the requirements.

In the initial requirements, the FIFO status registers for the transmit and receive functions were to be reported. This requirement appeared logical because it would provide to a CPU information about the *empty, almost-empty, half-full, almost-full,* or *full* status of each FIFO (transmit and receive). In addition, the initial specification required that an interrupt would occur when the desired enabled level was reached. However, after further analysis of the design, and during the definition of the verification plan, it became apparent from a system's operation viewpoint, that this requirement was somewhat meaningless. It is incorrect to have the interrupt occur for the transmit portion of the logic when the FIFO just reaches a trigger level. On the transmit logic, the trigger level would be reached when the CPU writes data into the FIFO. For example, if the trigger level is *half-full,* (e.g., 2), and two words are written into the FIFO, then it would be incorrect to have an interrupt at that time. The interrupt is desired when the FIFO gets off that *half-full* trigger level, or done sending enough data so that the FIFO is just below the *half-full* level. This would allow the CPU to send more data.

Another system implication is the value of the information obtained by reading the status register. A more meaningful item would be the knowledge that the trigger level was reached. This represents the value of the transmit and receive pending interrupt registers (PIR). The PIRs would be latched when the trigger level is reached, and then atomically cleared with the READ of the PIRs.

GUIDELINE: Evaluate the significance and accuracy of timing requirements.

The original specification specified that a new serial message shall be started within fifty system clock cycles. That number was based on an assumed baud rate. However, the baud rate is defined externally to the UART model, and fifty system clock cycles will not necessarily satisfy this requirement.

Another weakness in the original specification was the lack of consideration in the pipelining optimization for the transfer of consecutive queued messages.

Because of these two inadequacies in the original requirements in section 6.0, the updated document was rephrased to read as follows:

> *If all the conditions for transmission are satisfied, and no transmission is in progress, then a new serial message shall be **started within two baud cycles**. If a message **is queued no later than two baud cycles** from the completion of an on-going message, and all the conditions for the new message are satisfied, then the new message shall **immediately follow the on-going message** with no additional STOP bits between the two messages.*

9.1.4 Consistency

GUIDELINE: Maintain symmetry in the transfer of information within the design

The UART model consists of two separate, but related functions: the transmit and the receive of serial data. The CPU provides individual controls to these two functions. From a design, software programming, and verification viewpoint, the information is easier to handle if it is symmetrical, or similar in nature. For example, the identity of the bits within the PIR, and the interrupt enables are maintained in the same bit positions, whenever possible (e.g. *Empty, half-full, etc.)*.

9.2 DESIGN

GUIDELINE: In specifying subblocks, evaluate the significance and implications of the information provided by the ports. The subblock needs to provide information that would otherwise be difficult or costly to generate outside the subblock. Avoid the need to reconstruct information outside a subblock if that information can easily be performed from within the subblock.

GUIDELINE: Evaluate the cycle timing implications when the subblock is used in different partitions. Use the "generate" feature of VHDL to allow for various applications of the subblock

GUIDELINE: Reset registers that would otherwise cause problems in the system, and 'U's in the simulation. Initialize all registers to ZEROs, even if the desired value is ONEs (such as the register that generates the *DTRn* signal). Use inverters on the output of the registers (i.e., the RTL "not" function) to achieve the desired inversions.

The goal of using subblocks is reuse. For this design, the FIFO model was the perfect candidate for reuse since it used in the transmit and the received section of the UART. The architecture of the FIFO was initially designed in a "generic" sense, where the output of the FIFO was registered, and the FIFO provided status information (e.g., *empty*) though its ports. The FIFO subblock was verified visually with a testbench. However, in the integration of the FIFO in the final design, several surprises surfaced:

1. **CYCLE TIMING**: The cycle timing of the receive logic required that the output occurs unregistered, whereas the transmit logic required a delayed registered version. The code was modified with a generic as shown below.

```
RcvFifo_Gen : if Xmt1_RCV0_g = 0 generate
begin  -- generate RcvFifo_Gen
  DataOut_r <= FIFO_r(RdPntr_r);
end generate RcvFifo_Gen;

XmtFiFo_Gen : if Xmt1_RCV0_g = 1 generate
begin  -- generate XmtFiFo_Gen
  FiFoOUT_Proc : process is
  begin  -- process FiFoOUT_Proc
    wait until Clk = '1';
    if Resetn = '0' then
      DataOut_r <= (others => '0');
    elsif pop_n = '0' then
      DataOut_r <= FIFO_r(RdPntr_r);
    end if;
  end process FiFoOUT_Proc;
end generate XmtFiFo_Gen;
```

2. **STATUS DATA FOR PIR**: The original FIFO design provided status information that needed to be manipulated or reconstructed to create the

PIR. This involved an edge detection of the status information for the transmit logic to detect when the status got off one state and into another. The external processing of this information was necessary because the original design of the FIFO just provided status data, but not the transition into a status. This created more logic than necessary because the FIFO already includes a counter that can be used for this transition detection, rather than the external derivation of this information form the status bits. In addition, because of the meaning of the *almost-empty* state, there was a problem in setting the correct PIR for that state. The FIFO design was then changed to compute the setting of the PIR for the transmit logic, and provided this vector onto a port. The setting of the receive side PIR was not derived from within the FIFO because the design worked correctly. In retrospect, the FIFO should have also provided through a port the setting of the receive PIR, instead of deriving it externally.

