"VERIFICATION OF SoC WITH ETHERNET INTERFACE"

A Major Project Report

Submitted in Partial Fulfillment of the Requirements for the Degree of

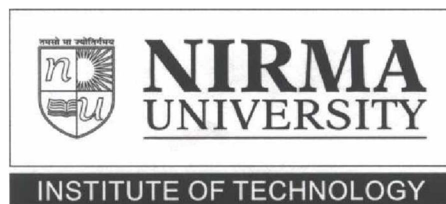
MASTER OF TECHNOLOGY

IN

ELECTRONICS & COMMUNICATION ENGG.

(VLSI Design)

By
Manish Raizada
(03MEC013)



Department of Electronics & Communication Engineering
INSTITUTE OF TECHNOLOGY
NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY,
AHMEDABAD 382 481

MAY 2005

CERTIFICATE

This is to certify that the Major Project Report (Part-I-II) entitled "**Verification of SoC with Ethernet interface**" submitted by Mr. Manish Raizada (Roll No.03MEC13) towards the partial fulfillment of the requirements for Semester III-IV of **Master of Technology (Electronics & Communication Engg.)** in the field of **VLSI Design** of **Nirma University of Science and Technology** is the record of work carried out by him under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this major project work to the best of our knowledge have not been submitted to any other University or Institution for award of any degree or diploma.

Date:

Project Guide:

Miss Mittal Patel
ASIC Verification Dept.
eInfochips Limited, Ahmedabad

Facilitator at Institute:

Prof. Y. N. Trivedi
Electronics & Comm. Engg. Dept.
Institute Of Technology,
Nirma University, Ahmedabad

Dr. M. D. Desai
HOD
Dept of Electronics & Comm. Engg.
Institute of Technology, Nirma University

Dr. H. V. Trivedi
Director
Institute of Technology
Nirma University, Ahmedabad

Signature of Examiners:

ACKNOWLEDGEMENT

It gives me a great pleasure to take this opportunity to thank to **eInfochips Pvt. Ltd**. and **Mr. Nilesh Ranpura** for giving me such a great opportunity to do project in their esteemed organization. I deem it my privilege to have carried out this dissertation work under this well-known quality conscious organization.

I express my deep sense of gratitude to **Mr. Pranav Tailor** and **Miss Mittal Patel** for their personal involvement in every facet of this work and readiness to resolve any point of confusion by mutual discussion. I would also like to thank **Mr. Vijay Patel** and **Mr. Dharmendra J. Patel** for their co-operation and providing the necessary facilities for carrying out this work.

I would like to thank **Prof. Y. N. Trivedi** for his help, valuable suggestions and moral support. Finally, I would like to thank my parents for their constant love and support and for providing me with the opportunity and the encouragement to pursue my goals.

(Manish Raizada)

Confidentiality Notice

The contents of this document constitute valuable proprietary and confidential property of Verisity Design, Inc. and its licensors, including eInfochips, Inc. No part of this information product may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise without prior written permission from Verisity Design, Inc.
Information in this product is subject to change without notice and does not represent a commitment on the part of Verisity. The information contained herein is the proprietary and confidential information of Verisity or its licensors, and is supplied subject to, and may be used only by Verisity's customers in accordance with, a written agreement between Verisity and its customers. Except as may be explicitly set forth in such agreement, Verisity does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy, or usefulness of the information contained in this document. Verisity does not warrant that use of such information will not infringe any third party rights, nor does Verisity assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraphs (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Destination Control Statement

All technical data contained in this product is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Abstract

Verification is not a test bench, nor is it a series of test benches. Verification is a process used to demonstrate the functional correctness of a design. By saying functional correctness means the design to be verified has to adhere with some predefined rules or standard. In other words design under test should behave as per specified rules. We all perform verification processes throughout our daily lives: balancing a checkbook, tasting a simmering dish, associating landmarks with symbols on a map. These are all verification processes.

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System-on-Chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers is usually twice the number of RTL designers. When design projects are completed, the code that implements the test benches makes up to 80% of the total volume. It is also the reason verification is currently the target of new tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher levels of abstraction and automation. Providing higher levels of abstraction enables you to work more efficiently without worrying about low-level details.

eVC consists of a complete set of elements for stimulating, checking and collecting coverage information on the device under test (DUT). The eVC expedites creation of a more efficient test bench for DUT. The eVC can work with both Verilog and VHDL devices and with all HDL simulators that are supported by Specman Elite. eVC can be used as full verification environment or can be added to existing environment. The eVC interface is viewable and thus can be the basis for user extensions.

This thesis report put some light on developing verification environment for the verification of Ethernet eVC (e Verification Component). Architecture of eVC has been discussed. eVC reduce verification time by atomizing the verification process. First few pages of report gives the answers of some fundamental questions like what is verification, why it is required and importance of verification. White Box approach has been used to verify eVC, and Coverage Driven Verification process used to verify functionality.

Contents

Page No.

List of Figures (vii)
List of Tables (ix)
Abbreviation (x)

CHAPTER 1: Introduction 1
1.1 Black Box Verification 2
1.2 White Box Verification 2
1.3 Grey Box Verification 2
1.4 Levels of Verification 3
1.5 Need of Verification languages 5
1.6 History and advantages of 'e' 7
1.7 Verification Reuse 7
1.8 e Verification Component 8
1.9 About Ethernet eVC 10
1.10 Ethernet Basics 10
1.11 Verification for Ethernet eVC 12

CHAPTER 2 : Review Of Literature 17
2.1 The IEEE 802.3 Logical relationship to the ISO Reference model 17
2.2 Ethernet MAC Sublayer 18
2.3 Media Access Control Frame Structure 19
2.4 Half duplex transmission 22
2.5 Full duplex transmission 25
2.6 Ethernet PHY layer 27
2.7 Encoding for signal transmission 28
2.8 The IEEE 803.2 Physical Layer relationships to the ISO Reference Model 30

CHAPTER 3 : System Review (Basic Theory) 39
3.1 e Verification Component 39
3.2 Ethernet eVC 39
3.3 Features of Ethernet eVC 40
3.4 Ethernet traffic emulation 40
3.5 Flow of data within the agents 42
3.6 Agent architecture 42
3.7 Monitor & BFM architecture 44
3.8 Scoreboard architecture 45
3.9 Scoreboard checking 46
3.10 Topologies for verification at the module level 47
3.11 Single port MAC dut for non-layered interfaces 48
3.12 Multi port MAC dut for non-layered interfaces 48
3.13 Single port PHY dut for non-layered interfaces 49
3.14 Multi port PHY dut for non-layered interfaces 49
3.15 Verification environment architecture 50
3.16 Is data collected correctly? 52

CHAPTER 4 : System Design **54**
4.1 Structure of sequence 54
4.2 Ethernet sequence structure 55
4.3 Management sequence structure 56
4.4 Injecting Ethernet packets with protocol errors 58
4.5 Monitoring Coverage and checkers 61

CHAPTER 5 : Coverage Driven Verification **63**
5.1 Higher Abstraction 64
5.2 Coverage requirements 65
5.3 Steps for achieving regression 67
5.4 Analyzing Bugs 74

CHAPTER 6 : Results & Discussion **80**
6.1 Normal Scenario 80
6.2 Collision Scenario 82
6.3 Zeroipg check Scenario 85
6.4 Injecting RX_ER in ipg phase of packet 86
6.5 Results 87

CHAPTER 7 : Conclusion & Future Scope **88**
7.1 Conclusion 88
7.2 Future Scope 88

REFERENCES **89**
Appendix A **90**
Appendix B **109**

List of Figures

Figure1.1: Design and verification flow: A typical sequence of steps for logical	4
Figure 1.2: Verification process flow	4
Figure 1.3: Typical verification component	9
Figure 1.4: An Ethernet network runs CSMA/CD over coaxial Cable	10
Figure 1.5: Example Point-to-Point Interconnection	11
Figure 1.6: Verification Environment	16
Figure 2.1: Ethernet's Logical Relationship to the ISO Reference Model	17
Figure 2.2: MAC and Physical Layer Compatibility	18
Figure 2-3: MAC Frame format	19
Figure 2.4: Address designation	20
Figure 2.5: MAC Frame with Gigabit Carrier Extension	24
Figure 2-6: A Gigabit Frame-Burst Sequence	25
Figure 2.7: Full Duplex Operation Allows Simultaneous Two-Way Transmission on the Same Link	26
Figure 2.8: An Overview of the IEEE 802.3 Flow Control	26
Figure 2.9: A Concept Example of Baseline Wander Sequence	29
Figure 2.10: Transition-Based Manchester Binary Encoding	29
Figure 2.11: The Generic Ethernet Physical Layer Reference Model	30
Figure 2.12: Transmission between MAC & PHY	32
Figure 2.13: System level diagram of Reduced Gigabit Media Independent Interface (RGMII)	35
Figure 3.1: Architecture of Ethernet eVC	40
Figure 3.2: Agent Architecture	43
Figure 3.3: Agent Architecture for Layered Interfaces	44
Figure 3.4: Monitor and BFM architecture	45
Figure 3.5: Functioning of Scoreboard	47

Figure3.6: Ethernet eVC in user's verification environment 47

Figure 3.7: Single Port MAC DUT 48

Figure 3.8: Multi-Port MAC DUT 48

Figure 3.9: Single Port PHY DUT 49

Figure 3.10: Multi Port PHY DUT 49

Figure 3.11: Ethernet *e*VC VE functional block diagram 50

Figure 4.1: Ethernet Sequence Structure 55

Figure 4.2: Management Sequence Structure 56

Figure5.1: Functional coverage serves multiple simulations Scenarios 64

Figure5.2: Functional coverage of Ethernet Packet 66

Figure5.3: Functional coverage model for Ethernet eVC 69

Figure 5.4: Illegal inter packet gap between two packets 77

Figure 6.1: Simulation of Normal Scenario 80

Figure 6.2: Coverage of Normal Scenario 81

Figure 6.3: Simulation of Normal Packet scenario (RGMII interface) 82

Figure 6.4: Simulation of Collision Scenario 83

Figure 6.5: Coverage of Collision Scenario (GMII Interface) 84

Figure 6.6: Simulating collision scenario (RGMII Interface) 84

Figure 6.7: Zeroipg check scenario (GMII Interface) 85

Figure 6.8: Coverage of Zeroipg check Scenario 86

Figure 6.9: TX_ER insertion in Data Phase of Ethernet Packet (RGMII Interface) 86

Figure 6.10: RX_ER insertion in IPG phase of Ethernet Packet (RGMII Interface) 87

List of Tables

Table 2.1: Limits for Half-Duplex Operation 24

Table 2.2: Permissible encoding of TXD<7:0>, TX_ER, and TX_EN 33

Table 2.3: Permissible encoding of RXD<7:0>, RX_ER, and RX_DV 34

Table 2.4: Signal description of RGMII 36

Table 2.5: Signal coding for TXD, TXERR and TX_EN 37

Table 2.6: Signal coding for RX_DV, RXERR and RX_ER 38

Table 5.1: Correlation between functional coverage and code Coverage 65

Table 5.2: Scenarios to be checked on various Ethernet Packets 68

Abbreviation

VE: Verification Environment, Aggregation of the eVC along with all other required verification components.

eVC: e Verification component, it represent complete verification environment for DUT.

DUT: Device Under Test, It is a device that is to be verified using eVC.

eRM: e Reusable Methodology, term used for eVC, to make eVC e Reusable in the sense it can be integrate with other eVC on Soc.

Ethernet eVC: Complete verification environment (eVC) of Ethernet protocol, it can be use to verify IEEE 802.3 Std, 2000 Edition compliant devices.

CDV: Coverage Driven Verification, ease the process of verification by focusing time and compute resources on simulations that are indicating coverage.

Agent: Top level agent, which contain active and passive agent.

Active Agent: Drives stimulus to DUT.

Passive Agent: Collect packets from DUT.

Monitor: A unit instance that passively monitors the DUT signal.

BFM: Bus Functional Model a unit instance that interacts with the DUT and drives or samples the DUT signals.

Chapter 1
Introduction

As the project title explains, this project work is regarding verification of Ethernet eVC in which protocol adherence of Ethernet eVC has been checked for media independent interfaces GMII, SGMII, RGMII. As eVC has been written in 'e' language hence, understanding of language and Specman Elite (tool) are prerequisite for project work.

Before going in to the detail discussion of verification of Ethernet standard let us find answers of some basic questions like What is verification? , Why it is required? , What are the kinds and levels of verification and importance of verification? Starting with IEEE Definition about Verification, Verification means "Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."

Verification is not a test bench, nor is it a series of test benches. Verification is a process used to demonstrate the functional correctness of a design. By saying functional correctness means the design to be verified has to adhere with some predefined rules or standard. In other words design under test should behave as per specified rules. We all perform verification processes throughout our daily lives: balancing a checkbook, tasting a simmering dish, associating landmarks with symbols on a map. These are all verification processes.

Today, in the era of multi-million gate ASICs, reusable Intellectual Property (IP), and System-on-Chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers is usually twice the number of RTL designers. When design projects are completed, the code that implements the test benches makes up to 80% of the total volume. It is also the reason verification is currently the target of new tools and methodologies. These tools and methodologies attempt to reduce the overall verification time by enabling parallelism of effort, higher levels of abstraction and automation. Providing higher levels of abstraction enables you to work more efficiently without worrying about low-level details.

Automation tools helps in reducing time but they requires standard processes with well-defined inputs and outputs. Not all processes can be automated. It is possible to automate

some portion of the verification process, especially when applied to a narrow application domain. Because of the variety of functions, interfaces, protocols, and transformations that must be verified, it is not possible to provide a general-purpose automation solution for verification.

The main purpose of functional verification is to ensure that a design implements intended functionality. Functional coverage reconciles a design with its specification. It is important to note that, unless a specification is written in a formal language with precise semantics, it is impossible to prove that a design meets the intent of its specification. Functional verification can be accomplished using three complementary but different approaches: black box, white-box, and grey-box.

1.1 Black-Box Verification:

With a black-box approach, the functional verification must be performed without any knowledge of the actual implementation of a design. All verification must be accomplished through the available interfaces, without direct access to the internal state of the design, without knowledge of its structure and implementation. This method suffers from an obvious lack of visibility and controllability. A black-box functional verification approach forms a true conformance verification that can be used to show that a particular design implements the intent of a specification regardless of its implementation. It is mostly used approach.

1.2 White-Box Verification:

As the name suggests, a white-box approach has full visibility and controllability of the internal structure and implementation of the design being verified. This method has the advantage of being able to quickly set up an interesting combination of states and inputs, or isolate a particular function. This approach is tightly integrated with a particular implementation and cannot be used on alternative implementations or future redesigns. It also requires detailed knowledge of the design implementation.

1.3 Grey-Box Verification:

Grey-box verification is a compromise between the aloofness of a black-box verification and the dependence on the implementation of white-box verification.

Verification is a necessary evil. It always takes too long and costs too much. Verification does not generate a profit or make money: after all, it is the design being verified that will be sold and ultimately make money, not the verification. Yet verification is indispensable.

Verification is a process that is never truly complete. The objective of verification is to ensure that a design is error-free, yet one cannot prove that a design is error-free. Verification can only show the presence of errors, not their absence.

1.4 Levels of verification:

There are four levels of verification:

1.4.1 Component testing: Testing conducted to verify the implementation of the design for one software element (unit, module) or a collection of software elements.

1.4.2 Integration testing: An orderly progression of testing in which various software elements and/or hardware elements are integrated together and tested. This testing proceeds until the entire system has been integrated.

1.4.3 System testing: The process of testing an integrated hardware and software system to verify that the system meets its specified requirements.

1.4.4 Acceptance Testing: Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

1.4.5 Verification and design process flow:

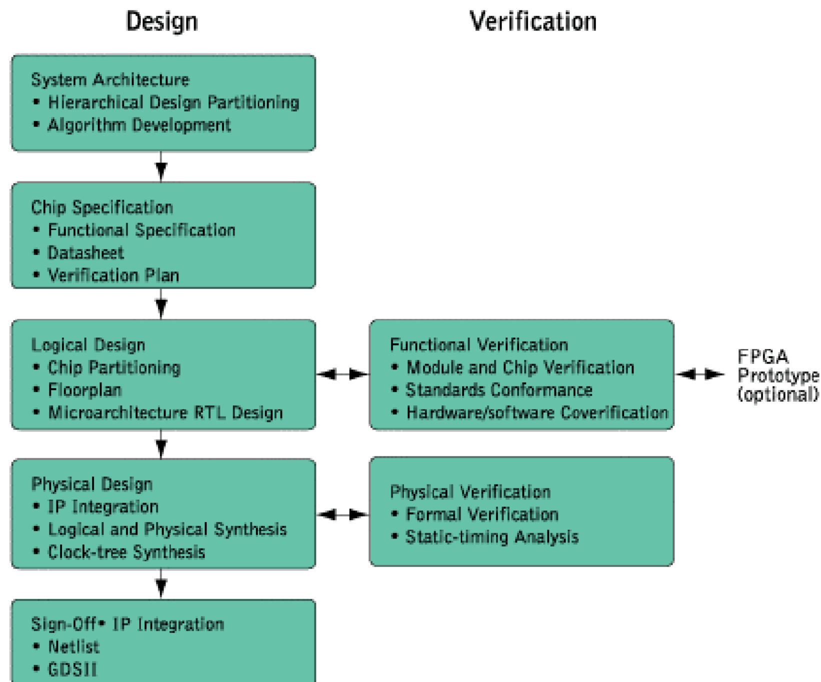


Figure1.1: Design and verification flow: A typical sequence of steps for logical and physical design, and for verification.

1.4.6 Verification Process flow:

Below figure shows the verification flow.

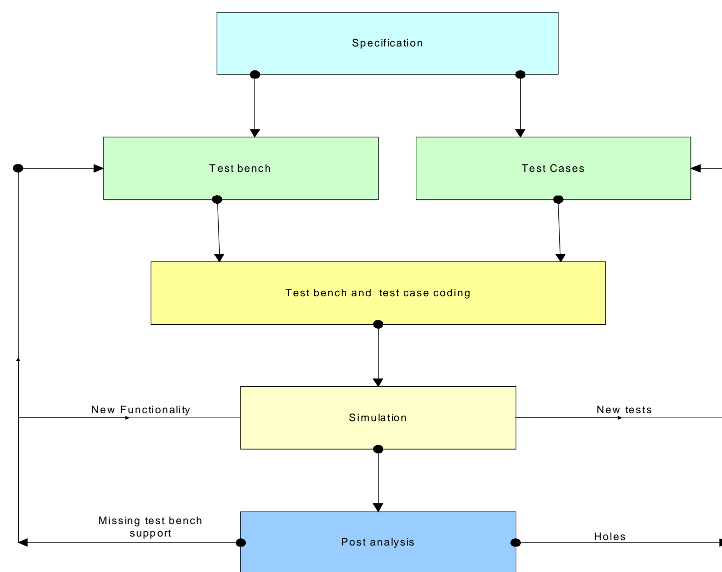


Figure 1.2: Verification process flow.

Verification flow starts with understanding specification of the chip/block under verification. Once the specification is understood, test cases document is prepared, which documents all the possible test cases. Once test case document is done to a level, where 70-80 percent functionality is covered, test bench architecture document is prepared. In the past, test bench architecture document is prepared first and test case document is prepared next. There is a draw back with this style, if test case document show a particular functionality to be verified and if test bench does not support as architecture document was prepared before test cases document. If we have test cases document to refer to, then writing architecture documented becomes much easier, as we know for sure what is expected from the test bench.

1.4.6.1 Test Cases

Identify the test cases from the design specification a simple task for simple cases. Normally requirement in test cases becomes a test case. Anything that specification mentions, "Can do", "will have" becomes a test case. Corner test cases normally take lot of thinking to identify.

1.5 Need of Verification Languages:

Today, project teams build huge verification environments, where verification consumes 40-70% of the resources needed in a typical cycle. Because a verification environment typically contains concurrent mechanisms for controlling traffic streams to device input ports, and for checking outstanding transactions at the output ports, Verilog and VHDL have traditionally been used for building verification environments. Unfortunately, it is widely recognized that for more complex verification environments and problems, these languages do not contain the necessary constructs for modeling the verification environment efficiently.

As a result, many project teams have moved to using higher-level languages such as C and C++ to be more efficient in creating the verification environment. Unfortunately, these general-purpose languages do not have any built-in constructs for modeling hardware concepts such as concurrency, operating in simulation time, or manipulating vectors of various bit widths. Without these constructs, handling device-specific needs such as controlling synchronization between traffic streams, checking correct timing and formatting traffic data are extremely difficult and time-consuming. Project teams often use a mix of HDL

and C/C++ code to attack this verification problem, spending a good deal of time on the interface between the languages. With these problems in mind, what should a verification language look like? It should combine the best features of the most popular HDLs and general-purpose languages:

- Specifying the traffic and traffic parameters in the same terms as the device specification.
- Automatically generating these traffic streams with the ability to target corner cases of the design.
- Storing and checking outstanding transactions.
- Checking protocol adherence.
- Collecting functional feedback on the stimulus and the device under test.
- Automatically responding to feedback from the device during simulation.

With these features in place, verification engineers can focus much sooner on what needs to be verified rather than how to do it (implementing the environment infrastructure)
One way to increase productivity is to raise the level of abstraction. High-level languages, such as C or Pascal, raised the level of abstraction from assembly-level, enabling engineers to become more productive. Similarly, computer languages specifically designed for verification are able to raise the level of abstraction compared to general-purpose simulation languages.

Verilog was designed with a focus on describing low-level hardware structures. It does not provide support for high-level data structures or object-oriented features. VHDL was designed for very large design teams. It strongly encapsulates all information and communicates strictly through well-defined interfaces. This creates an opportunity for verification languages designed to overcome the shortcomings of Verilog and VHDL.
Some popular verification languages Verisity, OpenVera from Synopsys and RAVE from Forte Design. Open-source solutions include the SystemC Verification Library (SCV) from Cadence and Jeda from Juniper Networks. There are also a plethora of homegrown solutions based on Perl, SystemC, C++ or TCL. Verification extensions to the Verilog language are also being added in SystemVerilog.

We have used “e/Specman” verification language for developing “Ethernet eVC” which is a ready made, highly configurable e verification environment suitable for verifying DUTs supporting Ethernet protocol.

1.6 History and advantages of e:

The e language was developed by Verisity as part of its Specman product as a tool for efficiently writing test benches. Like Vera, it is an imperative object-oriented language with concurrency; the ability to generate constrained random values, mechanisms for checking functional (variable value) coverage, and a way to check temporal properties (assertions). Books on e include Palnitkar [41] and Iman and Joshi [30]. The syntax of e is unusual. First, all code must be enclosed in < and > symbols; otherwise it is considered a comment. Unlike C, e declarations are written name: type. The syntax for fields in compound types (e.g., structs) includes particles such as % and !, which indicate when a field is to be driven on the device-under-test and not randomly computed respectively.

- Speeds up the design verification process:
- Automates manual processes
- Provides higher abstraction level than HDLs
- Supports modular, reusable, and extensible code
- Improves usability with graphical interface and debugger
- Improves design quality
- Supports all ranges of tests from directed to random
- Improves test base effectiveness with coverage analysis
- Supports both black box and white-box testing. (In black-box, connection to
- DUT’s interface only. In white-box, looking into the DUT s internals also.)

1.7 Verification Reuse:

“With verification consuming 60-80% of the manpower on complex chip projects, improving verification productivity is an economic necessity", said Moshe Gavrielov, Verisity CEO.Verification reuse directly addresses higher productivity, increased chip quality and overall verification investment. Reusable Methodology is the breakthrough technology required to create reusable verification environments and to ensure that all verification components effectively interoperate. Today's complex chips commonly incorporate many different protocols, interfaces and processors. Assembling appropriate verification

environments requires efficient integration of reusable, plug-and-play verification components.

Achieving reusability requires that all components be built and packaged uniformly. Reusability becomes even more challenging when design teams all over the world create verification components that need to fit together seamlessly. Every aspect of the component, including basic naming conventions and coding styles, debug message conventions, user interfaces, and interactions between components must be standardized in order to assure interoperability. World leader companies in the development faces many verification challenges Verification reuse is essential for our own productivity and to reduce time-to-market for these companies.

The capabilities introduced by Reusable methodology make a significant contribution to meeting these requirements by enabling a framework for reuse.

1.8 eVC (e Verification Component):

One of the key factors for reusing code is arranging it as an independent and easy to use code package. When developing verification code in e, the reusable package is typically organized as an eVC (e Verification Component). An eVC is a verification component. It is ready-to-use configurable environment, typically focusing on a specific protocol or architecture.

Each eVC consists of a complete set of elements for simulating, checking and collecting coverage information for a protocol or architecture. eVC expedite creation of a most efficient test bench for design under test (DUT). They can work with both Verilog and VHDL devices and with all HDL simulators that are supported by Specman.

eVC can be used to create an environment. The eVC interface is viewable and hence can be the basis for user extensions. Maintaining the eVC in its original form facilitates possible upgrades.

eVC implementation is often partially encrypted, especially in commercial eVCs where authors want to protect their intellectual property. Most commercial eVC requires special feature license to enable them. Following is the list of possible kind of eVC:

- Bus-based eVCs (Such PCI and AHB)
- Data-communication eVCs (for example Ethernet, MAC, Data link)
- CPU/DSP eVCs
- Higher level protocol eVCs (TCP/IP, HTTP). These usually sit on top of other eVCs.
- Platform eVCs (that is, an eVC for a specific, reusable SoC platform, into which you plug eVCs of various cores).
- Compliance test-suite eVCs. These are tests (and perhaps coverage definitions and more) that demonstrate compliance to a protocol. For example, there could be a PCI compliance eVC in addition to the basic PCI eVC.
- HW/SW co-verification eVCs, such as an eVC dedicated to verifying a HW/SW environment using a particular RTOS/CPU combination.

A complete verification component handles all the facets involved in verifying a given protocol, interface or processor within the device under test (DUT). This minimally includes the following items (see Figure 1.3):

- Input traffic generator to create stimulus for the DUT (e.g. packets/frames, bus transactions, etc.)
- Bus functional models (BFMs) to drive that traffic, communicating directly with the DUT
- Monitors, scoreboards, and protocol checkers to examine the actual response of the DUT relative to the expected response
- Functional coverage to measure and report on whether the transactions and scenarios defined in the test plan have been covered or not

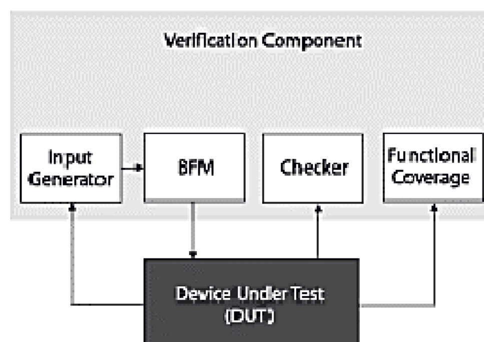


Figure 1.3: Typical verification component

1.9 About Ethernet eVC:

The Ethernet eVC is a ready made highly configurable e Verification Component suitable for DUT supporting the Ethernet Protocol.

All eVC behavior complies with the IEEE Std 802.3, 2000, IEEE Draft P802.3ae/D4.0, SGMII and SMII specification by Cisco systems-1998 and RGMII specification by Hewlet Packard, version2.0. The Ethernet eVC is eRM compliant.

1.10 Ethernet Basics:

The term Ethernet refers to the family of local area network (LAN) implementations that includes three principal categories.

• Ethernet and IEEE 802.3-LAN specifications that operate at 10 Mbps over coaxial cable.

• 100-Mbps Ethernet-A single LAN specifications, also known as Fast Ethernet, that operates at 100 Mbps over twisted-pair cable.

• 1000-Mbps Ethernet-A single LAN specifications, also known as Gigabit Ethernet, that operates at 1000 Mbps (1 Gbps) over fiber and twisted-pair cables.

Ethernet is a comprehensive international standard for Local Area Networks (LANs) employing CSMA/CD as the access method. This standard encompasses several media types and techniques for signal rates from 1 Mb/s to 1000 Mb/s. Ethernet is widely used LAN technology, which allows multiple end stations (such as computers, servers, printers, gateways, to other networks etc.) to exchange data among themselves within a single building or campus. It provides two distinct modes of operation: half duplex and full duplex. Figure 1-1 gives the view of Ethernet network.

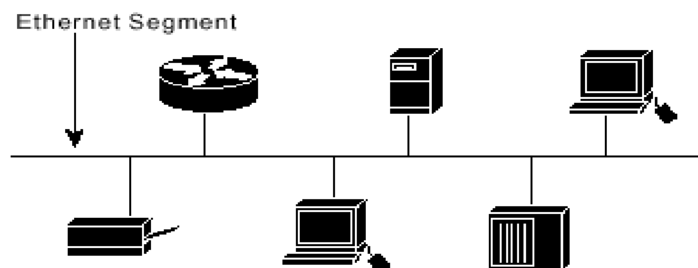


Figure 1.4: An Ethernet network runs CSMA/CD over coaxial cable.

1.10.1 Half duplex operation:

In half duplex mode, the CSMA/CD media access method is the means by which two or more stations share a common transmission medium. To transmit, a station waits for a quiet period on the medium (that is, no other station is transmitting) and then sends intended message in bit-serial form. If after initiating a transmission, the message collides with that of another station then each transition station intentionally transmits for an additional predefined period to ensure the propagation of collision throughout the system. The station remains silent for a random amount of time (back off time) before attempting to transmit again. Half duplex operation can be used with all media and configurations allowed by this standard.

1.10.2 Full duplex operation:

Full duplex operation allows simultaneous communication between a pair of stations using point-to-point media (dedicated channel). Full duplex operation does not require that transmitter defer, nor they monitor or react to receive activity, as there are no contentions on shared medium in this mode. Full duplex mode can be used when all of the following are true: The physical medium is capable of supporting simultaneous transmission and reception without interference.

There are exactly two stations connected with a full duplex point-to-point link. Since there is no contention for use of a shared medium, the multiple access (i.e., CSMA/CD) algorithms are unnecessary.

Both the stations on the LAN are capable of, and have been configured to use, full duplex operation.

The most common configuration envisioned for full duplex operation consists of central bridge (also known as a switch) with a dedicated LAN connecting each bridge port to a single device. Figure 1-2 showing point-to-point link.

Figure 1.5: Example Point-to-Point Interconnection

1.10.3 Ethernet Network Elements:

Ethernet LANs consist of network nodes and interconnecting media. The network nodes fall into two major classes:

1.10.3.1 Data terminal equipment (DTE): Devices that are either the source or the destination of data frames. DTEs are typically devices such as PCs, workstations, file servers, or print servers that, as a group, are all often referred to as end stations.

1.10.3.2 Data communication equipment (DCE): Intermediate network devices that receive and forward frames across the network. DCEs may be either standalone devices such as repeaters, network switches, and routers, or communications interface units such as interface cards and modems.
The current Ethernet media options include two general types of copper cable: unshielded twisted-pair (UTP) and shielded twisted-pair (STP), plus several types of optical fiber cable.

1.11 Verification Environment for Ethernet eVC:

Steps for building verification environment:

For building verification environment, steps which are very common and applicable to any other verification environment as follows:

- Generating traffic streams
- Driving traffic into the design (stimuli)
- Checking these data streams
- Checking protocols and timing
- Tracking progress
- Modifying the environment due to spec changes or product derivatives
- Writing scenarios

Step 1: Generating traffic streams

When building a verification environment, the verification engineer often starts by modeling the device input stimulus. In Verilog, the designer is limited in how to model this traffic because of the lack of high-level data structures and the notion of dynamic lists. As a result, sequences of traffic items are often represented as a series of task calls to initialize the frame, build its header, and payload. This ?cut-and-paste? code is lengthy, hard to read and hard to maintain (if there is need to add another attribute passed to all task calls).

The verification engineer often needs constructs better suited for modeling and manipulating frame sequences. C provides some of these. However, it becomes apparent that C was not dev eloped with hardware in mind as soon as bit vectors need to be specified on non-four byte boundaries and manipulated. Unfortunately, the engineer still has to write the functions to manipulate the lists and, to use memory efficiently, now has to worry about correctly allocating and de-allocating memory and handling pointers. These memory problems can result in hours of chasing bus errors and segmentation faults.

The e language provides constructs for modeling traffic streams with built-in functions for generating them and automatically takes care of memory allocation and garbage collection (memory de-allocation). Also in e, there is no need to create generation functions for each struct. The generator is built-in, and once the structs are declared, meaningful stimuli can be generated. Defining a field ('frames' in the above example), which is a ?list of frames?, is enough to have a sequence of frames generated. The built in generator performs memory allocation dynamically, and a built-in garbage collection takes care of de-allocation.

Step 2: Driving stimuli into the device

With the data structures in place, the engineer has to consider how data will flow though the verification environment. Typically, a task or function will drive data into each of the device's input ports and a task or function will pull data from each of the device's output ports.

In Verilog, the device specific synchronization for that port is fairly easy because the engineer has direct control over the signals. However, many other functions are typically needed to synchronize the basic traffic flow, format the data at each end, and print the data structures for debugging and post-run checking.

C and C++ help with organizing and formatting data, but the benefit is eaten away because these general-purpose languages have no concept of simulation time or hardware signals. Special code must be written t o interface with the specific Verilog Programming Language Interface (PLI) to drive and sample simulation signals at the appropriate simulation time. Moreover, PLI-related code often needs to be tailored for different simulators. In addition, traffic structures built in C/C++ must be manually converted to the bit, byte or word format the device expects to receive and send.

The e language in this case combines the benefit of both worlds. It has the knowledge of simulation time, constructs for concurrency and built-in controllability, and observability of the simulation signals, regardless of the simulator. Further, each structure in the environment automatically has pre-defined pack and unpack functions for converting it to a bit stream, or vice versa, from bit stream into the data structure format and fields.

Step 3: Checking data streams

To check that the device works correctly, monitoring tasks or functions need to be written. In the case of an Ethernet device, such tasks would pull data from the output ports and compare it with the expected data. The expected data is often determined within a self-checking environment by storing the outstanding transactions, or through a reference model of the device.

As with traffic streams, Verilog's lack of built-in lists makes it impossible to create dynamic scoreboards to verify the data integrity and routing of outstanding transactions. This inflexibility leads to built-in list sizes that use up memory and create assumptions in the code that are difficult to maintain if this code is to be portable.

Built-in lists and list functions, recursive data structure comparison, and automatic memory allocations are key features in the e verification language. These constructs make it much easier to implement data integrity checking

Step 4: Checking protocols and timing - using assertions

Because Verilog and VHDL are directly tied to the simulator, it is easy to interact with the device to get timing information. Unfortunately, most of the interesting timing checks are not straightforward and contain multiple edge dependencies. A timing check might be easy to specify in an English specification, but it is another story to write it in procedural code, be it Verilog, VHDL, C or C++. As a result, in the past few years, declarative assertions (in languages such as PSL/Sugar) have become an appealing approach.

The e language includes a built-in assertion language, similar in expressiveness to PSL/Sugar. Constructs to capture timing scenarios, automated capabilities to observe the device and checking constructs to relate timing scenarios are all built-in. Such an assertion language makes it easy to specify and combine complex timing scenarios that can be used for even the most difficult protocol checking.

Step 5: Tracking progress

A key requirement in verification is having a reliable metric that shows the progress towards hitting all the verification goals. Functional coverage, a metric that tracks which functionality of the device was verified, is being recognized as a highly reliable measure. Neither Verilog, VHDL nor C/C++ have the notion of functional coverage, causing many project teams to create elaborate combinations of log files, parsed by various scripts which then produce some summary reports. This enables very limited capabilities, and yet another maintenance burden on the implementers of the verification environment.

The e verification language has built-in functional coverage constructs, supporting simple value coverage, value-transition coverage and cross-coverage.

Step 6: Specification changes and product derivatives - extensibility

Once the environment is in place, the engineer might need to extend or modify it to accommodate changes or updates to the specification. Verilog, VHDL and C clearly fall short in this area because they provide no constructs to incorporate any changes: modifications need to be done within the original code. Changing the original code often results in bugs because the intent of the original code is not clearly known.

The e verification language offers the notion of extensibility, which is a key feature of an Aspect-Oriented Programming (AOP) language, and provides the constructs necessary to modify or change the functionality of the environment without having to change the original code in any form.

Step 7: Writing scenarios

Once the verification environment is in place, the project team now needs to focus on writing scenarios to verify that the device behaves according to specification.

The problem with Verilog and C is that you have to write explicit randomization code for each attribute the designer wants to randomize. There is also often interest in mechanisms to weight particular attributes towards values that denote typical or corner case scenarios. Such mechanisms in Verilog and C are typically non-obvious, maintenance burdens.

‘e’ provides not only the built-in capability to randomly generate any data structure - simple or complex - automatically with weighted distribution, but it also provides constructs for constraining the generated values within acceptable ranges. Additionally, e provides mechanisms to easily layer test-specific constraints for particular tests, and take run-time feedback from the device to focus this random test into hard-to-reach corner cases. Combining this generation capability with the temporal assertion language gives the verification engineer a powerful capability for identifying complex, internal corner case scenarios and generating specific traffic to create conflicts deep within the device.

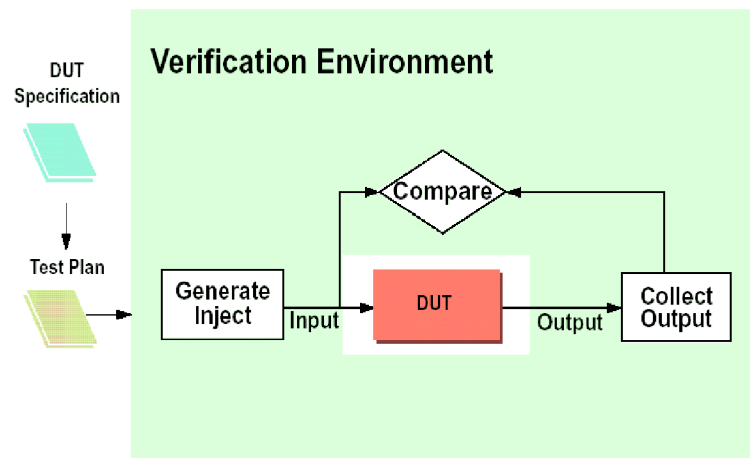


Figure 1.6: Verification Environment

Chapter 2

Review of Literature

2.1 The IEEE 802.3 Logical Relationship to the ISO Reference Model

Figure 2-1 shows the IEEE 802.3 logical layers and their relationship to the OSI reference model. As with all IEEE 802 protocols, the ISO data link layer is divided into two IEEE 802 sub layers, the Media Access Control (MAC) sub layer and the MAC-client sub layer. The IEEE 802.3 physical layer corresponds to the ISO physical layer.

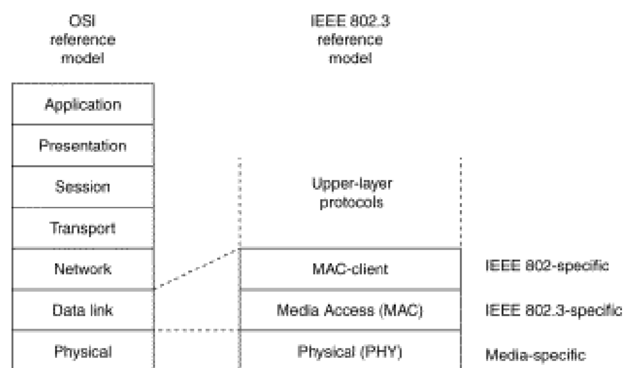


Figure 2.1: Ethernet's Logical Relationship to the ISO Reference Model

The MAC-client sub layer may be one of the following:

- Logical Link Control (LLC), if the unit is a DTE. This sublayer provides the interface between the Ethernet MAC and the upper layers in the protocol stack of the end station. The LLC sublayer is defined by IEEE 802.2 standards.
- Bridge entity, if the unit is a DCE. Bridge entities provide LAN-to-LAN interfaces between LANs that use the same protocol (for example, Ethernet to Ethernet) and also between different protocols (for example, Ethernet to Token Ring). Bridge entities are defined by IEEE 802.1 standards.

Because specifications for LLC and bridge entities are common for all IEEE 802 LAN protocols, network compatibility becomes the primary responsibility of the particular network protocol. Figure 2-2 shows different compatibility requirements imposed by the MAC and physical levels for basic data communication over an Ethernet link.

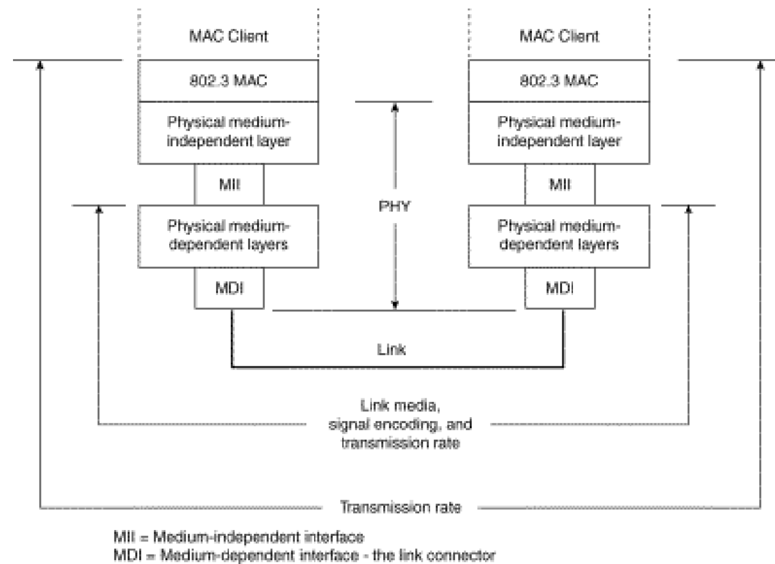


Figure 2.2: MAC and Physical Layer Compatibility Requirements for Basic Data Communication

The MAC layer controls the node's access to the network media and is specific to the individual protocol. All IEEE 802.3 MACs must meet the same basic set of logical requirements, regardless of whether they include one or more of the defined optional protocol extensions. The only requirement for basic communication (communication that does not require optional protocol extensions) between two network nodes is that both MACs must support the same transmission rate.

The 802.3 physical layer is specific to the transmission data rate, the signal encoding, and the type of media interconnecting the two nodes. Gigabit Ethernet, for example, is defined to operate over either twisted-pair or optical fiber cable, but each specific type of cable or signal-encoding procedure requires a different physical layer implementation.

2.2 The Ethernet MAC Sublayer

The MAC sublayer has two primary responsibilities:

- Data encapsulation, including frame assembly before transmission, and frame parsing/error detection during and after reception
- Media access control, including initiation of frame transmission and recovery from transmission failure

2.3 Media Access Control frame structure:

This section gives the detail for the communication system using the CSMA/CD MAC. It defines the various components of MAC frame.

2.3.1 MAC frame format:

Figure 2-3 shows the nine fields of the frame: the preamble, SFD (start frame delimiter), the addresses of the frame's source and destination, length or type field to indicate the length or protocol type of the following field that contains the MAC client data, a field that contains padding if required, the frame check sequence field containing a cyclic redundancy check value to detects error in received frame, and the extension field if required (for 1000 Mb/s half duplex operation only). Of these nine fields all are of fixed size except for the data, pad and extension fields which may contain an integer number of octets between the minimum and maximum values that are determined by specific implementation of the CSMA/CD MAC for a particular interface.