3. **REGISTER RESETS:** During the initial design, the FIFO was not initialized. However, it held status information that caused 'U's to propagate in the PIR. That demonstrated the need to reset critical registers. All registers were reset to ZEROs because it allows the use of scanable resets, where ZEROs are shifted for the scan reset.

9.3 VERIFICATION

GUIDELINE: Consider the cycle synchronization issues between the verifier and the UUT (i.e., is the verifier in synchronism with the UUT?). Designs that are control intensive are more sensitive to this cycle synchronization issue than data path designs, such as filters.

GUIDELINE: Log all transactions onto a file. Enable this logging function with a generic.

GUIDELINE: Consider the ease of modifying the sequence of transactions. Use TextIO command files or procedures to identify the sequences. Avoid putting extensive in-line code since making changes becomes more difficult.

GUIDELINES: Use a verifier model for the automatic detection and reporting of errors and links to the requirements.

As expected, verification was the most difficult, and important aspect of the design. The verifier model consisted of two main functions:

1. **Logging of transactions:** The logging of transactions (asserted by the clients and observed at the ports) provided invaluable debugging information because it abstracted the complex waveforms into readable

information. This task was relatively easy.

2. **Error detection and logging:** This aspect of verification felt like a necessary evil because it both correctly detected design errors, and at times, incorrectly reported errors when none really occurred. The initial design of the verifier model had the goals of treating the unit under test (UUT) as a black box. However, as discussed in chapter 6, the lack of accurate cycle synchronization created false error reports. To correct this situation, internal UUT signals (transmit *PUSH* and *POP*) were used to achieve accurate cycle synchronization between the design under test and the verifier**.** The values of those signals were transferred into global signals. Another potential alternatives to the application of global signals is the use of PLI. The subdirectory *ModelimSpy* includes compiled PLIs, and examples to read signals internal to a design with ModelSim. Note that the use of *ModelimSpy* is a simulator specific solution, and is not portable. Verification languages typically provide access to internal signals of a design. The issue of portability with verification languages arises here also, unless the verification languages are open.

9.3.1 Value of verifier

The verifier was an invaluable tool in detecting and reporting errors, particularly as the design was tuned for proper operation.

9.3.2 Code coverage

Only statement code coverage was available during the design of the UART for this book (for economics reasons). However, code coverage for the RTL models were very beneficial at determining whether sufficient test patterns were exercised. Code coverage was found useful in the evaluation of the verifier model to evaluate the traversal of error-detection code.

9.3.3 Debugger/LINTing

A good debugging and linting tool provides insights into the structure of a model, errors in synthesis constructs, and unused resources. In addition, it provides a link between the HDL model, the implied structures, and the simulation waveforms. This tool can also be used to understand an inherited design with poor documentation.

9.3.4 When is design fully verified

This was discussed in chapter 6. However, this design needs further testing and error insertions to provide a more thorough level of assurance that the design is correct. Since the purpose of this book is to demonstrate processes rather than complete verification of a design, this author is leaving to the user the task of continuing the verification task to a satisfactory level. However, directed verification tests of the required functions were exercised, as per test plan.

9.3.5 Text Command Files

The text command file proved beneficial for this design because the command sequence is readable, and acts like procedure calls. The command files can be changed with no model recompilation. In addition, different command files can be selected in the configuration declarations. With the *CALL* instruction, the sequence can easily be changed, again with no recompilation. The *CALL* instruction also allowed for the reuse of test sequences with different initial setups (e.g., no parity, even parity, odd parity, error mode). The *TextIO* parsing of the command file, and *TextIO* WRITE of the transaction and error logs were not very taxing on the simulator. At the RTL level, 40 ms of simulation time on a 500 MHz Pentium III took 70 seconds of real time, including the model loading time. If *TextIO* speed is an issue for large designs, then the text command files can easily be converted to binary files[1].

9.3.6 Review of testplan against verifier implementation

The testplan called for the flagging of the requirement number related to the error when reporting an error. However, this was missed during the implementation of the verifier. It was not a critical overview because the error message provided enough information about the error. This points to the importance of closely reviewing the verification model against the testplan.