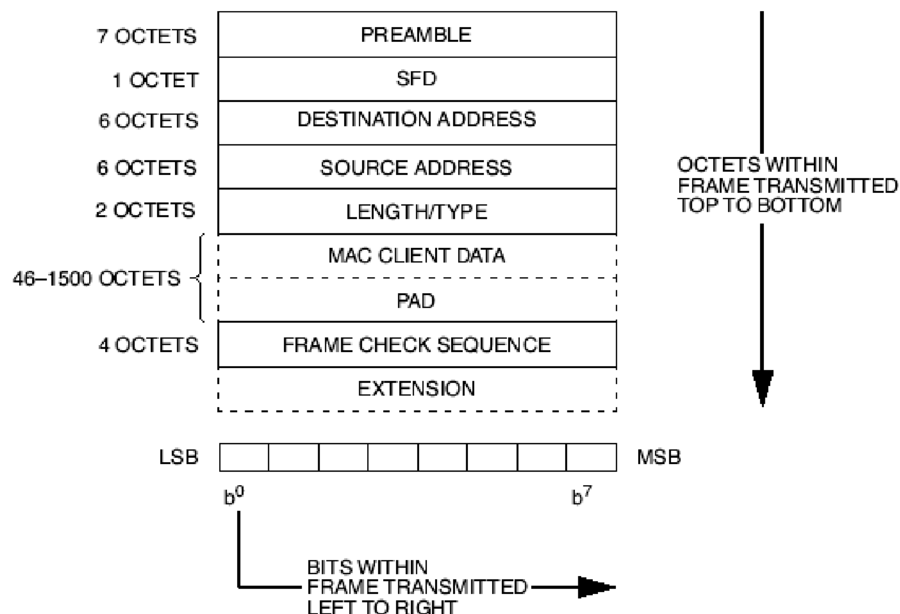


Figure 2-3: MAC Frame format

2.3.2 MAC frame elements:

2.3.2.1 Preamble field:

The preamble field is a 7-octate field that is used to achieve steady-state synchronization with received frame's timing.

2.3.2.2 Start Frame Delimiter (SFD):

The SFD field is the sequence 10101011. It immediately follows the preamble pattern and indicates the start of frame.

2.3.2.3 Address Fields:

Each MAC frame shall contain two address fields: the Destination address and the Source address field, in the order. The destination address field specifies the address(s) for which the frame is intended. The source address field identifies the station from which the frame was initiated. The representation of address field shall be as follows (see figure 2.4):

I/G = 0 INDIVIDUAL ADDRESS
I/G = 1 GROUP ADDRESS
U/L = 0 GLOBALLY ADMINISTERED ADDRESS
U/L = 1 LOCALLY ADMINISTERED ADDRESS

Figure 2.4: Address designation

Note Individual addresses are also known as unicast addresses because they refer to a single MAC and are assigned by the NIC manufacturer from a block of addresses allocated by the IEEE. Group addresses (multicast addresses) identify the end stations in a workgroup and are assigned by the network manager. A special group address (all 1 the broadcast address) indicates all stations on the network.

- **Destination address (DA):** Consists of 6 bytes. The DA field identifies which station(s) should receive the frame. The left-most bit in the DA field indicates whether the address is an individual address (indicated by a 0) or a group address (indicated by a 1). The second bit from the left indicates whether the DA is globally administered (indicated by a 0) or locally administered (indicated by a 1). The remaining 46 bits are a uniquely assigned value that identifies a single station, a defined group of stations, or all stations on the network.
- **Source addresses (SA):** Consists of 6 bytes. The SA field identifies the sending station. The SA is always an individual address and the left-most bit in the SA field is always 0.
- **Length/Type:** Consists of 4 bytes. This field indicates either the number of MAC-client data bytes that are contained in the data field of the frame, or the frame type ID if the frame is assembled using an optional format. If the Length/Type field value is less than or equal to 1500, the number of LLC bytes in the Data field is equal to the Length/Type field value. If the Length/Type field value is greater than 1536, the frame is an optional type frame, and the Length/Type field value identifies the particular type of frame being sent or received.
- **Data:** Is a sequence of *n* bytes of any value, where *n* is less than or equal to 1500. If the length of the Data field is less than 46, the Data field must be extended by adding a filler (a pad) sufficient to bring the Data field length to 46 bytes.
- **Frame check sequence (FCS):** Consists of 4 bytes. This sequence contains a 32-bit cyclic redundancy check (CRC) value, which is created by the sending MAC and is recalculated by the receiving MAC to check for damaged frames. The FCS is generated over the DA, SA, Length/Type, and Data fields.

2.3.4 Frame Transmission

Whenever an end station MAC receives a transmit-frame request with the accompanying address and data information from the LLC sublayer, the MAC begins the transmission sequence by transferring the LLC information into the MAC frame buffer.

- The preamble and start-of-frame delimiter are inserted in the PRE and SOF fields.
- The destination and source addresses are inserted into the address fields.

- The LLC data bytes are counted, and the number of bytes is inserted into the Length/Type field.
- The LLC data bytes are inserted into the Data field. If the number of LLC data bytes is less than 46, a pad is added to bring the Data field length up to 46.
- An FCS value is generated over the DA, SA, Length/Type, and Data fields and is appended to the end of the Data field.

After the frame is assembled, actual frame transmission will depend on whether the MAC is operating in half-duplex or full-duplex mode.

The IEEE 802.3 standard currently requires that all Ethernet MACs support half-duplex operation, in which the MAC can be either transmitting or receiving a frame, but it cannot be doing both simultaneously. Full-duplex operation is an optional MAC capability that allows the MAC to transmit and receive frames simultaneously.

2.4 Half-Duplex Transmission: The CSMA/CD Access Method

The CSMA/CD protocol was originally developed as a means by which two or more stations could share a common media in a switch-less environment when the protocol does not require central arbitration, access tokens, or assigned time slots to indicate when a station will be allowed to transmit. Each Ethernet MAC determines for itself when it will be allowed to send a frame. The CSMA/CD access rules are summarized by the protocol's acronym:

- **Carrier sense:** Each station continuously listens for traffic on the medium to determine when gaps between frame transmissions occur.
- **Multiple access: S**tations may begin transmitting any time they detect that the network is quiet (there is no traffic).
- **Collision detect:** If two or more stations in the same CSMA/CD network (collision domain) begin transmitting at approximately the same time, the bit streams from the transmitting stations will interfere (collide) with each other, and both transmissions will be unreadable. If that happens, each transmitting station must be capable of detecting that a collision has occurred before it has finished sending its frame. Each must stop transmitting as soon as it has detected the collision and then must wait

a quasirandom length of time (determined by a back-off algorithm) before attempting to retransmit the frame.

The worst-case situation occurs when the two most-distant stations on the network both need to send a frame and when the second station does not begin transmitting until just before the frame from the first station arrives. The collision will be detected almost immediately by the second station, but it will not be detected by the first station until the corrupted signal has propagated all the way back to that station. The maximum time that is required to detect a collision (the collision window, or "slot time") is approximately equal to twice the signal propagation time between the two most-distant stations on the network.

This means that both the minimum frame length and the maximum collision diameter are directly related to the slot time. Longer minimum frame lengths translate to longer slot times and larger collision diameters; shorter minimum frame lengths correspond to shorter slot times and smaller collision diameters.

The trade-off was between the need to reduce the impact of collision recovery and the need for network diameters to be large enough to accommodate reasonable network sizes. The compromise was to choose a maximum network diameter (about 2500 meters) and then to set the minimum frame length long enough to ensure detection of all worst-case collisions.

The compromise worked well for 10 Mbps, but it was a problem for higher data-rate Ethernet developers. Fast Ethernet was required to provide backward compatibility with earlier Ethernet networks, including the existing IEEE 802.3 frame format and error-detection procedures, plus all applications and networking software running on the 10-Mbps networks.

Although signal propagation velocity is essentially constant for all transmission rates, the time required to transmit a frame is inversely related to the transmission rate.

At 100 Mbps, a minimum-length frame can be transmitted in approximately one-tenth of the defined slot time, and the transmitting stations would not likely detect any collision that occurred during the transmission. This, in turn, meant that the maximum network diameters specified for 10-Mbps networks could not be used for 100-Mbps networks. The solution for Fast Ethernet was to reduce the maximum network diameter by approximately a factor of 10 (to a little more than 200 meters).

The same problem also arose during specification development for Gigabit Ethernet, but decreasing network diameters by another factor of 10 (to approximately 20 meters) for 1000-Mbps operation was simply not practical. This time, the developers elected to maintain approximately the same maximum collision domain diameters as 100-Mbps networks and to increase the apparent minimum frame size by adding a variable-length nondata extension field to frames that are shorter than the minimum length (the extension field is removed during frame reception).

Figure 2.5 shows the MAC frame format with the gigabit extension field, and Table 2.1 shows the effect of the trade-off between the transmission data rate and the minimum frame size for 10-Mbps, 100-Mbps, and 1000-Mbps Ethernet.

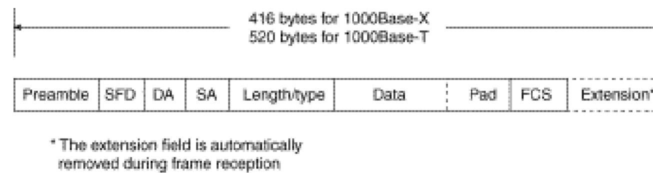


Figure 2.5: MAC Frame with Gigabit Carrier Extension

Table 2.1: Limits for Half-Duplex Operation

Parameter	10 Mbps	100 Mbps	1000 Mbps
Minimum frame size	64 bytes	64 bytes	520 bytes[1] (with extension field added)
Maximum collision diameter, DTE to DTE	100 meters UTP	100 meters UTP 412 meters fiber	100 meters UTP 316 meters fiber
Maximum collision diameter with repeaters	2500 meters	205 meters	200 meters
Maximum number of repeaters in network path	5	2	1

520 bytes apply to 1000Base-T implementations. The minimum frame size with extension field for 1000Base-X is reduced to 416 bytes because 1000Base-X encodes and transmits 10 bits for each byte.

Another change to the Ethernet CSMA/CD transmit specification was the addition of frame bursting for gigabit operation. Burst mode is a feature that allows a MAC to send a short sequence (a burst) of frames equal to approximately 5.4 maximum-length frames without having to relinquish control of the medium. The transmitting MAC fills each interframe interval with extension bits, as shown in Figure 2.6, so that other stations on the network will see that the network is busy and will not attempt transmission until after the burst is complete.

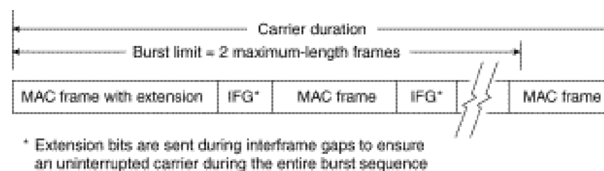


Figure 2-6: A Gigabit Frame-Burst Sequence

If the length of the first frame is less than the minimum frame length, an extension field is added to extend the frame length to the value indicated in Table 2-1. Subsequent frames in a frame-burst sequence do not need extension fields, and a frame burst may continue as long as the burst limit has not been reached. If the burst limit is reached after a frame transmission has begun, transmission is allowed to continue until that entire frame has been sent. Frame extension fields are not defined, and burst mode is not allowed for 10 Mbps and 100 Mbps transmission rates.

2.5 Full-Duplex Transmission: An Optional Approach to Higher Network Efficiency

Full-duplex operation is an optional MAC capability that allows simultaneous two-way transmission over point-to-point links. Full duplex transmission is functionally much simpler than half-duplex transmission because it involves no media contention, no collisions, no need to schedule retransmissions, and no need for extension bits on the end of short frames. The result is not only more time available for transmission, but also an effective doubling of the

link bandwidth because each link can now support full-rate, simultaneous, two-way transmission.

Transmission can usually begin as soon as frames are ready to send. The only restriction is that there must be a minimum-length inter frame gap between successive frames, as shown in Figure 2.7, and each frame must conform to Ethernet frame format standards.

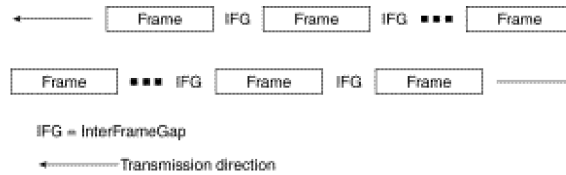


Figure 2.7: Full Duplex Operation Allows Simultaneous Two-Way Transmission on the Same Link

2.5.1 Flow Control

Full-duplex operation requires concurrent implementation of the optional flow-control capability that allows a receiving node (such as a network switch port) that is becoming congested to request the sending node (such as a file server) to stop sending frames for a selected short period of time. Control is MAC-to-MAC through the use of a pause frame that is automatically generated by the receiving MAC. If the congestion is relieved before the requested wait has expired, a second pause frame with a zero time-to-wait value can be sent to request resumption of transmission. An overview of the flow control operation is shown in Figure 2.8.

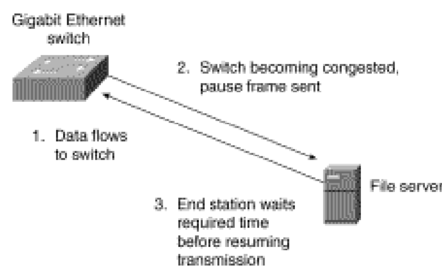


Figure 2.8: An Overview of the IEEE 802.3 Flow Control Sequence

The full-duplex operation and its companion flow control capability are both options for all Ethernet MACs and all transmission rates. Both options are enabled on a link-by-link basis, assuming that the associated physical layers are also capable of supporting full-duplex operation.

Pause frames are identified as MAC control frames by an exclusive assigned (reserved) length/type value. They are also assigned a reserved destination address value to ensure that an incoming pause frame is never forwarded to upper protocol layers or to other ports in a switch.

2.5.2 Frame Reception

Frame reception is essentially the same for both half-duplex and full-duplex operations, except that full-duplex MACs must have separate frame buffers and data paths to allow for simultaneous frame transmission and reception.

Frame reception is the reverse of frame transmission. The destination address of the received frame is checked and matched against the station's address list (its MAC address, its group addresses, and the broadcast address) to determine whether the frame is destined for that station. If an address match is found, the frame length is checked and the received FCS is compared to the FCS that was generated during frame reception. If the frame length is okay and there is an FCS match, the frame type is determined by the contents of the Length/Type field. The frame is then parsed and forwarded to the appropriate upper layer.

2.6 The Ethernet Physical Layers

Because Ethernet devices implement only the bottom two layers of the OSI protocol stack, they are typically implemented as network interface cards (NICs) that plug into the host device's motherboard. The different NICs are identified by a three-part product name that is based on the physical layer attributes.

The naming convention is a concatenation of three terms indicating the transmission rate, the transmission method, and the media type/signal encoding. For example, consider this:

- 10Base-T = 10 Mbps, baseband, over two twisted-pair cables
- 100Base-T2 = 100 Mbps, baseband, over two twisted-pair cables
- 100Base-T4 = 100 Mbps, baseband, over four-twisted pair cables
- 1000Base-LX = 100 Mbps, baseband, long wavelength over optical fiber cable

A question sometimes arises as to why the middle term always seems to be "Base." Early versions of the protocol also allowed for broadband transmission (for example, 10Broad), but broadband implementations were not successful in the marketplace. All current Ethernet implementations use baseband transmission.

2.7 Encoding for Signal Transmission

In baseband transmission, the frame information is directly impressed upon the link as a sequence of pulses or data symbols that are typically attenuated (reduced in size) and distorted (changed in shape) before they reach the other end of the link. The receiver's task is to detect each pulse as it arrives and then to extract its correct value before transferring the reconstructed information to the receiving MAC.

- Filters and pulse-shaping circuits can help restore the size and shape of the received waveforms, but additional measures must be taken to ensure that the received signals are sampled at the correct time in the pulse period and at same rate as the transmit clock:
- The receive clock must be recovered from the incoming data stream to allow the receiving physical layer to synchronize with the incoming pulses.

Compensating measures must be taken for a transmission effect known as baseline wander. Clock recovery requires level transitions in the incoming signal to identify and synchronize on pulse boundaries. The alternating 1s and 0s of the frame preamble were designed both to indicate that a frame was arriving and to aid in clock recovery. However, recovered clocks can drift and possibly lose synchronization if pulse levels remain constant and there are no transitions to detect (for example, during long strings of 0s).

Baseline wanders results because Ethernet links are AC-coupled to the transceivers and because AC coupling is incapable of maintaining voltage levels for more than a short time. As a result, transmitted pulses are distorted by a droop effect similar to the exaggerated example shown in Figure 2-9. In long strings of either 1s or 0s, the droop can become so severe that the voltage level passes through the decision threshold, resulting in erroneous sampled values for the affected pulses.

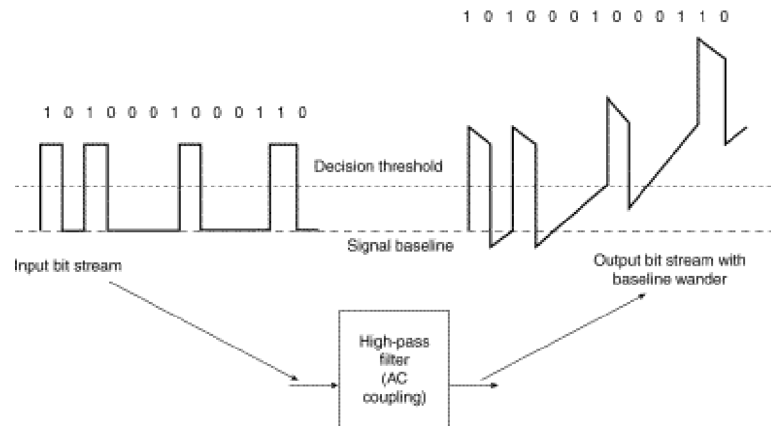


Figure 2.9: A Concept Example of Baseline Wander

Fortunately, encoding the outgoing signal before transmission can significantly reduce the effect of both these problems, as well as reduce the possibility of transmission errors. Early Ethernet implementations, up to and including 10Base-T, all used the Manchester encoding method, shown in Figure 2-10. Each pulse is clearly identified by the direction of the midpulse transition rather than by its sampled level value.

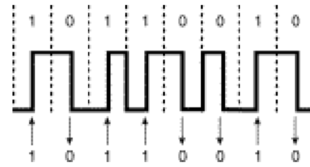


Figure 2.10: Transition-Based Manchester Binary Encoding

Unfortunately, Manchester encoding introduces some difficult frequency-related problems that make it unsuitable for use at higher data rates. Ethernet versions subsequent to 10Base- T all use different encoding procedures that include some or all of the following techniques:

- **Using data scrambling:** A procedure that scrambles the bits in each byte in an orderly (and recoverable) manner. Some 0s are changed to 1s, some 1s are changed to 0s, and some bits are left the same. The result is reduced run-length of same-value bits, increased transition density, and easier clock recovery.
- **Expanding the code space:** A technique that allows assignment of separate codes for data and control symbols (such as start-of-stream and end-of-stream delimiters, extension bits, and so on) and that assists in transmission error detection.

- **Using forward error-correcting codes:** An encoding in which redundant information is added to the transmitted data stream so that some types of transmission errors can be corrected during frame reception.

Note Forward error-correcting codes are used in 1000Base-T to achieve an effective reduction in the bit error rate. Ethernet protocol limits error handling to detection of bit errors in the received frame. Recovery of frames received with uncorrectable errors or missing frames is the responsibility of higher layers in the protocol stack.

2.8 The 802.3 Physical Layer Relationship to the ISO Reference Model

Although the specific logical model of the physical layer may vary from version to version, all Ethernet NICs generally conform to the generic model shown in Figure 2.11.

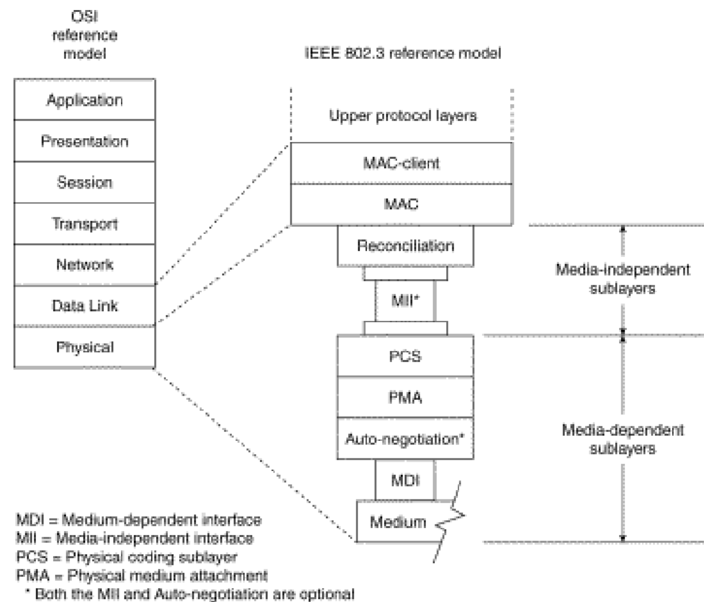


Figure 2.11: The Generic Ethernet Physical Layer Reference Model

The physical layer for each transmission rate is divided into sub layers that are independent of the particular media type and sub layers that are specific to the media type or signal encoding.

- The reconciliation sublayer and the optional media-independent interface (MII in 10-Mbps and 100-Mbps Ethernet, GMII in Gigabit Ethernet) provide the logical connection between the MAC and the different sets of media-dependent layers. The MII and GMII are defined with separate transmit and receive data paths that are bit-serial for 10-Mbps implementations, nibble-serial (4 bits wide) for 100-Mbps

implementations, and byte-serial (8 bits wide) for 1000-Mbps implementations. The media-independent interfaces and the reconciliation sublayer are common for their respective transmission rates and are configured for full-duplex operation in 10Base-T and all subsequent Ethernet versions.

- The media-dependent physical coding sublayer (PCS) provides the logic for encoding, multiplexing, and synchronization of the outgoing symbol streams as well symbol code alignment, demultiplexing, and decoding of the incoming data.

The physical medium attachment (PMA) sublayer contains the signal transmitters and receivers (transceivers), as well as the clock recovery logic for the received data streams.

- The medium-dependent interface (MDI) is the cable connector between the signal transceivers and the link.
- The Auto-negotiation sublayer allows the NICs at each end of the link to exchange information about their individual capabilities, and then to negotiate and select the most favorable operational mode that they both are capable of supporting. Auto-negotiation is optional in early Ethernet implementations and is mandatory in later versions.
- Depending on which type of signal encoding is used and how the links are configured, the PCS and PMA may or may not be capable of supporting full-duplex operation.

2.9 Ethernet Interfaces:

Ethernet Interfaces basically defines the logical and electrical characteristics for the data transmission between MAC and PHY layer. One such interface is Gigabit Media Independent Interface (GMII). (See Figure 2.12)

Gigabit Media Independent Interface (GMII):

This interface has following characteristics:

- It is capable of supporting 1000 Mb/s operation.
- Data and delimiters are synchronous to clock references.
- It provides independent eight-bit-wide transmit and receive data paths.
- It provides a simple management interface.
- It uses signal levels, compatible with common CMOS digital ASIC processes.
- It provides full and half duplex operation.
- Supports 1000 Mb/s speed and clock frequency is 125 MHz.

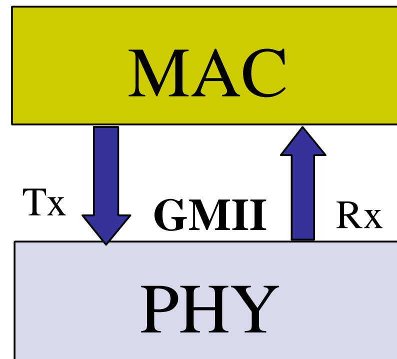


Figure 2.12: Transmission between MAC & PHY.

2.9.1 Signal Description:

2.9.1.1 TX_EN (transmit enable):

TX_EN in combination with TX_ER indicates the Reconciliation sublayer is presenting data on the GMII for transmission. It shall be asserted by the Reconciliation sublayer synchronously with the first octet of the preamble and shall remain asserted while all octets to be transmitted are presented to the GMII. TX_EN shall be negated prior to the first rising edge of clock following the final data octet of a frame. TX_EN is driven by the Reconciliation sublayer and shall transition synchronously with respect to the clock.

2.9.1.2 TXD (transmit data):

TXD is a bundle of eight data signals (TXD<7:0>) that are driven by the Reconciliation sublayer. TXD<7:0> shall transition synchronously with respect to the clock. For each clock period in which TX_EN is asserted and TX_ER is de-asserted, data are presented on TXD<7:0> to the PHY for transmission. TXD<0> is the lease significant bit. While TX_EN and TX_ER are both de-asserted, TXD<7:0> shall have no effect upon the PHY. Table 2.2 specifies the permissible encodings of TXD<7:0>, TX_ER, and TX_EN.

Table 2.2: Permissible encoding of TXD<7:0>, TX_ER, and TX_EN

TX_EN	TX_ER	TXD<7:0>	Description
0	0	00 through FF	Normal inter-frame
0	1	00 through 0E	Reserved
0	1	0F	Carrier Extend
0	1	10 through 1E	Reserved
0	1	1F	Carrier Extend error
0	1	20 through FF	Reserved
1	0	00 through FF	Normal data transmission
1	1	00 through FF	Transmit error propagation
NOTE- Values in TXD<7:0> column are in hexadecimal.			

2.9.1.3 TX_ER (transmit coding error):

TX_ER is driven by the Reconciliation Sublayer and shall transition synchronously with respect to the clock. When TX_ER is asserted for one or more TX_CLK periods while TX_EN is also asserted, the PHY shall emit one or more code-groups that are not part of the valid data or delimiter set somewhere in the frame being transmitted. The relative position of the error within the frame need not be preserved.

2.9.1.4 RX_DV (receive data valid):

RX_DV is driven by the PHY to indicate that the PHY is presenting recovered and decoded data on the RXD<7:0> bundle. RX_DV shall transition synchronously with respect to the RX_CLK. RX_DV shall be asserted continuously from the first recovered octet of the frame through the final recovered octet. In order for a received frame to be correctly interpreted by the Reconciliation sublayer and the MAC sublayer, RX_DV must encompass the frame, starting no later than Start Frame Delimiter (SFD) and excluding any End-of-Frame delimiter.

2.9.1.5 RXD (receive data):

RXD is a bundle of eight data signals (RXD <7:0>) that are driven by the PHY. RXD<7:0> shall transition synchronously with respect to RX_CLK. For each RX_CLK period in which RX_DV asserted, RXD<7:0> transfer eight bits of recovered data from the PHY to the Reconciliation sublayer. RXD<0> is the least significant bit. While RX_DV is de-asserted, the PHY may provide a False Carrier Indication by asserting the RX_ER signal while driving the specific value listed in Table 2.3.

Table 2.3: Permissible encoding of RXD<7:0>, RX_ER, and RX_DV

RX_DV	**RX_ER**	**RXD<7:0>**	**Description**
0	0	00 through FF	Normal inter-frame
0	1	00	Normal inter-frame
0	1	01 through 0D	Reserved
0	1	0E	False Carrier indication
0	1	0F	Carrier Extend
0	1	10 through 1E	Reserved
0	1	1F	Carrier Extend Error
0	1	20 through FF	Reserved
1	0	00 through FF	Normal data reception
1	1	00 through FF	Data reception error
NOTE- Values in RXD<7:0> column are in hexadecimal.			

2.9.1.6 RX_ER (receive error):

RX_ER is driven by the PHY and shall transition synchronously with respect to RX_CLK. When RX_DV is asserted, RX_ER shall be asserted for one or more RX_CLK periods to indicate to the Reconciliation sublayer that an error (e.g. a coding error, or another error that the PHY is capable of detecting that may otherwise be undetectable at the MAC sublayer) was detected somewhere in the frame presently being transferred from the PHY to the Reconciliation sublayer.

2.9.2 Reduced Gigabit Media Independent Interface (RGMII):

RGMII is intended to be an alternative to the IEEE802.3u MII (Media Independent Interface), the IEEE802.3z GMII. The principle objective is to reduce the number of pins required to interconnect the MAC and the PHY from a maximum of 28 pins to 12 pins in a cost effective and technology independent manner. In order to accomplish this objective, the data paths and all associated control signals will be reduced and control signals will be multiplexed together and both edges of the clock will be used. For Gigabit operation, the clock will operate at 125 MHz and for 10/100 Mbps operation, the clocks will operate at 2.5 MHz or 25 MHz respectively.

2.9.2.3 System Diagram:

Figure 2.13 shows system level diagram.

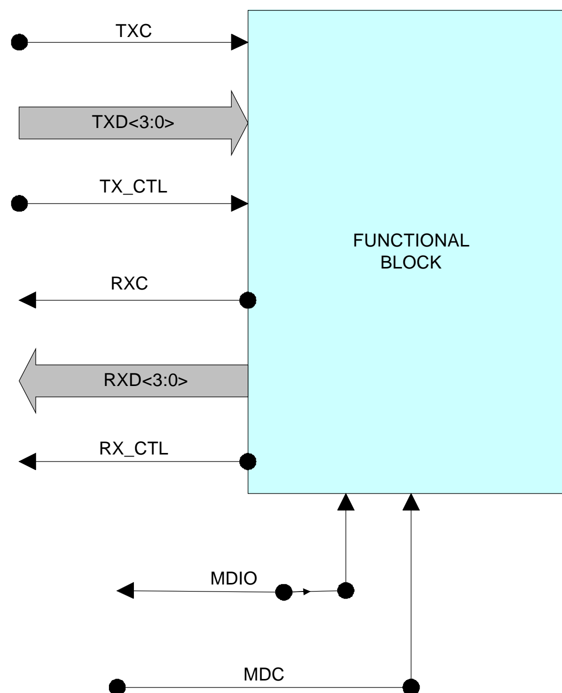


Figure 2.13: System level diagram of Reduced Gigabit Media Independent Interface (RGMII)

2.9.4 Signal Definition:

Following table gives the signal definition:

Table 2.4: Signal description of RGMII.

Signal Name	RGMII	Description
TXC	MAC	The transmit reference clock will be 125 MHz, 25 MHz, 2.5 MHz depending on speed.
TD<3:0>	MAC	In RGMII mode, bits 3:0 on positive edge of TXC, bits 7:4 on negative edge of TXC.
TX_CTL	MAC	In RGMII mode, TX_EN on positive edge of TXC, and a logical derivative of TX_EN and TX_ER on negative edge of TXC.
RXC	PHY	The transmit reference clock will be 125 MHz, 25 MHz, 2.5 MHz and shall be derived from received data stream.
RD<3:0>	PHY	In RGMII mode, bits 3:0 on positive edge of RXC, bits 7:4 on negative edge of RXC.

2.9.5 Multiplexing of Data and Control:

Multiplexing of data and control information is done by taking advantage of both edges of the reference clocks and sending the lower 4 bits on the positive edge and the upper 4 bits on the negative edge of clock. Control signals can be multiplexed into a single clock cycle using the same technique.

2.9.6 TXERR and RXERR Coding:

To reduce power of this interface, TXERR and RXERR, will be encoded in a manner that minimizes transitions during normal network operation. This is done by the following encoding method. Note that the value of TX_ER and TX_EN are valid at the rising edge of clock while TXERR is presented on the falling edge of the clock. RXERR coding behaves in the same way.

TXERR <= TX_EN (XOR) TX_ER

RXERR <= RX_DV (XOR) RX_ER

When receiving a valid frame with no errors, RX_DV=true is generated as a logic high on the rising edge of RXC and RXERR=false is generated as logic high on falling edge of RXC. When no frame is being received, RX_DV=false is generated as a logic low on the rising edge of RXC and RXERR=false is generated as a logic low on the falling edge of RXC.

While receiving a valid frame with errors, RX_DV=true is generated as logic high on the rising edge of RXC and RXERR=true is generated as a logic low on the falling edge of RXC.

TXERR is treated in a similar manner. During normal frame transmission, the signal stays at logic high for both edges of TXC and during the period between frames where no errors are to be indicated, the signal stays low for both edges.
Following table shows the allowable encoding of TXD, TXERR and TX_EN. Table 2.5 shows the allowable encoding of RXD, RXERR and RX_DV.

Table 2.5: Signal coding for TXD, TXERR and TX_EN.

TX_CTL	**TX_EN**	**TX_ER**	**TXD<7:0>**	**Description**
0,0	0	0	00 through FF	Normal inter-frame
0,1	0	1	00 through 0E	Reserved
0,1	0	1	0F	Carrier Extend
0,1	0	1	10 through 1E	Reserved
0,1	0	1	1F	Carrier Extend error
0,1	0	1	20 through FF	Reserved
1,1	1	0	00 through FF	Normal data transmission
1,0	1	1	00 through FF	Transmit error propagation
NOTE- Values in TXD<7:0> column are in hexadecimal.				

Table 2.6: Signal coding for RX_DV, RXERR and RX_ER.

RX_CTL	**RX_DV**	**RX_ER**		**RXD<7:0>**	**Description**	**PHY Status Parameters**
0,0	0	0	#	xxx1 or xxx0	Normal inter-frame	Indicates link status 0=down, 1=up
0,0	0	0	#	x00x or x01x or x10x or x11x	Normal inter-frame	Indicates RXC speed 00=2.5 MHz, 01=25 MHz, 10=125 MHz, 11=reserved
0,0	0	0	#	1xxx or 0xxx	Normal inter-frame	Indicates duplex status 0=half-duplex, 1=full-duplex
0,1	0	1	*	00	Normal inter-frame	
0,1	0	1	*	01 through 0D	Reserved	
0,1	0	1	*	0E	False Carrier indication	
0,1	0	1	*	0F	Carrier Extend	
0,1	0	1	*	10 through 1E	Reserved	
0,1	0	1	*	1F	Carrier Extend Error	
0,1	0	1	*	20 through FE	Reserved	
0,1	0	1	*	FF	Carrier Sense	
1,1	1	0	*	00 through FF	Normal data reception	
1,0	1	1	*	00 through FF	Data reception error	

* NOTE- (Required Function) Values in RXD<7:0> column are in hexadecimal.

NOTE- (Optional) Values in RXD<7:0> column are in binary.

Chapter 3
System Review (Basic Theory)

3.1 eVC (e Verification Component):

e Verification Components (eVCs) are reusable, configurable, pre-verified, plug-and-play Verification environments. They offer the easiest to use, most complete module, and chip and system level verification solution available. eVCs integrate automatic stimulus generation, assertion checking, and functional coverage analysis all within in a single, extensible component. eVCs drastically reduce the time needed to compose a verification environment. The philosophy underlying eVCs differs significantly from alternative products. Rather than use thousands of directed tests, the eVC employs automatic generation and a coverage driven methodology. Using automated scenario generation the eVC can typically achieve 90%+ coverage of the protocol. With the addition of a few tests the remaining corner cases are then exercised. This approach uncovers more bugs faster and frees engineering time to focus on testing the DUT's proprietary functionality.

3.2 Ethernet eVC:

The Ethernet eVC can be used to verify IEEE 802.3 compliance MAC and PHY devices. The eVC can be used for the functional verification of IP cores and SoC designs incorporating Ethernet MAC and PHY functionality. Figure 3.1 shows the architecture of Ethernet eVC.

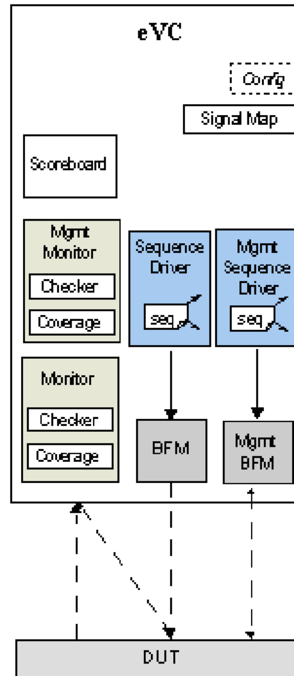


Figure 3.1: Architecture of Ethernet eVC

The Ethernet eVC environment is represented by vr_enet_env.e. The eVC can simulate either MAC or PHY behavior for the Media independent interfaces.

3.3 Features of the Ethernet *e*VC

Like any verification environment built with Specman Elite, the Ethernet ***eVC*** can:

- Generate traffic stimuli to the DUT
- Check that the DUT adheres to the protocol
- Collect coverage related to the DUT

3.4 Ethernet Traffic Emulation

The Ethernet traffic consists of:

- Ethernet packets coming from the MAC or PHY port to the DUT

The ***eVC*** sequences handle the generation of traffic and BFMs handle the emulation of the traffic. The ***eVC*** can:

- Generate Ethernet packets and random data packets and drive them according to the protocol.

- Generate and collect management interface traffic.

3.4.1 Elements of Ethernet eVC:

Overall wrapper of eVC is vr_enet_env.e. All active and passive agents are instantiated under env. Active agents and their types can be configured according to the design under test. As shown in the architecture DUT is having two ports one is for transmission and other is for reception and can have some control signal as per different interfaces. Width of TX and RX path and control signals will change according to interface being used.

3.4.1.1 Config: A group of fields that allow configuration of the agent's attributes and behavior.

3.4.1.2 Agnets: For each port of the interface, the **e**VC implements an agent. These agents can emulate the behavior of a legal device, and they have standard construction and functionality. Each env also has a group of fields, marked in Figure as *Config*. This allows configuration of the env's attributes and behavior. Agents are of two types Active and Passive agent.

- **Active agent:**

  The active agents drive traffic to the DUT with the Ethernet sequence driver. The Ethernet sequence driver generates various sequences and these sequences produce Ethernet packets or random lists of data. The Ethernet packets or random lists of data are injected into the DUT by the BFM. The BFM injects them on the Tx lines for the MAC agent and Rx lines for the PHY agent. The active MAC or PHY agents generate Ethernet packets depending on the constraints provided by the user on various item fields. Active agents also contain a monitor to do the checking and collecting coverage. Active agent can generate traffic to the DUT and can also respond to traffic from DUT.

- **Passive agent:** The passive agent consist of:

  - A monitor, represented by vr_enet_monitor.
  - A scoreboard, represented by vr_enet_scoreboard.

The passive agent has both the Tx and Rx collectors in the monitor, by default. The Tx monitor senses signals on the Tx path and the Rx collector senses signals on the Rx path. The

monitor collects the packets and emits events on the status of traffic to and from the DUT. The monitor contains predefined coverage definitions and you can create additional coverage definitions and protocol checks to meet the test bench requirements. The monitor also contains predefined checks that verify the DUT's adherence to the Ethernet protocol. The monitor in the passive MAC agent checks for the protocol violation on the Tx lines and monitor in the passive PHY agent checks for protocol violation on the Rx line. You can configure the passive agents for non-layered interfaces only. The scoreboard unit verifies the data integrity of Ethernet packets by comparing the sent and received Ethernet packets.

3.5 Flow of Data within the Agents

The BFM initiates a new packet for transmission by calling the sequence (seq.) driver if transmission of previous packet is over and other required conditions match. If an Ethernet packet needs to be transmitted, then the sequence driver generates the required packet and passes it to the Ethernet BFM. If a management packet needs to be transmitted, then the management sequence driver generates the required management packet and passes it to the management BFM. During reception cycle both the active and passive vr_enet_agent(s) collect list of bits, list of di-bits, list of nibbles or list of bytes depending on the type of interface. The agents then re-group the packet in the collector and check for packet related errors.

3.6 Agent Architecture

Figure 3.2 displays the agent architecture at an overview level for non-layered interfaces and Figure 3.3 shows the agent Architecture for layered interfaces, which displays each of the units present in the agents of the Ethernet ***eVC*** in detail.

Within each Ethernet ***eVC*** agent, the following units are instantiated:

- A config block that has the signals for configuration of the agent.
- A signal map block, the ***eVC*** signals mapped to the DUT.
- A monitor to check the DUT behavior and collect coverage information.
- A scoreboard, to check the data items, can be instantiated.

Additionally, the active agents consist of:

- An Ethernet sequence driver, represented by vr_enet_driver, and a BFM, represented by vr_enet_bfm.

- A management sequence driver, represented by vr_enet_mgmt_driver, and a BFM, represented by vr_enet_mgmt_bfm. The management sequence driver and BFM are present only in the active MAC agent.

The passive agents do not drive any signals. They use the monitor to check the DUT behavior and collect coverage information.

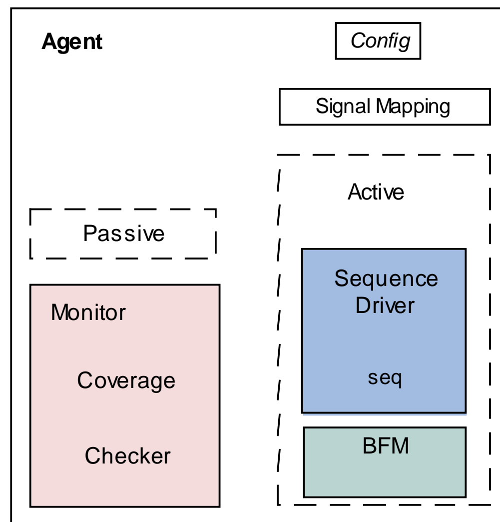


Figure 3.2: Agent Architecture

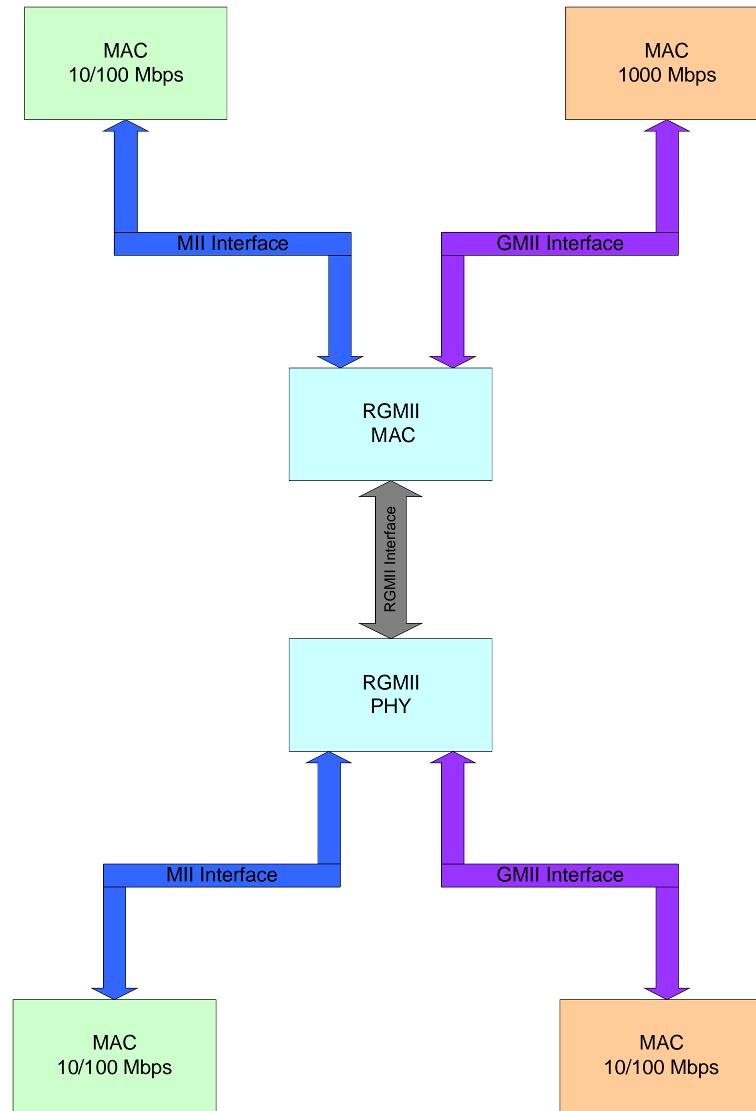


Figure 3.3: Agent Architecture for Layered Interface

3.7 Monitors and BFM Architecture

The monitors of the Ethernet ***eVC*** are completely passive. The BFM drives and generates the packets. The BFM can make use of the monitor or duplicate some of the monitor's logic. Most passive activity is done by the monitor, while all active interactions with the DUT are done by the BFM. For example, the monitor collects the packets and then emits an event for each packet received. The monitor consists of two collectors as shown in Figure 3.4:

- Tx Collector: The Tx collector packets from the Tx data path. By default, the collector is disabled for active MAC agents and enabled for passive agents.