---

[1] See chapter 11 of *VHDL Coding Styles and Methodology, 2nd Edition*, Ben Cohen,
isbn 0- 7923-8474-1 Kluwer Academic Publishers 1999

9.4 SUMMARY AND CONCLUSIONS

The front-end processes of specifying the requirements and the planning of both the implementation and verification of a design are necessary steps to ensure that the implemented design meets its intended goals and costs. Design reuse must be considered during the planning stage. Reuse has several meanings. For an original subblock design, it is taken in the sense of the user being the customer of that subblock. For a subblock that will be applied in several designs, then reuse of that subblock is more like an in-house IP. Other times, it is beneficial to purchase an IP for efficient reuse. See chapter 8.2 for a greater discussion on reuse and higher level component extraction and integration.

Ideally, different teams will be responsible for the design and verification efforts. Figure 9.4 reiterates the levels of efforts for a typical design. Many factors will modify these percentages, including:

1. Level of understanding of the requirements
2. Availability and maturity of IPs
3. Levels of designers' experiences
4. Availability of mature tools including synthesizers, linting, simulators, debuggers, IP integration, layout, timing analyzer, etc.
5. Availability of languages, and levels of language supported for the design and verification. The choice of HDL extends beyond VHDL and Verilog, and now includes versions of C. Even within an HDL, there are levels of design descriptions and language constructs that are (or will be) acceptable. This includes RTL and behavioral levels. Within the RTL arena, in both VHDL and Verilog, the IEEE standardization committee is working at extending and defining the constructs that represents hardware.
6. Availability of verification languages (see section 4.1.3)
7. Efficiency of reviews for all design phases

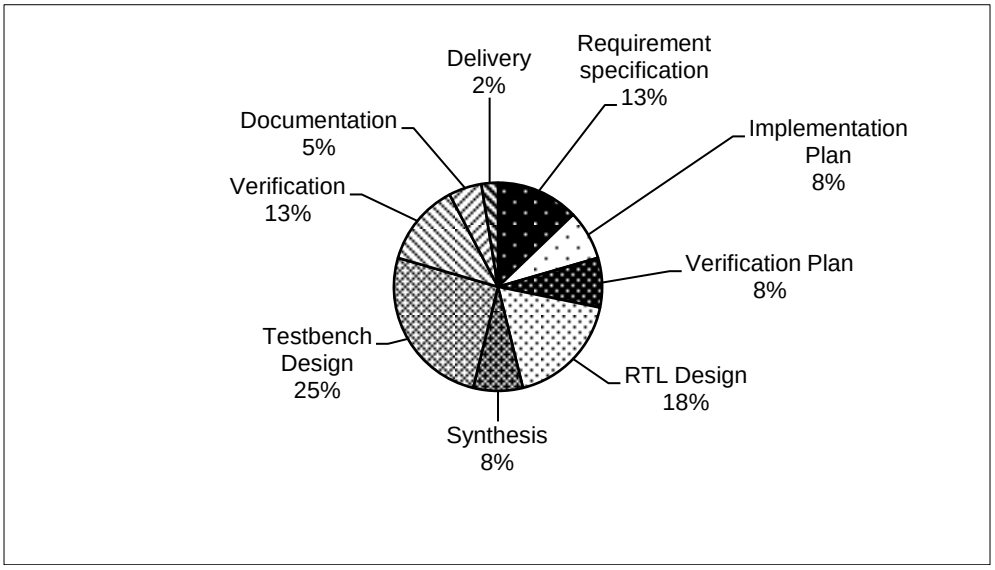


Figure 9.4 Levels of Efforts for a Typical Design

It is very important to seriously consider all reviews of the design. These reviews are summarized in Table 9.4.

Table 9.4 Design Reviews

ITEM	COMMENTS
Requirement specification	This answers the question of what is to be designed.
Implementation plan	This answers the question of how the design will be partitioned, which IP will be used, which device or technology will be implemented, and what tools will be used.
Verification plan	This answers the question of how the design verified, which feature will be tested, and what tools and languages will be used.
RTL design	This represents a detailed code review for compliance to requirements and to coding standards
Synthesis	This represents performance (area and timing) review, including optimization scripts.
Testbench design	This represents a code review for compliance to requirements and to style.
Verification	This represents the design detailed performance at the functional level
layout and timing	This represents performance (area and timing) review
Documentation	This represents a review to ensure documentation of all necessary items. Documentation is an item that is often considered with good intentions in the initial phases of a design, but is then indefinitely postponed during the release phase.

INDEX

D

E

F

G

I

L

M

O

P

R

S

T

U

V

W