- Rx Collector: The Rx collector packets from the Rx data path. By default, the collector is disabled for active PHY agents and enabled for passive agents.

The monitor has predefined checks to verify protocol adherence of the DUT and predefined coverage definitions to collect coverage. By default, the checks and coverage are enabled in the passive agents for non-layered interfaces and enabled in the active agents for layered interfaces. The monitor also has the hooks-has_tx_collector and has_rx_collector to enable the Tx and Rx collectors.

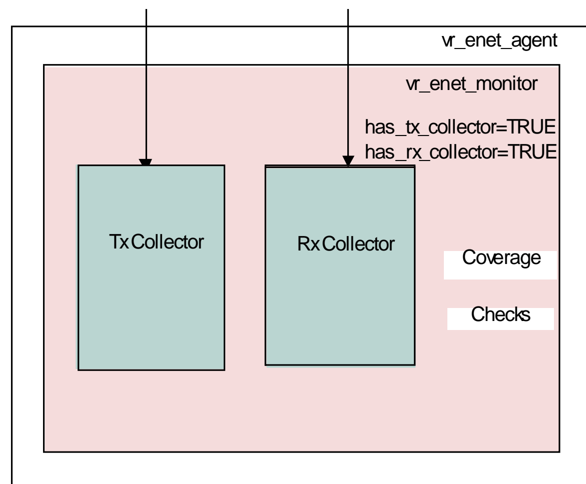


Figure 3.4: Monitor and BFM architecture

The monitor within the passive MAC agent checks for the protocol violation on the TX line and the monitor within the passive PHY agent checks for the protocol violation on the Rx line.

Similarly, the management monitor is used to collect management packets from the management interface line and the management BFM is used to drive management packets to the management unit of PHY.

3.8 Scoreboard Architecture

The scoreboard unit, represented by vr_enet_scoreboard, is used to check that the number and order of data items collected from each DUT output agent are as expected. In the Ethernet ***eVC,*** the scoreboard functionality is optional, and you can choose not to instantiate the scoreboard. The scoreboard can be instantiated either at the environment or the agent level.

By default, the scoreboard is enabled at the agent level. The scoreboard can be used for different configurations as listed below:

- A switch DUT: The scoreboard should be instantiated at the environment level.
- A repeater DUT: The scoreboard should be instantiated at the agent level.
- A single port DUT: The scoreboard can be instantiated at the agent or at the environment level.
- User-specific: The scoreboard can be instantiated at the agent or at the environment level for any other user-specific configurations. In these cases, you have to write your own hooks for adding and matching the packets in the scoreboard.

3.9 Scoreboard Checking:

One basic concern when checking data is to verify that the output data items collected from the DUT match the corresponding data items injected into the DUT. This kind of checking is called scoreboard checking. With scoreboard checking you verify that:

- Every input has a matching output
- Every output has a matching input

In the Ethernet ***eVC***, the scoreboard can be instantiated at the environment or agent level. By default, it is enabled at the agent level. It can disable by setting the has_scoreboard field to FALSE. The scoreboard collects the packets going in and coming out of the DUT and compares them. It can also verify that the packet has been sent to the intended port. If any mismatch is found, a scoreboard error is issued. The packet comparison is done by means of a Unique ID (UID). In case when the UID get corrupted, the packet comparison is done on the basis of 32-bit Cyclic Redundancy Check (CRC) calculation on the whole packet.

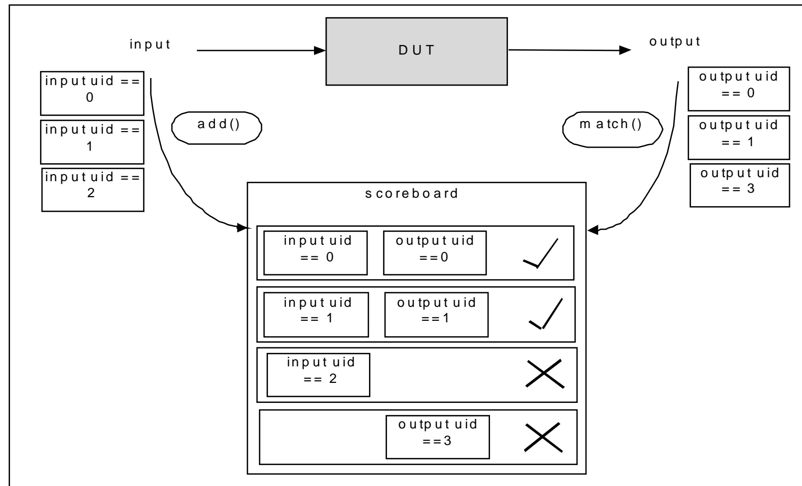


Figure 3.5: Functioning of Scoreboard

3.10 Topologies for Verification at the Module Level

The Ethernet ***eVC*** can simulate MAC and PHY behavior for verifying either of the device type. Following are configurations of eVC for verifying MAC and PHY devices. Figure 3.6 shown below gives overview of Ethernet eVC in user's verification environment.

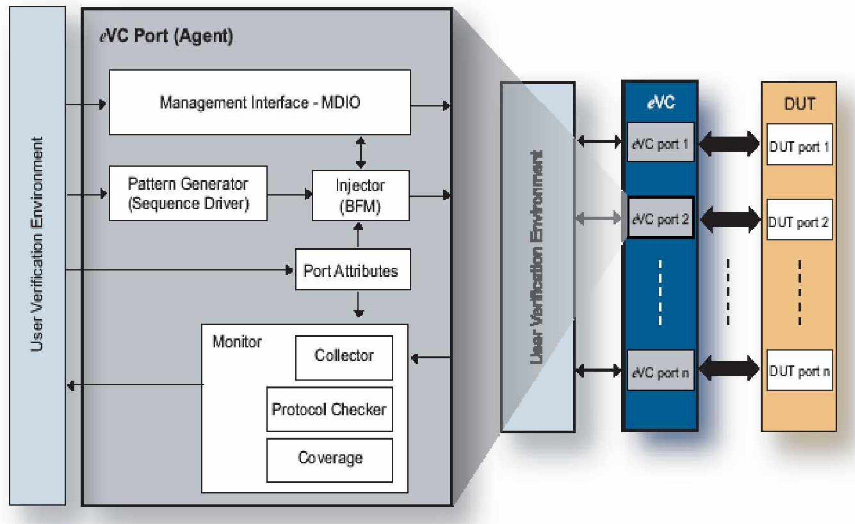


Figure3.6: Ethernet eVC in user's verification environment.

3.11 Single Port MAC DUT for non-layered Interfaces

To verify a Single Port MAC DUT, you must have an active PHY agent to drive traffic to the DUT and a passive MAC agent to monitor the DUT, as shown in Figure 3.7.

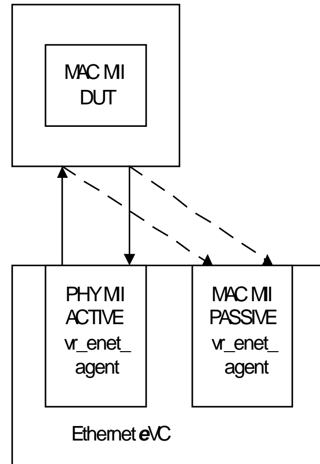


Figure 3.7: Single Port MAC DUT

3.12 Multi-Port MAC DUT

To verify a multi-port MAC DUT, you must have equal number of active PHY agents (as the DUT) to drive traffic to the DUT and equal number of passive MAC agents (as the DUT) to monitor the DUT, as shown in Figure 3.8.Figure 3-7. Each port of the DUT might have same or different interfaces.

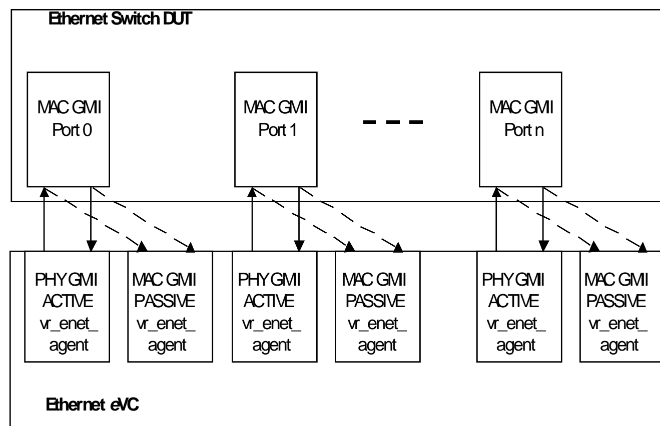


Figure 3.8: Multi-Port MAC DUT

3.13 Single-Port PHY DUT for non-layered Interfaces

To verify a single-port PHY DUT, you must have an active MAC agent to drive traffic to the DUT and a passive PHY agent to monitor the DUT, as shown in Figure 3.9. The active MAC agent has the capability to generate Ethernet packets as well as random list of nibbles in case of MII; random list of di-bits in case of RMII, and random list of bytes in case of GMII and XGMII.

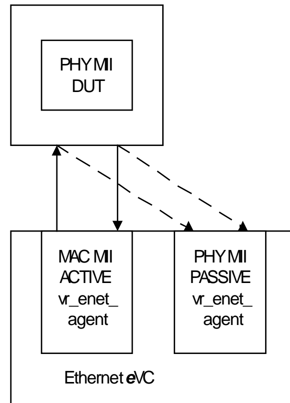


Figure 3.9: Single-Port PHY DUT

3.14 Multi-Port PHY DUT

To verify a multi-port PHY DUT, you must have equal number of active MAC agents to drive traffic to the DUT and equal number of passive PHY agents to monitor the DUT, as shown in Figure 3.10. Each port of the DUT can have the same or different interfaces.

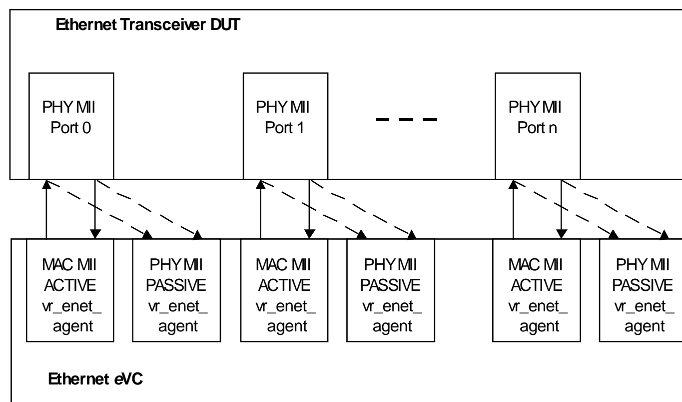


Figure 3.10:Multi-Port PHY DUT

3.15 Verification Environment Architecture:

Ethernet eVC can be used for verifying IP cores of MAC and PHY supporting IEEE 802.3 Std. It is requires in first place to verify this eVC. It should be carefully checked that whether it fulfills all the requirements of Ethernet Protocol, is eVC functionally adhering to protocol or not? So for t is required to build the mock verification environment (VE), and configuring the eVC for different kinds of possible DUTs. The Ethernet verification environment can be set up under system level environment. In the first half of the project, we put more wattage on verification of Ethernet eVC itself. Our goal was to make eVC fault (bug) free and to achieve 100% coverage of Ethernet eVC with reference to IEEE protocol for Ethernet. For this purpose, we created a verification environment (VE) for the eVC. Below figure shows VE architecture at functional level.

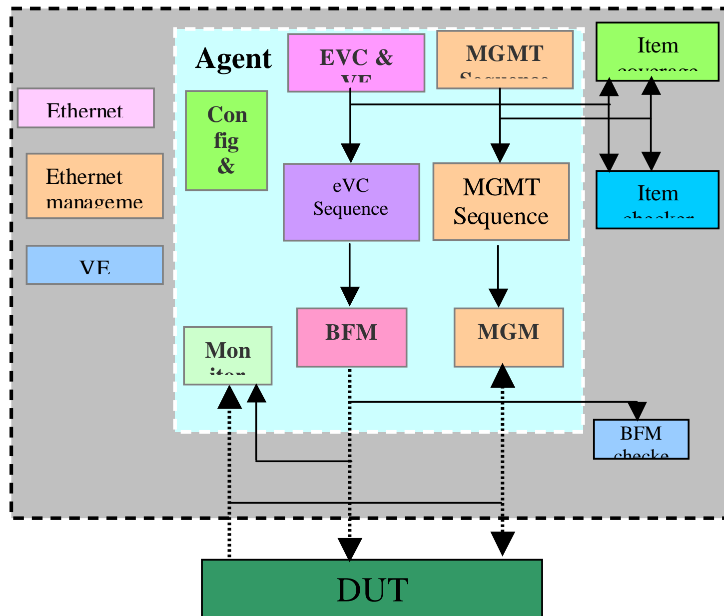


Figure 3.11: Ethernet *e*VC VE functional block diagram

3.15.1 Item Coverage: This module covers the item generated by the sequence. Here VE coverage definition is different from eVC coverage. The eVC coverage defines all the rules and items Ethernet protocol supports. The list of coverage items for eVC is as per appendix C. For VE coverage item list, it should cover all the checkers that are introduced in the eVC, scoreboard items and different possibilities of eVC configuration in addition to items listed in eVC coverage list.

While defining coverage items, some questions related to specifications must be answered. Below are some sample questions.

- Have all packet/transaction types been tried?
- Have all CPU opcodes and operand combinations been tested?
- Have all legal state transitions occurred?
- Have all instruction types been interrupted?
- Have all cases of resource contention been tested?
- Have all queue limits been stressed?

3.15.2 BFM checker: This unit checks whether the item fields (virtual fields) constrained by the sequence driver; is driven as per constrains through the BFM or not. Verification target: Verify that the virtual fields of the item are driven correctly from the bfm to the monitor.

Feature Name	**Verification Target**
Error injection	Injected error corresponds to the configuration.
	Timing for error assertion
	Error duration
	Data length
BFM duplex mode	HALF/FULL duplex
Bad CRC error injection	If TRUE CRC is valid
Short frame error injection	
Long frame error injection	
Sfd error injection	
Alignment error injection	
Pause Opcode error injection	
Length error injection	
Ipg	Duration of ipg.
Packet kind	Packet kind
Preamble	Preamble length

3.15.3 VE Sequences: The VE sequences are built on the basic sequences provided by *e*VC to generate complex scenarios. The VE has its own sequence library to simulate various complex scenarios so that checking of the checks can be done through them. The appendix – B list various sequence scenarios that are simulated by the VE sequences with brief description.

3.15.4 Agent Configuration Coverage: This module covers the various fields of the agent config struct.

3.15.5 Ethernet Error Logger: This module is used to expect errors based on packet injected by Ethernet BFM of agent. When DUT completes its packet collection, occurred errors are copied to this module. It compares both expected and occurred errors and gives error on mismatch.

3.15.6 Ethernet Management Error Logger: This module is used to expect errors based on management packet injected by MGMT BFM of agent. Rest of the functionality is same as Ethernet Error Logger module.

3.15.7 VE Monitor: This unit checks the validity of the checkers by comparing expected and occurred errors of error logger unit. Whether a check is fired only when there is an error condition violating the protocol or not; is checked by the VE monitor.

3.16 Is Data Collected Correctly?

Verification target: verify that the monitor collects items correctly. It use s scoreboard to verify data integrity.

3.16.1 Checker Is Correct?

Verification target: verify that the checks fired are correct.

Generate the sequence to generate both erroneous/non-erroneous behavior and check against expected behavior with a separate checker.

Whenever a sequence is intended to fire checks from eVC monitor, expected errors list is updated with the list of expected error tag names. The errors occurred during the execution of sequence are logged into occurred errors list. At the end of sequence both the lists are compared and result is indicated. If both the lists match then a message saying: **"Expected and occurred errors are matched"** is displayed else a message saying: **"Errors expected but not occurred:"** or **"Errors occurred but not expected"** is displayed.

For checker, coverage buckets are defined for each of them with their corresponding tag name. Whenever a dut_error occurs, the bucket for that particular check is filled.

3.16.2 Coverage is correct

Verify that following coverage definitions are exercised at least once and coverage is being collected correctly.

3.16.3 Configuration check / coverage

This module checks and covers for the configuration of the *e*VC. It also checks that the *e*VC is configured as per the user configuration.
It checks for the following.

- § Topologies
- § User configuration.

3.16.4 Scoreboard

The scoreboard is used to check the data integrity between the BFM of one agent on one end and the monitor of other agent on the other end. Following is the list of scoreboard instances and corresponding input items, which are to be compared.

Scoreboard1: Item 1 – Ethernet BFM item (MAC ACTIVE Agent).
Item 2 – Monitor item. (PHY ACTIVE Agent)

Scoreboard2: Item 1 – Ethernet BFM item (PHY ACTIVE Agent).
Item 2 – Monitor item. (MAC ACTIVE Agent)

Chapter 4

System Design

The **eVC** supplies a default sequence library with a predefined set of sequences that execute typical scenarios. Sequences are made up of three main entities:

- **Item**: A struct that represents the basic data item to the DUT (for example, a packet).
- **Sequence**: A struct that represents a stream of items signifying a high-level scenario of stimuli. This is done by generating items one after the other, according to some specific rules. The sequence struct has a set of predefined fields and methods. The sequence struct can also be extended for adding more functionality.
- **Sequence Driver:** Each sequence driver has a MAIN sequence, within which all other sequences are generated. It is a unit that serves as the mediator between the sequences and the verification environment. The sequence driver acts on the items generated through sequences, typically passing them to the BFM (Bus Functional Model).

For the purpose of driving the data into the DUT, the sequence driver interacts only with the BFM. The sequence driver and the BFM work as a pair, where the sequence driver serves as the interface upwards towards the sequences so that the sequences can always see a standard interface to the DUT.

The BFM serves as the interface to the DUT, pulling items from the sequence driver and passing them to a device.

4.1 Structure of Sequences

In the Ethernet ***eVC***, the individual Layers—PHY and MAC are responsible for generating the packets. Both layers use a common sequence driver and have their own sets of pre-defined sequences to generate all the sequence items as shown in:

- 4.2 Ethernet Sequence Structure
- 4.2 Management Sequence Structure

4.2 Ethernet Sequence Structure

The Ethernet ***eVC*** has a sequence driver and Figure 4.1: Ethernet Sequence Structure displays internal structure of Ethernet sequences. In the sequence structure:

- sequence driver is *vr_enet_seq_driver*
- sequence struct is *vr_enet_seq*
- sequence item is *vr_enet_packet*

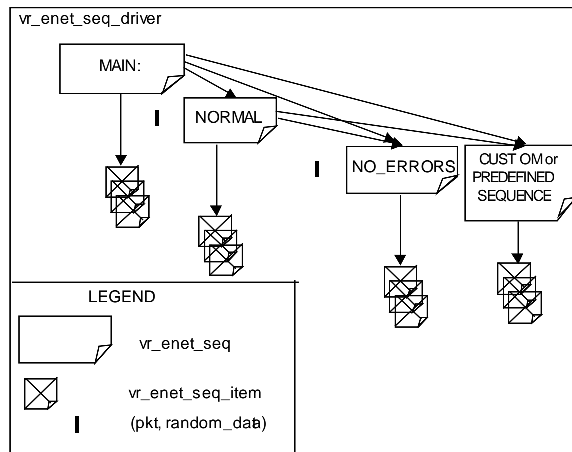


Figure 4.1: Ethernet Sequence Structure

The Ethernet sequence driver generates packets using either MAIN, predefined, or custom sequences. The MAIN sequence is responsible for generating all types of sequences and the sequences generate basic data items. The items generated by sequences are passed to the BFM, which sends them to the DUT. The MAIN sequence is started automatically upon *run*(). It is used as the root for the whole sequence tree.

4.2 Management Sequence Structure

The Ethernet ***eVC*** has a management sequence driver. Sequence Structure displays internal structure of management sequences as shown in Figure 4-2. In the sequence structure:

- Sequence driver is vr_enet_mgmt_seq_driver
- Sequence struct is vr_enet_mgmt_seq
- Sequence item is vr_enet_mgmt_packet

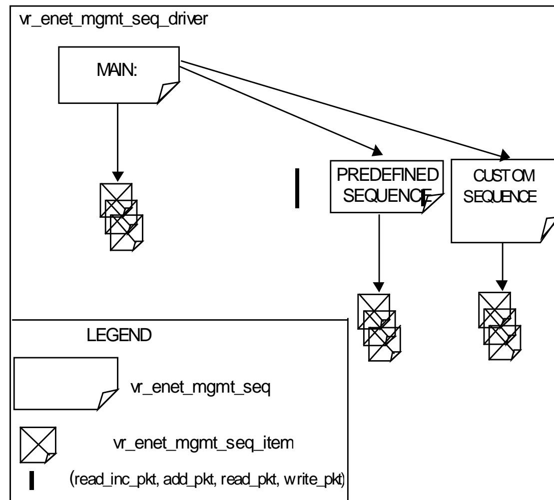


Figure 4.2: Management Sequence Structure

4.3 Injecting Ethernet Packets with Packet Errors

Packet errors are common to all interfaces. This section describes methods for injecting packet errors on different packets. The various user interfaces to inject packet errors in the Ethernet packet are:

1. Generating Ethernet packet with any one-packet error selected randomly.

```
extend MAIN vr_enet_seq {
    !pkt_err : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- The packet error kind is random.
        do SINGLE_PACKET_ERROR pkt_err;
    };
};
```

2. Generating Ethernet packets with specific single packet error.

```
extend MAIN vr_enet_seq {
    !eth_pkt : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- The packet error kind is CRC error.
        do crc_err eth_pkt;
```

```
-- The packet error kind is SFD error.
      do sfd_err eth_pkt;
-- The packet error kind is short frame error.
      do short_frame_err eth_pkt;
-- The packet error kind is long frame error.
      do long_frame_err eth_pkt keeping  {
         .packet_kind != ETHERNET_JUMBO
         };
-- The packet error kind is pause OPCODE error.
      do  pause_opcode_err eth_pkt keeping {
         .packet_kind == ETHERNET_PAUSE
      };
-- The packet error kind is length error.
      do len_err eth_pkt keeping {
         .packet_kind in  [ETHERNET_802_3,ETHERNET_MAGIC,
         ETHERNET_SNAP]
         };
   };
};
```

3. Generating Ethernet packet with specific single packet error and specific value.

```
extend MAIN vr_enet_seq {
   !err_pkt : ETHERNET INJECT vr_enet_packet;
   body() @driver.clock is only {
-- The packet error kind is CRC error with a specific value.
      do crc_err err_pkt keeping {.crc == 32'hffffffff};
   };
};
```

4. Generating Ethernet packet with random multiple packet errors.

```
extend MAIN vr_enet_seq {
   !error_pkt : ETHERNET INJECT vr_enet_packet;
   body() @driver.clock is only {
-- More than two random packet errors are generated in a packet.
      do MULTI_PKT_ERR error_pkt;
   };
```

```
};
```

5. Generating Ethernet packet with specific multiple packet errors.

```
extend MAIN vr_enet_seq {
    !pkt_error : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- More than two packet errors; one being CRC error.
        do crc_err MULTI_PKT_ERR error_pkt;
-- Two specific errors.
        do  crc_err sfd_err pkt_error;
    };
};
```

4.4 Injecting Ethernet Packets with Protocol Errors

Protocol errors are specific to interfaces. These errors can be injected in the various phases of the Ethernet packet such as IPG phase, preamble phase, header phase, data phase, CRC phase, and extension phases. There are some protocol errors that can be injected in a particular phase only and there are some that can be injected in more than one phase. The packet phase, the start time, the duration, and occurrences of a protocol error can be controlled.

The various user interfaces to inject protocol errors are:

1. Generating Ethernet packets with protocol errors of single kind.

```
extend MAIN vr_enet_seq {
    !prot_err_pkt : ETHERNET INJECT vr_enet_packet;
    body()@driver.clock is only {
-- The err_kind,err_occurrence, err_phase, error timings, and error
-- lengths are random.
        do SINGLE_PROT_ERR_KIND prot_err_pkt;
    };
};
```

2. Generating Ethernet packets with protocol errors of multiple kind.

```
extend MAIN vr_enet_seq {
```

```
    !eth_prot_pkt : ETHERNET INJECT vr_enet_packet;
    body()@driver.clock is only {
-- The err_kind, err_occurrence, err_phase, error timings, and error
-- lengths are random.
        do MULTI_PROT_ERR_KIND eth_prot_pkt;
    };
};
```

3. Generating Ethernet packets with protocol error of a specific error kind and single error occurrence.

```
extend MAIN vr_enet_seq {
    !pkt : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- Generates Ethernet packet with single TX_ER error
-- The error-phase, error timing, and error length are random.
-- The error occurrence is single time.
        do SINGLE_PROT_ERR_KIND pkt keeping {
            for each (e) in .protocol_errs {
                e.err_kind == TX_ER and e.err_occurrence ==
                SINGLE_TIME;
            };
        };
    };
};
```

4. Generating Ethernet packets with protocol error of a specific error kind and multiple error occurrence.

```
extend MAIN vr_enet_seq {
    !pkt : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- Generates Ethernet packets with multiple TX_ER errors.
-- The error-phase, error timings, and error lengths are random.
-- The error occurrence is multiple times.
        do SINGLE_PROT_ERR_KIND pkt keeping {
            for each (e) in .protocol_errs {
                e.err_kind == TX_ER and e.err_occurrence == MULTI_TIME;
```

```
            };
        };
    };
};
```

5. Generating Ethernet packets with protocol errors of two specific error kinds.

```
extend MAIN vr_enet_seq {
    !pkt : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- Generates Ethernet packets with TX_ER and
     TX_CARRIER_EXTENSION_ERROR
-- errors.
-- The error-occurrence, error-phase, error timings, and error lengths
-- are random.
        do MULTI_PROT_ERR_KIND pkt keeping {
            .protocol_errs.size()==2 and
            for each (e) in .protocol_errs {
                index == 0 => e.err_kind == TX_ER;
                index == 1 => e.err_kind == TX_CARRIER_EXTENSION_ERROR;
            };
        };
    };
};
```

6. Generating Ethernet packets with protocol errors of specific error kind, in a specific phase, and single error occurrence.

```
extend MAIN vr_enet_seq {
    !pkt : ETHERNET INJECT vr_enet_packet;
    body() @driver.clock is only {
-- Generates Ethernet packet with single TX_ER error in DATA phase.
-- The error timing and error length are random.
-- The error phase is data and error occurrence is single time.
        do SINGLE_PROT_ERR_KIND pkt keeping {
            for each (e) in .protocol_errs {
                e.err_kind == TX_ER and e.err_phase == DATA and
                e.err_occurrence == SINGLE_TIME;
```

```
            };
        };
    };
};
```

4.5 Monitoring, Coverage and Checks:

The vr_enet_monitor and vr_enet_mgmt_monitor units are responsible for monitoring the ***e***VC and DUT. The monitor relies on its agent for initial setup. Thereafter, it is independent of the agent, only looking at signals. The monitor recognizes packets going over the line, analyzes them, and then emits corresponding events. The checker, coverage mechanism, and scoreboards are extensions of the monitor. They add checks and coverage groups, based mainly on the monitor events.

4.5.1 Using the Monitor

The Ethernet ***eVC*** has a monitor for both the Ethernet and management interfaces. By default, both the passive and active agents have monitor instantiation. The coverage and checkers by default are enabling in active agents for non-layred interfaces like MII, GMII, RMII, XGMII, and SMII. This is because the passive agents are not applicable for the layered interfaces.

4.5.2 Collecting Coverage

Coverage can be implemented either as a separate unit in the agent or in a has_coverage subtype of the monitor. By default, the has_coverage flag is TRUE in passive agent and FALSE in active agent. If you want to verify the ***e***VC active agent's capabilities, the has_coverage flag can be set to TRUE.

Active agent's capabilities can be verified for various scenarios at each layer by analyzing the coverage of generated sequence items for the following:

- injected packet errors
- packets with various data lengths
- packets with different packet formats
- injected protocol errors

The Ethernet ***eVC*** has predefined coverage definitions for each interface. These definitions include:

- Coverage of various packet fields
- Cross coverage of the required coverage items.

4.5.3 Checking the Protocol

The checkers are responsible for checking the DUT behavior. e***VC*** contains data related check like long frame error, short frame error, start frame delimiter error, CRC error etc which are common to all interface. Those predefined checks can be disabled if required. To disable all the checks for all the interfaces, constrain the *has_checks* field of the *vr_enet_agent*. For example:

```
extend vr_enet_agent {
   keep has_checks == FALSE;
};
```

Disabling checks for a specific check for a specific interface, say the MII, as shown in the example below:

```
extend sys {
   setup() is also {
      set_check("ERR_ENET017_MAC_MII_IPG_TOO_SHORT",
ignore);
   };
};
```

Chapter 5

Coverage Driven Verification

Functional verification already consumes most of the IC logical design flow, as some studies suggest, what's going to happen as chip complexity reaches 10 million or 100 million gates?

The answer is sheer chaos-unless the functional-verification process can be made more manageable. Coverage-driven verification can help today, but the long-range answer lies in rethinking both verification and design. Some experts say as chip complexity grows, there's an exponential increase in the number of things that could potentially go wrong, and hence need verifying. Can that be done without hiring armies of verification engineers to churn out directed tests?

Formal verification can provide targeted, exhaustive tests, but it doesn't cover everything. Acceleration and emulation speed the process, but you've still got to generate and monitor the tests. Faced with a multitude of design styles, tools and verification techniques, one point is clear: Designers have got to have a plan.

A verification plan starts by identifying what portions of the design are going to be tested, and how. It identifies input scenarios to apply to the design under test, and calls out tough corner cases that might not be found by simulation. It also does, or should, set forth a plan to measure progress by applying coverage-driven verification.

Engineers today are most familiar with code coverage, which checks to see if there are unexecuted areas of code. Most people would agree it is unacceptable to synthesize that is either dead or unverified. Nevertheless, code coverage is not enough. Most functional scenarios cannot be mapped in to lines of code. For example code coverage cannot indicate whether we have been thorough all the legal combination of states in two orthogonal state machines.

Another example might be whether we have tried all the possible inputs while the design under test (DUT) was in all the different internal states. Also, code coverage does not look at sequences of events, such as what else happened before, during, or after a line of code has

been executed. Thus, code coverage does not ensure completeness and does not fulfill most of the requirements that allow expediting the verification task.

An emerging and more difficult technique is functional coverage, which can be used to check various corner cases-making sure, for instance, that a FIFO actually empties and fills.

With functional coverage to provide feedback, random-test generation becomes more practical and tougher bugs can be found. Finally, the growth of assertion-based verification has given rise to assertion coverage, which, among other things, checks to see if assertions actually fired or not. Figure 5.1, below, provides an example of functional coverage in an environment that creates many simulation scenarios.

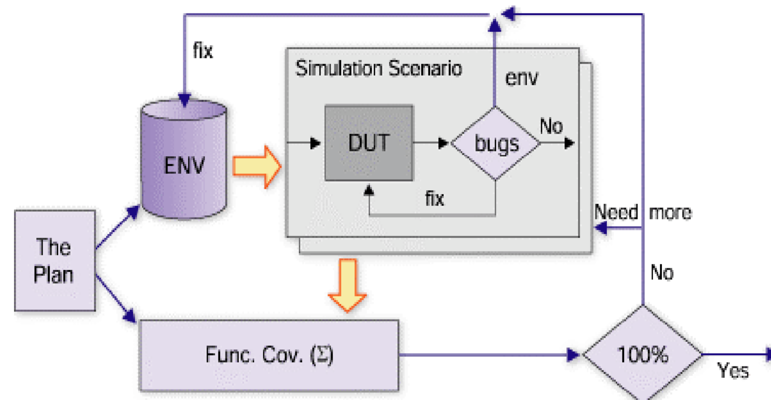


Figure5.1: Functional coverage serves multiple simulation scenarios

5.1 Higher abstraction:

But that's just a start. To really manage the verification process, some observers say, it will be necessary to move to higher levels of abstraction, and to start with a system-level, rather than a block-level, view. Others say the design process itself will have to be improved so there are fewer bugs in the first place.

Functional coverage provides an excellent indication of how we're meeting the goals set by the test plan. However, it may not correlate exactly to the actual RTL implementation, which may have diverged over time. For example, code coverage results can find a "hole" in the test plan -- functionality that is implemented in the RTL, but never targeted by the test plan. Therefore, code coverage and functional coverage are complementary. Table 5.1 illustrates

how functional coverage and code coverage correlate to each other, and how the combination of both provides a much more reliable indication of complete coverage.

Table 5.1: Correlation between functional coverage and code coverage.

Functional Coverage	Code Coverage	Indication
Low	Low	Early in verification
Low	High	Missing sequences and corner cases.
High	Low	Need to improve test plan
High	High	High confidence of quality

5.2 Coverage requirements:

As all types of coverage are complementary in nature, a tool or methodology that combines approaches is extremely beneficial. As mentioned earlier, this combined methodology will provide a complete overview of the verification progress and a clearer correlation between the functional coverage definitions and the actual design implementation.

The requirements for coverage can be categorized into two groups: demands for the data gathering and analysis engine, and requirements for the surrounding test bench that will allow efficient usage of the accumulated information. Following are the main requirements for coverage driven verification:

- Informative reports:
  Getting coverage results should be readable and intuitive. Both a textual user interface and a graphic user interface (GUI) should be provided. Usually, engineers use the GUI since it provides an easy means to review, query and print the coverage database. The textual interface is useful when trying to forward the results to other automatic tools or manipulate the data into custom reports. The coverage engine in Specman Elite collects functional coverage information based on user inputs that is easily driven from the test plan. Below figure 5.2 shows the functional coverage of Ethernet Packet in graphical user interface (GUI).

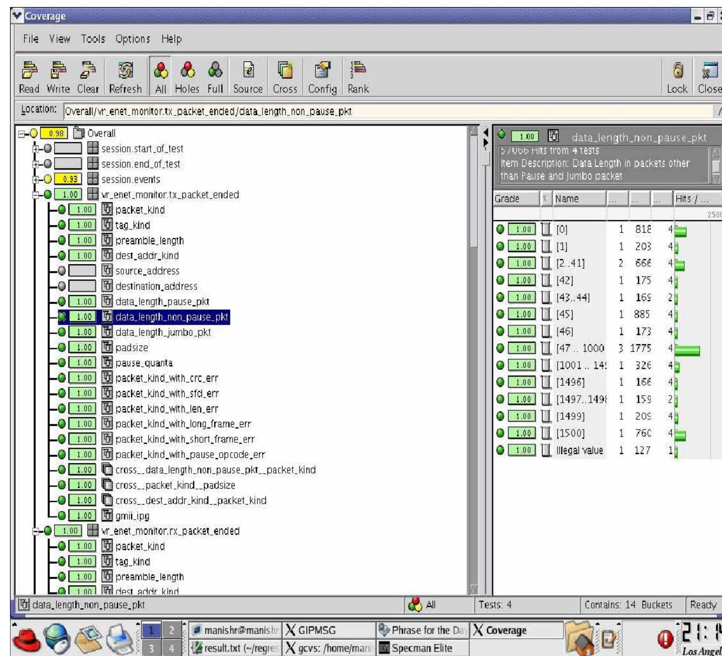


Figure5.2: Functional coverage of Ethernet Packet.

- Efficient coverage analysis:
  When coverage results are less than satisfying, it should be easy to deduce the appropriate adjustments and generate tests to improve the coverage results.
- Test base ranking:
  The ability to accumulate and analyze coverage reports from multiple simulation runs is crucial. Test suites today comprise of large number of tests. This ability to analyze cumulative coverage allows you to:
  - Get an overall picture of the entire verification environment, and measure your recent progress. Using these measurements, you can objectively predict the tape-out date.
  - Avoid test redundancy. By measuring the amount of coverage added by each test, redundant tests can be identified and removed.
- Timing of analysis:
  The coverage tool should allow the engineer to analyze the coverage information both between simulations and during a test run. The first approach requires the ability to save the collected information to be reviewed later on. The second approach requires a run-time interface to the coverage database that allows it to be used during simulation.

- Grading:
  Various coverage holes may be prioritized and the overall progress can be better represented by setting individual goals for each concern (how many times must I cover each scenario?), and by setting weights to distinguish the relative importance of different coverage scenarios.

- Optimizing the test suite:
  Ranking capabilities should allow you to create a subset of tests that verify the DUT to a significant degree, with a minimum amount of resources. Running this subset of tests instead of your entire test suite drastically reduces the total number of cycles needed for verification.

- Open environment:
  Intellectual property reuse and design complexity have turned our verification environment into a mix of design representations and verification languages. Having the flexibility to use the same methodology on all types of designs is critical.

5.3 Steps for achieving coverage:

The following flow incorporates all coverage metrics. These guidelines provide a more complete metric and methodology that can be examined through the various phases, while steering the verification process towards a rapid completion.

- Phase one-Test plan:
  A good test plan should list all the interesting test cases to verify the design. In specific, it should include all configuration attributes, all variations of every data item, interesting sequences for every DUT input port, all corner cases to be tested, all error conditions to be created and all erroneous inputs to be injected.
  An encompassing test plan is a good start to ensure complete verification. Experience and creativity can be used to identify areas that are prone to bugs.
  Note that no test plan can cover every possible bug, which highlights the importance of directed-random test generation. However, a good test plan is still essential to an efficient verification strategy and becomes the basis for a functional coverage model. Following table shows the packet related test plan:

Table 5.2: Scenarios to be checked on various Ethernet packets.

NORMAL		
Fields	**Valid values**	**Invalid values**
Errors	Errors=0	Errors>0
Preamble length	Preamble_length=56	Preamble_length!=56

PAD_FRAME		
Fields	**Valid values**	**Invalid values**
Data_length & tag_kind	Data_length in [1..45] & tag_kind = UNTAGGED	Data_length>45 & tag_kind = UNTAGGED
Data_length & tag_kind	Data_length in [1..42] & tag_kind = VLAN_TAG	Data_length > 42 & tag_kind = VLAN_TAG
Data_length & tag_kind	Data_length in [1..38] & tag_kind = DOUBLE_VLAN_TAG	Data_length > 38 & tag_kind = DOUBLE_VLAN_TAG

CRC_TEST		
Fields	**Valid values**	**Invalid values**
Crc_err	Crc_err = TRUE	Crc_err = FALSE

LONG_FRAME_TEST		
Fields	**Valid values**	**Invalid values**
Data_length & tag_kind	Data_length > 1500 & tag_kind = UNTAGGED	Data_length <= 1500 & tag_kind = UNTAGGED
Data_length & tag_kind	Data_length > 1496 & tag_kind = VLAN_TAG	Data_length <= 1496 & tag_kind = VLAN_TAG
Data_length & tag_kind	Data_length > 1492 & tag_kind = DOUBLE_VLAN_TAG	Data_length <= 1492 & tag_kind = DOUBLE_VLAN_TAG

SFD_TEST		
Fields	**Valid values**	**Invalid values**
sfd_err	sfd_err = TRUE	Sfd_err = FALSE

SHORT_FRAME_TEST		
Fields	**Valid values**	**Invalid values**
Data_length	Data_length < 45 & pad = 0	Data_length > 45 & pad != 0
Short_frame_err	Short_frame_err = TRUE	Short_frame_err = FALSE

LENGTH_FRAME_TEST		
Fields	**Valid values**	**Invalid values**
Length_type	Length_type <= 1500 & length_type ! = data_length	Length_type > 1500 & length_type = data_length

*see appendix A for code of test_case.

- Phase two-Functional coverage specification: Define what should be covered. Decide on the interesting data fields/registers. Define separate buckets for legal values, illegal values, and boundary values, such as corner cases. Examine both interfaces and internal states. Choose the state registers and state transitions of important state machines. Identify interesting interactions between some of the above states or data, such as, the state of one state machine relative to another, or to a value of a signal. The functional coverage specification is, in essence, an executable form of test plan. Following figure shows functional coverage model for Ethernet eVC.

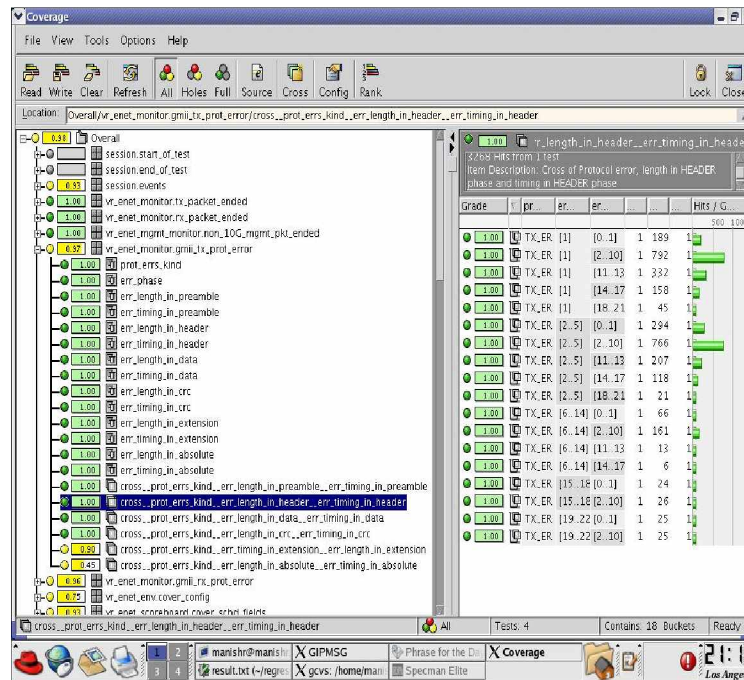


Figure5.3: Functional coverage model for Ethernet eVC.

- Phase three-Build the test bench:
  Build environment parameterized in a way that each test can direct it to a specific area of concern. This enables us to use the coverage results and translate coverage holes into new tests. At this point functional coverage and assertion coverage code should be written. It should be sure that the verification strategy we have chosen is suitable for the entire verification needs. Following table shows the sequences for various interface of Ethernet eVC.

TX_ER (only MII and GMII)		
Fields	**Valid Values**	**Invalid Values**
Errors	errors > 0	errors = 0
Length	length > 0	length = 0
Err_kind	err_kind = TX_ER	err_kind != TX_ER
Err_phase	err_phase !=CARRIER_EXTENSION	err_phase= CARRIER_EXTENSION

TX_CARRIER_EXTENSION_ERROR (only GMII and RGMII)		
Fields	**Valid values**	**Invalid values**
Errors	Errors > 0	Errors = 0
Length	Length > 0	Length = 0
Data_length	Data_length < 512	Data_length > 512
Err_kind	Err_kind = TX_CARRIER_ EXTENSION_ ERROR	Err_kind != TX_CARRIER_ EXTENSION_ ERROR
Err_phase	Err_phase = CARRIER_EXTENSION	Err_phase != CARRIER_EXTENSION

RX_ER(only MII,RMII and GMII)		
Fields	**Valid Values**	**Invalid Values**
Errors	errors > 0	errors = 0
Length	length > 0	length = 0
Err_kind	Err_kind = RX_ER	err_kind != RX_ER
Err_phase	Err_phase !=IPG	err_phase =IPG

CRS_DOWN_INJECT (only MII and GMII)		
Fields	**Valid Values**	**Invalid Values**
Errors	errors > 0	errors = 0
Length	length > 0	length = 0
Err_kind	err_kind = CRS_DOWN_INJECT	err_kind != CRS_DOWN_INJECT
Err_phase	err_phase!=IPG	err_phase=IPG

IPG_RX_ER (only MII, GMII and RMII)		
Fields	**Valid Values**	**Invalid Values**
Errors	errors = 0	Errors > 0
Length	length > 0	Length = 0
Err_kind	err_kind = IPG_RX_ER	err_kind != IPG_RX_ER
Err_phase	err_phase= IPG	err_phase !=IPG

FALSE_CARRIER_INDICATION (only MII, RMII, GMII and RGMII)		
Fields	**Valid values**	**Invalid values**
Errors	Errors > 0	Errors = 0
Length	Length > 0	Length = 0
Err_kind	Err_kind = FALSE_CARRIER_ INDICATION	Err_kind != FALSE_CARRIER_ INDICATION
Err_phase	Err_phase = IPG	Err_phase != IPG

RX_CARRIER_EXTENSION_ERROR (only GMII & RGMII)		
Fields	**Valid values**	**Invalid values**
Errors	Errors > 0	Errors = 0
Length	Length > 0	Length = 0
Data_length	Data_length < 512	Data_length > 512
Err_kind	Err_kind = RX_CARRIER_ EXTENSION_ ERROR	Err_kind != RX_CARRIER_ EXTENSION_ ERROR
Err_phase	Err_phase = CARRIER_EXTENSION	Err_phase != CARRIER_EXTENSION

Below table shows the management interface related scenarios for GMII, RGMII.

MGMT_RESET		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	none
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	Bit 15 of data = 1	Bit 15 of data = 0

SPEED_SELECT_10 (only MII, RMII)		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	Bit 15 = 0, bit 13=0 and bit 6 =0	Bit 15 = 1 or bit 13=1 or bit 6 =1

SPEED_SELECT_100 (only MII, RMII)		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	Bit 15 = 0, bit 13=1 and bit 6 =0	Bit 15 = 1 or bit 13=0 or bit 6 =1

SPEED_SELECT_1000 (GMII)		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	Bit 15 = 0, bit 13=0 and bit 6 =1	Bit 15 = 1 or bit 13=1 or bit 6 =0

LOOPBACK (only MII and GMII)		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	bit 15 = 0 and bit 14=1	bit 15 = 1 or bit 14=0

POWERDOWN		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	bit 15 = 0 and bit 11=1	bit 15 = 1 or bit 11=0

ISOLATION		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000
Data	bit 15 = 0 and bit 10=1	bit 15 = 1 or bit 10=0

READ_PHY_MGMT		
Fields	**Valid Values**	**Invalid Values**
addr1	addr1 = 5’b00000 to 5’b11111	None
addr2	addr2 = 5’b00000	addr2 !=5’b00000

*see appendix A for code of test case.

- Phase four- Writing tests and simulation:
  Write tests and run them. Try to enhance the test suite by using the iterative process of analyzing coverage reports and adding additional tests to fill the uncovered areas. From time to time, update and optimize regression suite using the ranking capabilities. There is no need to frequently run tests that have only marginal contribution to the verification process.
  Note that from the beginning, the best tests are directed-random tests. In other words, tests should be targeted at a specific area, but anything that need not be specified should be randomized. By changing the random seed, each test can become thousands of tests, each testing the same target from different paths and randomizing data. This is the most efficient way to increase coverage and find bugs!
- Phase five-Code coverage integration:
  Once RTL code is mature enough, add in code coverage. Start with block coverage. Unreachable code should be carefully analyzed; it may save time to ask the implementer to identify the code's functionality. Dead code should be removed. In cases of reachable non-exercised logic, identify the untested scenario and write tests or constrain the test generator to fill coverage holes.

At the same time identify the untested functionality. Once identified, review of test plan should be done and make sure it is not an overlooked area in the test plan. Update the test plan and functional and assertion coverage code as necessary.

- Phase six-Regression testing: Throughout the process, regression suites that maximize code, functional and assertion coverage should be created. Regressions can be created per functional area or to fit timeslots, such as overnight or weekly regressions than may run 60 hours, potentially on multiple servers. It is critical to leverage compute and simulation resources to maximize coverage and find bugs faster.

5.4 Running and Tracking Regression:

In a coverage-driven verification program, typically there is a 10X or greater increase in the amount of simulations run on a daily basis. The increase in simulations is usually an optimization of existing resources so that they are more fully utilized. This way, tool licenses and computers which have grown accustomed to having nights and weekends off are exercised around the clock. This increase, while providing deeper coverage of the design and high quality bugs, also creates a whole bunch of information that needs to be managed. In order to track completion of test suites, parse and distribute failures, and verify the failures are fixed.

After regression is over we have to generate a list of failure types using Unix Shell scripts. This means that either based on error type or test name, test failures are assigned, and we have to analyze and find the bugs in eVC.

Following a nightly regression, all failures will be categorized by failure type, sorted by cycles-to-failure, and we have to debug and fix test bench problems, eVC bugs, VE bugs or if bug is critical we assign them to the eVC design team. When fixes are completed we change the status of that the fix was made. Then we have to validate each fix by rerunning the simulation in the subsequent regression and finally bug will be closed using bug tracking system.

5.4.1 Analyzing coverage:

As the project progresses, the focus shifts to analyzing the coverage in order to both focus effort on the coverage holes and in order to produce weekly indicators to increase accuracy. Coverage analysis is done on following basis:

- Defining priorities and weights per feature
- Defining perspective views for intermediate milestones and users interested in specific features
- Identifying the constraints that cause functionality not to be achieved

Specman Elite tool, facilitating a system where coverage can be prioritized, masked or viewed in different perspectives, will open up a whole new array of possibilities in managing phased projects. For example, if the first spin of the design is just for a demo, a smaller percentage of the functionality of the design needs to be validated to release the eVC. In that case 100% coverage can be defined by choosing just the coverage points are needed.

If the subsequent spin is defined to be more sensitive to time than features, the coverage set can defined to include only the priority-1 features, leaving the subsequent features to the next spin. Overall, the coverage can be monitored by different stakeholders based on their priorities and perspectives, while the verification team can focus on the goal at hand without having to design several separate plans.

5.4.2 Optimizing regressions

One of the bottlenecks toward the end of a coverage-driven program is the amount of computer time and license resources required to produce the coverage. A management automation tool should provide a means to identify optimal tests for achieving the coverage. This way the cycles that are run can be focused to help reach goals faster and more efficiently. Functional coverage is the best indicator of the efficiency of a random test. Therefore, running hundreds or thousands of tests which do not contribute to the coverage is likely not the best use of resources. To find optimal test suite, key tests are graded for their efficiency in providing coverage.

Redundant tests can be eliminated and certain tests can be marked for running only once, or for running multiple times. This enables finding the optimal regression for running in

random, a regression, which covers the broadest feature, base in the fewest tests. Running that regression in random is likely to be a significant savings in resources.

A tool which enables identification of the optimal test suite, based on the coverage perspectives described above, will also allow creation of several smaller regressions which can be used for multiple purposes. A feature-regression is a subset of all the tests, which covers an entire feature in a minimum number of runs.

This regression is run before a check-in of files modifying a feature's code. A mini-regression is a subset of highly stable tests that establish that each of the features is still alive. This is used before any top-level or modeling change. Also, like in the example from the previous section, a spin-1 regression can be defined to retain 100% of spin-1 coverage in case the team decides to re-release the eVC at a later date and wants to maintain the a consistent level of quality. Overall, the use of a management automation tool should drive many of the manual tasks done by the verification team and managers as well as open the doors to new possibilities in managing verification projects efficiently.

5.5 Analyzing bugs:

5.5.1 Manual Bug Analysis

In this process, the simulation results are checked with waveform viewer. The waveforms of output from dut will be regoursly checked for any of the protocol violence or any mismatch with expected output with reference to input. Figure shows the waveform for an illegal inter packet gap between two packets.

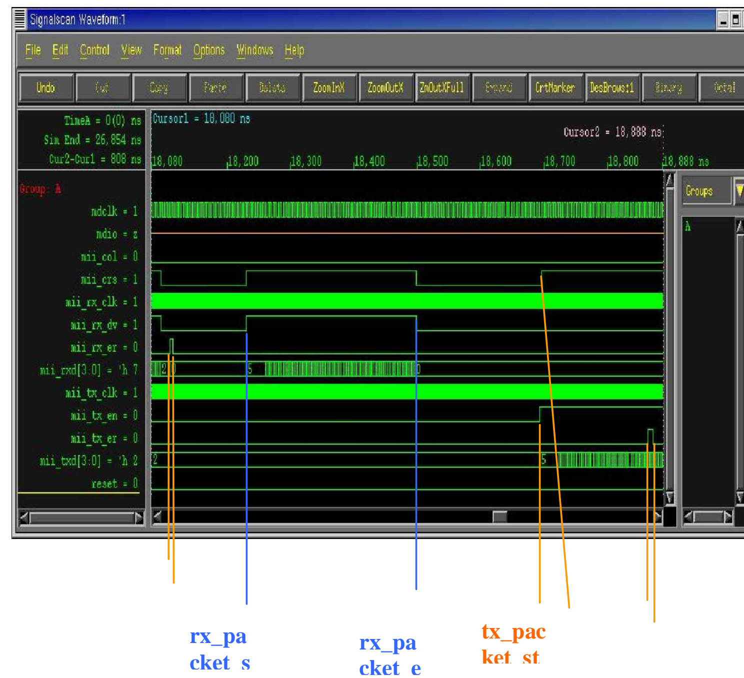


Figure 5.4: Illegal inter packet gap between two packets.

5.5.2 Automated bug analysis

When test scenarios are fired to eVC, the VE will expect some errors to be occurred as these errors are designed in test case itself and log the list of expected errors for further use. Now the output is observed for all the errors occurred. Then there is one matching mechanism which will match the occurred error with list of expected error and discard the error if it matches otherwise it gives message "error occurred but not expected" For the error which are expected but didn't occurred are displayed with message "error expected but not occurred" From this information of occurred and expected error, further analysis must be done to find out any bug in system or reason of unexpected behavior.

Once the bug is found, one should correct that bug locally and verify the behavior of corrected code and then it must be notified with GCVS Graphical Concurrent System Version. And this way that bug will be removed from entire system.

5.5.3 Checkers

To automate the bug analysis process, it is required to add more and more checkers. Checkers will give warning or error message when some predefined conditions are not satisfied or protocol violence happens, and from this message, the bug analysis can be done. But precise care should be taken while defining checkers otherwise checker itself will cause some misinterpretation.

5.5.3.1 Checking

Checking of a design can be done against:

- A set of rules: Here in this case we are checking DUT against a set of rules, specification. In our case we are following IEEE standards.
- A reference model (written in e, C, etc.), synchronized with a simulator.: for this purpose, we have designed a mock DUT in Verilog language which follows the rules defined by IEEE standard for Ethernet.

Generally Specman Elite can perform two types of checks:

- Data checks verifying data correctness
- Temporal checks verifying timing protocols

5.5.4 When to Check

Checking can be done:

- Post-run: After simulation is over, in the check phase
- On the fly: During the simulation run phase. This method is more fruitful as it allows analysis with full state of DUT, saves memory because less data is accumulated and also avoids wasted simulation time, because simulation stops on error.

5.5.5 Data Checking

To check the data coming out of the DUT the procedure is as follow:

1. Collect DUT output.
2. Convert the raw output data to a higher abstraction. (This step is optional but usually preferable.)
3. Check the converted data.

The first and third steps are obvious. Before you check, you must collect the output. Converting the raw data is in accordance with one of Specman s basic ideas - to work on a higher level of abstraction. We generate packets but inject bytes. Conversely, we collect bytes but check packets. Checking is easier when comparing on a higher abstraction level. It is easier to understand from error messages when the error occurred.

5.5.6 Data Check Constructs

Check action Syntax:

```
check that bool-exp [else dut_error(string)];
```

Example:

```
    check that '~/top/data_out' == '~/top/data_in'
          else dut_error( "DATA MISMATCH: ",
                "Expected : ", '~/top/data_in');
```

dut_error is a predefined method that specifies a DUT error with a message string. It also updates predefined corresponding fields: 1. num_of_dut_errors 2. num_of_dut_warnings. It should be noted that the check's effect on simulation is controllable.

1. check that: like an if statement, with the addition that:
    - You can control the effect of a failure on the simulation run.
    - Global error counters and flags are set by Specman.
2. dut_error(): like append(), accepts any number of parameters that are converted to strings and concatenated to form the error message

5.5.7 Data Checking Using Scoreboard

For comparing output to input, the scoreboard technique is recommended.:

1. Keep input in a scoreboard.
2. Compare collected output to matching input.

Following are examples of scoreboard checking for:

- Comparing raw data (byte by byte)
- Comparing at a higher abstraction level (struct fields)

We follow data checking at a higher abstraction level. We are using it at packet level. The architecture of scoreboard checking is as followed. A uid number is inserted in to a packet . The packet is stored in scoreboard with a given uid and inserted to DUT also. The output from DUT is matched for the same uid in scoreboard. If output uid matches with input uid then that packet and uid is discarded from scoreboard otherwise it gives scoreboard error.

Chapter 6

Results and Discussion

This chapter provides the simulation of some basic scenarios like normal, collision and zeroipg between two packets.

6.1 Normal Scenario:

Test case “test_gmii_normal” is able to generate normal scenario by firing packets of different lengths, types for Ethernet packets and Random data. As per the standard IEEE 802.3 data length of packet can be varied from 46 bytes to 1500 bytes. Number of packets to be fired can be varied by constraining *count* variable. All the information regarding which bfm has fired the packet, packet no., packet type, duplex mode, length of data, start time of packet, end time of packet etc. will be stored in a log file. Figure 6.1 shows the waveform for normal packet.

Figure 6.1: Simulation of Normal Scenario.

Only MAC bfm can fire packets on 'tx' line. It can be seen in above figure signal 'tx_en' will go high and data on txd line will be 55h (i.e. 10101010) for 7 clock cycles. Carrier sense signal will go high after one cycle if 'tx_en' signal is high. Monitor will collect these packets and coverage will be collected on event defined inside monitor. Figure 6.2 shows the collected coverage.

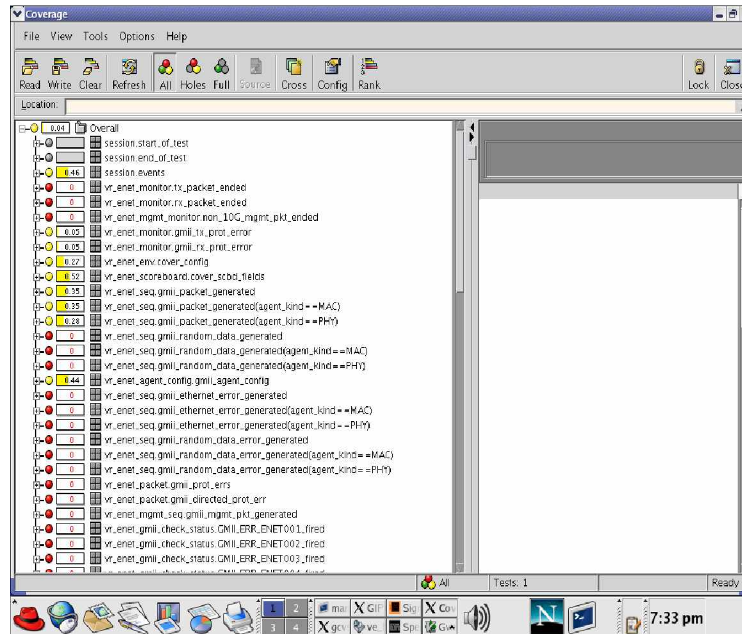


Figure 6.2: Coverage of Normal Scenario.

In above figure yellow fields shows partial collection with grades in percentage. Red fields show the hole, and green field shows full collection with 100% coverage.

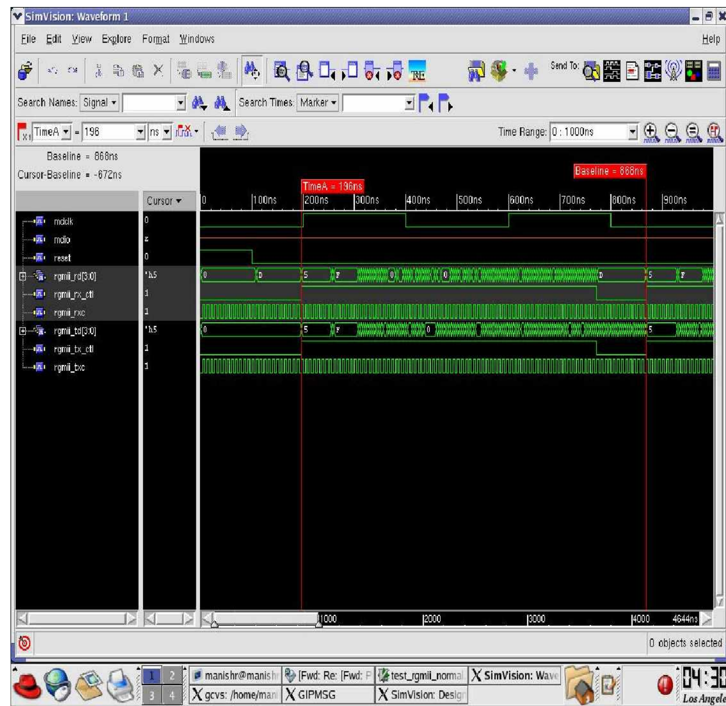


Figure 6.3: Simulation of Normal Packet scenario (RGMII interface).

6.2 Collision Scenario:

As per standard collision will occur if both the stations start transmission simultaneously. If we fire packets simultaneously through MAC and PHY bfm's then collision scenarios could be simulated. If collision occur in preamble phase of the packet then both station will complete preamble and then transmit 32-bit jam sequence (i.e. 4 bytes of data with pattern 11110010) to ensure occurrence of collision and then they wait for random time and start transmission again. Figure 6.4 shows waveform for collision condition.

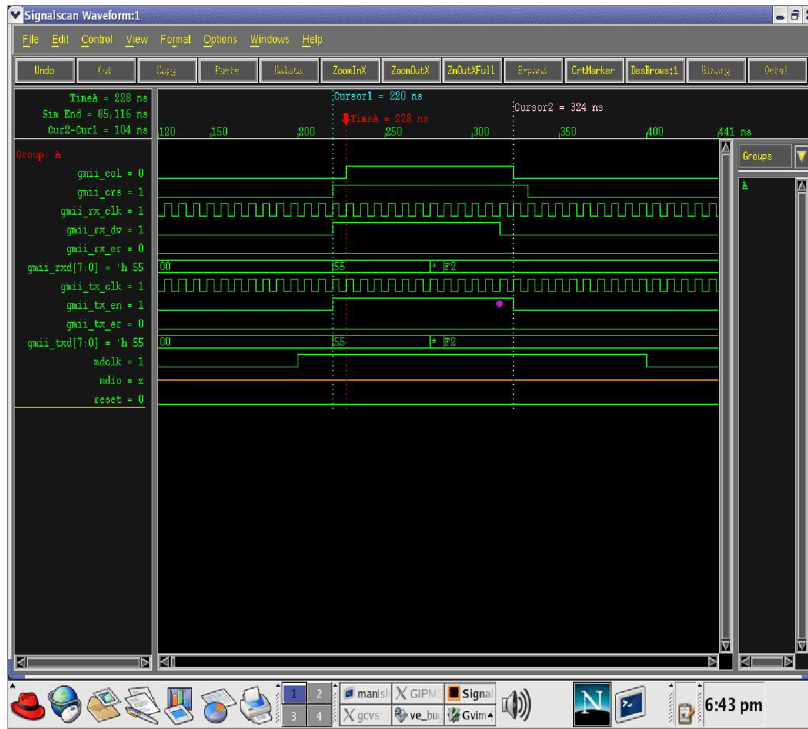


Figure 6.4: Simulation of Collision Scenario.

Figure 6.4 shows the fired check and collected coverage from test case simulating collision. It can be seen from the figure that checks 26,27,28 and 29, which are related to collision, has been covered.

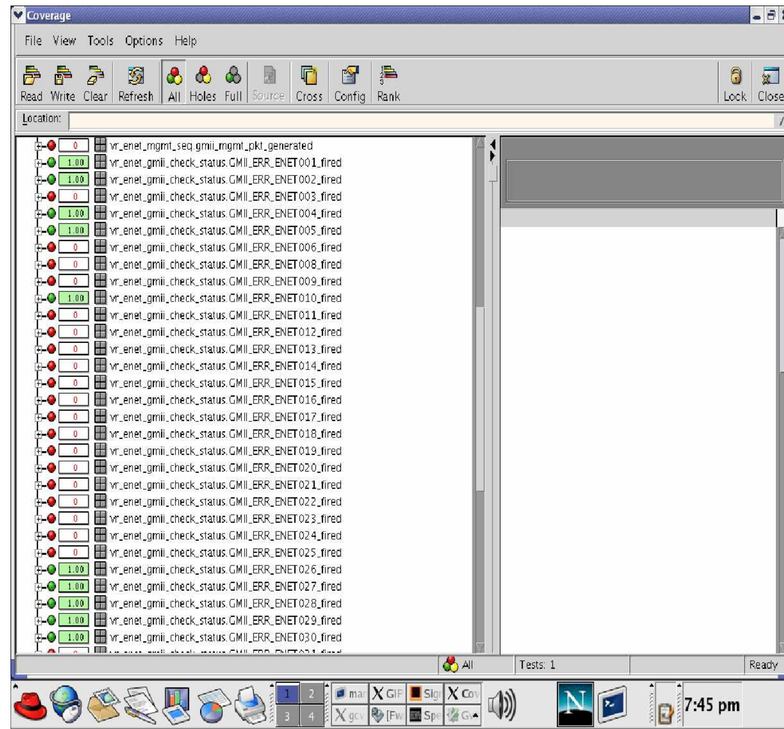


Figure 6.5: Coverage of Collision Scenario (GMII Interface).

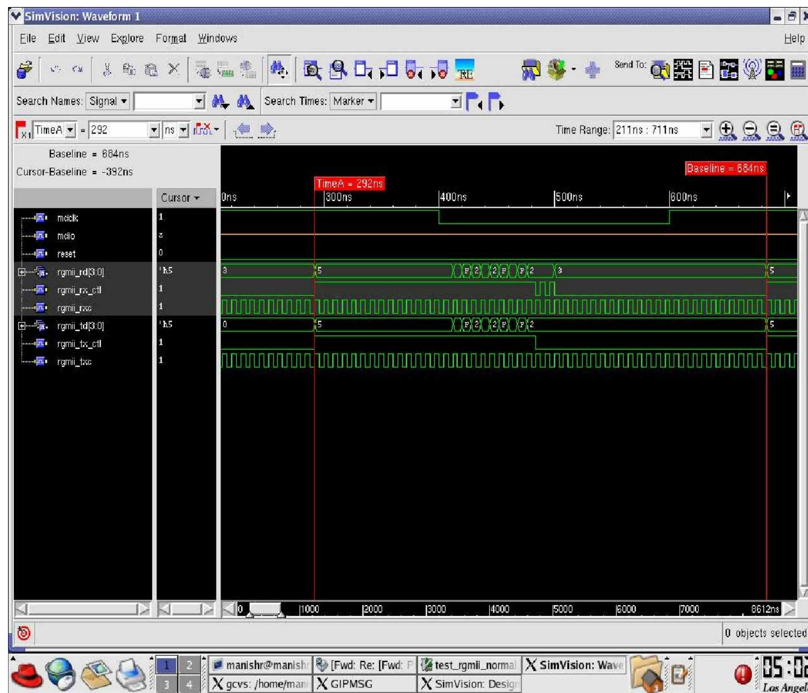


Figure 6.6: Simulating collision scenario (RGMII Interface).

6.3 Zeroipg check Scenario:

Test case “test_gmii_zeroipg_check” fires packets with zero inter frame gap. As per standard minimum inter frame gap between two packets should be 96-bit. We can fire packets with zero inter frame gap by constraining ipg field of packet to zero value or any value less then 96-bits. Figure 6.7 shows coverage of zeroipg scenario.

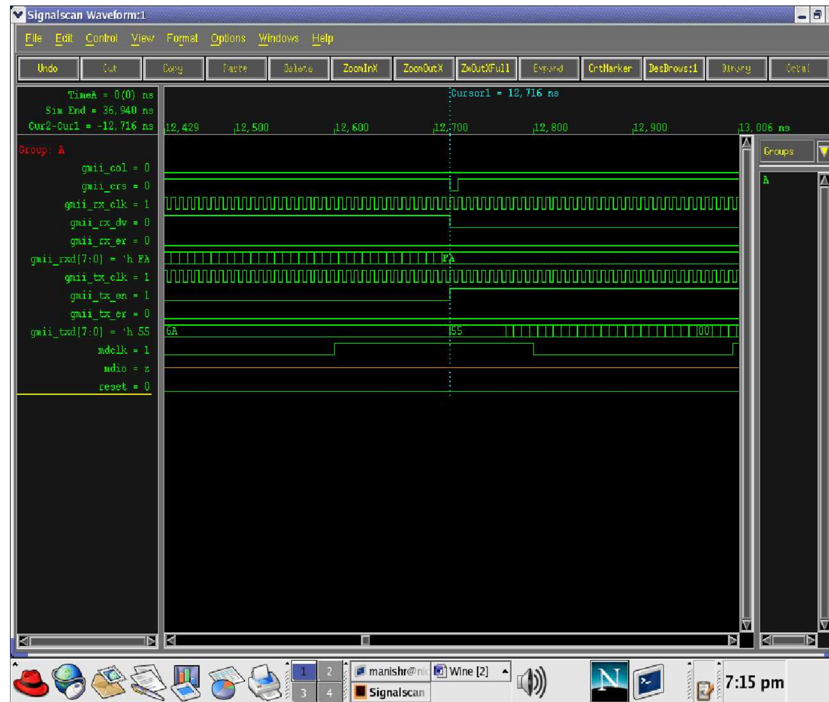


Figure 6.7: Zeroipg check scenario (GMII Interface).

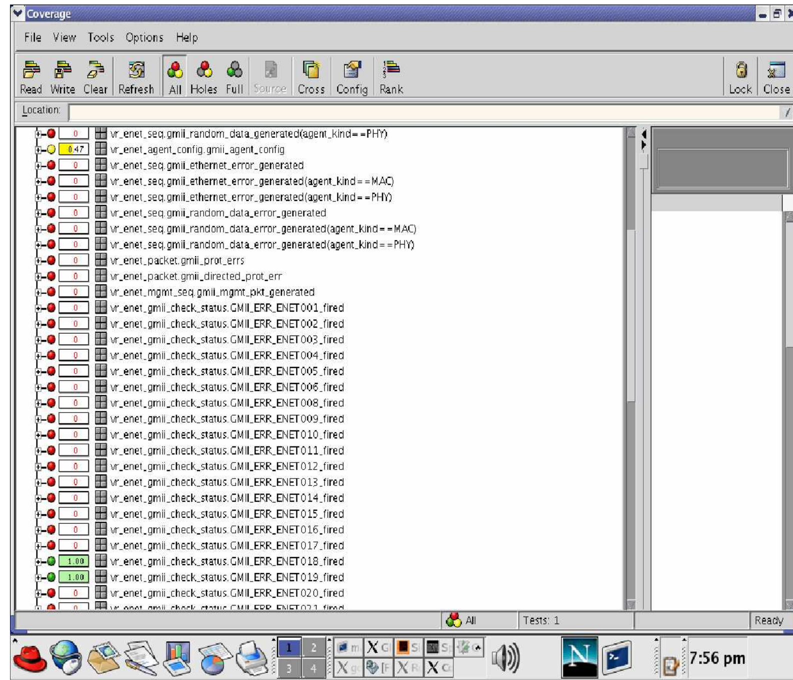


Figure 6.8: Coverage of Zeroipg check Scenario.

6.4 Injecting RX_ER in ipg phase of packet:

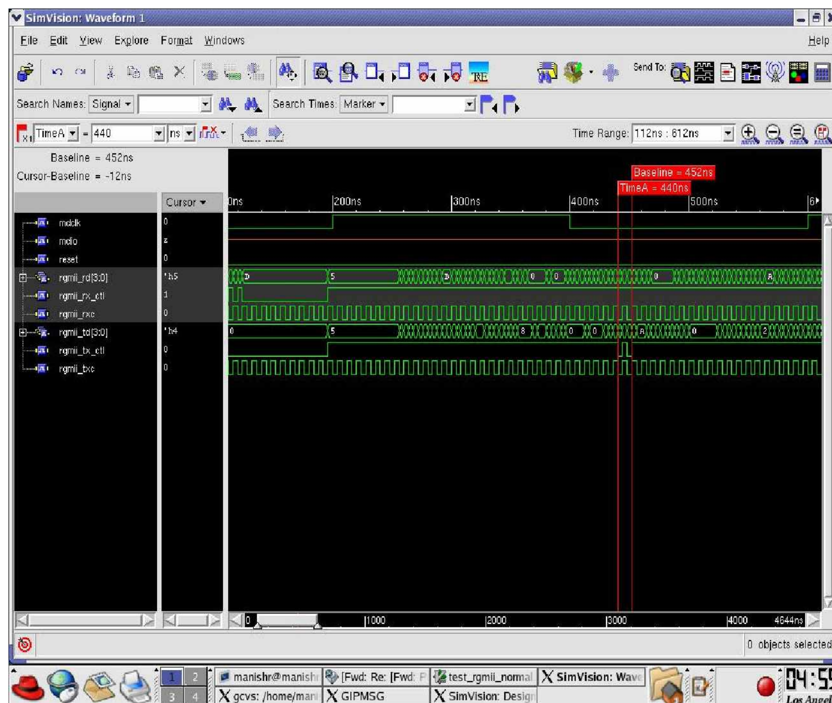


Figure 6.9: TX_ER insertion in Data Phase of Ethernet Packet (RGMII Interface).

Figure 6.10: RX_ER insertion in IPG phase of Ethernet Packet (RGMII Interface).

6.5 Results

1. No. of Bugs found in Verification Environment 21
2. No. of Bugs found in eVC 43
3. No. of test cases written 31
4. Coverage achieved in each interface

GMII	98%
RMII	97%
SGMII	98%
XSBI	94%
RGMII	92%

5. Automization done in each interface

GMII	90%
RGMII	70%
SGMII	80%
XSBI	70%

Chapter 7

Conclusion and Future Scope

7.1 Conclusion:

This project work has helped me in understanding Ethernet protocol and various interfaces used by it. I learned Unix operating system and e/Specman verification language during the training, which is prerequisite for verification.

Project work majorly focused on verification and designing of eVC (e Verification Component) using eRM (e Reusable Methodology). As we all know verification takes 70% of the whole design cycle, so using these kinds of ready-made verification component bugs can be found on early stages of design and hence can be fixed in same stages. This can increase probability of first time success of chip and also the time to market which is very stringent now-a-days.

7.2 Future Scope:

Ethernet protocol supports lot many interfaces, layered as well as non-layered interfaces. Now layered interface are evolving at speed of light so possibility of adding new interface to Ethernet eVC will always be there.

Also the methodology followed to implement the Ethernet eVC can be implemented in other environment too. It can be implemented using other Hardware Verification Languages like System C, Open Vera Assertion, and System Verilog etc.

References

1. IEEE Std 802.3, 2000 Edition “Carrier sense multiple access collision detection (CSMA/CD) access method and physical layer specification”.
2. ANSI/IEEE Std 802.3, “Carrier sense multiple access collision detection (CSMA/CD) access method and physical layer specification”, Fifth Edition, 1996.
3. IEEE Draft P802.3z/D4, “Media access control (MAC) parameters, physical layer, repeater and management parameters for 1000 Mbps operation”, December 1997.
4. R. M. Metcalfe and D. R. Boggs, “ Ethernet: Distributed Packet Switching for Local Computer Networks, Communications of the ACM” vol. 19, pp. 395-404, July 1976.
5. IEEE 802.1Q-in-Q VLAN Tag Termination
6. IEEE Std 802-1990 IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture
7. IEEE P802.3ae 10 Gigabit Ethernet Task Force http://grouper.ieee.org/groups/802/3/ae/
8. Ethernet in the First Mile Alliance http://www.efmalliance.org/
9. Gigabit Networking: High-Speed Routing and Switching, http://www.cse.ohio-state.edu/~jain/cis788-97/gigabit_nets/index.htm
10. IEEE 802.3 CSMA/CD (ETHERNET) Working Group Web site, http://grouper.ieee.org/groups/802/3/
11. Computer Networks by Tennanbaum.
12. Rich Seifert, “Gigabit Ethernet: Technology and Applications for High-Speed LAN’s” Addison-Wesley, May 1988.

Appendix A Test Cases

```
---------------------------------------------------
---------------------------------------------------
File name    : test_gmii_normal.e
Title        :
Project      : Gigabit Ethernet eVC
Created On   :
Developers   : eInfochips Ltd
Purpose      :
Description  :This testcase is used to fire normal packets
Assumptions  : none
Limitations  : none
Known Errors : none
Notes        :
---------------------------------------------------
---------------------------------------------------
```


```
Copyright(c)2000-2003 eInfochips. - All rights reserved
This software is authored by eInfochips and is eInfochips intellectual
property, including the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights,
including ownership rights, being retained by eInfochips. This file may not
be distributed, copied, or reproduced in any manner,
Electronic or otherwise, without the express written consent of eInfochips.
```


```
---------------------------------------------------
Revision History :
---------------------------------------------------


o Some specific sequences:

<'
import vr_enet_ve/tests/vr_enet_gmii_ve_config;
==================================================
extend vr_enet_env {
  crs_active_in_full_duplex : bool;
};

extend vr_enet_agent {
  keep config.crs_active_in_full_duplex ==
get_enclosing_unit(vr_enet_env).crs_active_in_full_duplex;
  keep config.has_log.reset_soft();
  keep soft config.has_log == select {
                               50: TRUE;
                               50: FALSE;
                               };
};

extend ETHERNET vr_enet_packet {
  keep soft data_length == select {
                                  25: [0,1,1499,1500];
                                  25: [43,44,45,46,47,2..6];
                                  20: [7..40,48..1495];
                                  40: [41,447,448,449,42,1496,1497..1498];
                                 };
};

extend ETHERNET vr_enet_packet {
  keep pause_quanta.reset_soft();
  keep soft pause_quanta == select {
                                   33: [0,1..10];
```

```
                                    33: [11..50];
                                    33: [51..65536];
                                     };
};
extend RANDOM_DATA vr_enet_packet {
  keep soft data_length == select {
                               33: [0,1,1499,1500];
                               33: [45,46,47];
                               33: [2..44,48..1498];
                               };
};


==================================================
extend MAIN vr_enet_mgmt_seq {
  keep soft count == 0;
};
==================================================
extend MAIN FALSE'random_data_mode vr_enet_seq {
  keep count ==20;
  keep sequence.kind in [NO_ERRORS,PAD_FRAME,NORMAL];
};

extend MAIN TRUE'random_data_mode vr_enet_seq {
  keep count ==20;
  keep sequence.kind == GMII_RANDOM_NORMAL;
};
'>
---------------------------------------------------
```


```
Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is
authored by eInfochips and is eInfochips intellectual property, including
the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights,
including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner,
electronic or otherwise, without the express written consent of eInfochips.
-------------------------------------------------
```


```
--------------------------------------------------
---------------------------------------------------
File name    : test_gmii_error.e
Title        :
Project      : Gigabit Ethernet eVC
Created On   :
Developers   : eInfochips Ltd
Purpose      :
Description  :This testcase is used to fire                 erroneous
packets
Assumptions  : none
Limitations  : none
Known Errors : none
Notes        :
---------------------------------------------------
---------------------------------------------------
```


```
Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is
authored by eInfochips and is eInfochips intellectual property, including
the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights,
including ownership rights, being retained by eInfochips. This file may not
be distributed, copied, or reproduced in any manner,
electronic or otherwise, without the express written consent of eInfochips.
---------------------------------------------------
```


```
Revision History :
---------------------------------------------------
```

o Some specific sequences:

```
<'
import vr_enet_ve/tests/vr_enet_gmii_ve_config;
==================================================
extend vr_enet_packet {
  keep packet_kind.reset_soft();
};
extend vr_enet_agent {
  keep config.retry_limit.reset_soft();
  keep soft config.retry_limit == select {
                                  25: 0;
                                  25: 1;
                                  25: [2..15];
                                  25: 16;
                                 };

  keep config.jam_length.reset_soft();
  keep soft config.jam_length == select {
                                 33: 0;
                                 33: [1..7];
                                 33: 8;
                                 };
};
==================================================
extend PHY MAIN FALSE'random_data_mode vr_enet_seq {
 keep count ==20;
 keep soft sequence.kind == select {
                                20:
[CRC_TEST,SHORT_FRAME_TEST,ILLEGAL_PAUSE_OPCODE,
SFD_TEST,GMII_PHY_RX_ER,GMII_PHY_CRS_DOWN_INJECT,GMII_LONG_FRAME_TEST,GMII_
FRAME_LENGTH_TEST,GMII_PHY_RX_CARRIER_EXT_ERR,GMII_PHY_IPG_RX_ER,
GMII_PHY_FALSE_CARRIER_INDICATION,GMII_MIN_RECEIVED_IPG_TEST];

                                40:
[GMII_ILLEGAL_IPG,GMII_ILLEGAL_PHY_PREAMBLE,GMII_ILLEGAL_FCI,GMII_IPG_RX_ER
RS,GMII_ILLEGAL_SOURCE_ADDRESS,GMII_ILLEGAL_EXTENSION,
GMII_ILLEGAL_FIRST_BYTE_PREAMBLE];

                                20:
[GMII_RX_ERRS,GMII_ERR_DOUBLE_VLAN_HEADER,GMII_ERR_VLAN_HEADER];

                                20: [ALIGNMENT_ERR,
PAUSE_OPCODE_ERR_IN_NON_PAUSE_PKT,LONG_SHORT_FRAME,
LENGTH_ERR_WITH_ALL, NO_BOOL_CONSTRINT,
LONG_SHORT_ALIGNMENT,ALIGNMENT_ERR_WITH_LENGTH_ERR,
ALIGNMENT_ERR_WITH_ILLEGAL_PAUSE_OPCODE,ALL_ERRS_TOGATHER];

                                };
};



extend MAC MAIN FALSE'random_data_mode vr_enet_seq {
  keep count ==20;
  keep soft sequence.kind == select {
                               20:
[CRC_TEST,SHORT_FRAME_TEST,ILLEGAL_PAUSE_OPCODE,SFD_TEST,GMII_LONG_FRAME_TE
ST, GMII_FRAME_LENGTH_TEST,GMII_MAC_TX_ER,
GMII_MAC_TX_CARRIER_EXT_ERR,GMII_MIN_RECEIVED_IPG_TEST];

                               40:
[GMII_ILLEGAL_IPG,GMII_ILLEGAL_MAC_PREAMBLE,GMII_POST_TX_ER,GMII_POST_TX_ER
```

```
RS,GMII_ILLEGAL_EXTENSION,GMII_ILLEGAL_SOURCE_ADDRESS,
GMII_ILLEGAL_FIRST_BYTE_PREAMBLE];

                                20:
[GMII_TX_ERRS,GMII_NO_DATA,GMII_MAX_DATA,
GMII_ERR_DOUBLE_VLAN_HEADER,GMII_ERR_VLAN_HEADER];

                                20: [ALIGNMENT_ERR,
PAUSE_OPCODE_ERR_IN_NON_PAUSE_PKT,LONG_SHORT_FRAME,
LENGTH_ERR_WITH_ALL,NO_BOOL_CONSTRINT,LONG_SHORT_ALIGNMENT,ALIGNMENT_ERR_WI
TH_LENGTH_ERR,
ALIGNMENT_ERR_WITH_ILLEGAL_PAUSE_OPCODE,ALL_ERRS_TOGATHER];
                                };

};

extend MAC MAIN random_data_mode vr_enet_seq {
  keep count == 5;
  keep sequence.kind in
[GMII_RANDOM_TX_ER,GMII_RANDOM_TX_CARRIER_EXTENSION_ERROR,GMII_RANDOM_NORMA
L];
};

extend PHY MAIN random_data_mode vr_enet_seq {
  keep count == 5;
  keep sequence.kind in
[GMII_RANDOM_RX_ER,GMII_RANDOM_RX_CARRIER_EXTENSION_ERROR,GMII_RANDOM_CRS_D
OWN_INJECT,GMII_RANDOM_RX_ER_FCI];
};
'>
===================================================
<'
extend vr_enet_agent {
  keep config.duplex_kind == HALF;
};
extend GMII vr_enet_system_ve {
  keep soft random_data_mode == select {60:FALSE;
                                       40:TRUE;
    };
};
'>
```


```
---------------------------------------------------
Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is
authored by eInfochips and is eInfochips intellectual property, including
the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights,
including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner,
electronic or otherwise, without the express written consent of eInfochips.
---------------------------------------------------
```


```
<'
package vr_enet;
import vr_enet_ve/tests/vr_enet_gmii_ve_config;
'>

<'
extend RANDOM_PROT_ERR vr_enet_packet {
    change_zero_timing() is {
      for each (e) in directed_prot_errs {
        if (e.timing == 0) {
          e.timing = 1;
```

```
        };
      };
    };
    post_generate() is also {
      change_zero_timing();
    };
};
extend vr_enet_system_ve {
  keep random_data_mode == FALSE;
};
extend vr_enet_agent{
   keep config.duplex_kind == HALF;
};

extend MAIN vr_enet_seq {
  keep count == 50;
  keep sequence.kind in
[TX_RX_IN_IPG,EXTENSION_ERR_IN_ALL,ILLEGAL_ERR_FROM_MAC_PHY,FCI_IN_ALL,HALF
_DUPLEX_ERR_IN_FULL,

NO_ERROR,SINGLE_ERROR, MULTIPLE_ERROR, MULTI_TIME_SINGLE_ERROR,
SINGLE_TIME_SINGLE_ERROR_PHASE,
SINGLE_TIME_SINGLE_ERROR,MULTI_TIME_SINGLE_ERROR_PHASE
,SINGLE_TIME_TWO_ERROR,MULTI_TIME_TWO_ERROR,
MULTI_TIME_TWO_ERROR_PHASE,SINGLE_TIME_TWO_ERROR_PHASE];
};

'>
===================================================
<'
extend vr_enet_seq_kind : [NO_ERROR,SINGLE_ERROR, MULTIPLE_ERROR,
MULTI_TIME_SINGLE_ERROR,
SINGLE_TIME_SINGLE_ERROR_PHASE,SINGLE_TIME_SINGLE_ERROR,MULTI_TIME_SINGLE_E
RROR_PHASE,SINGLE_TIME_TWO_ERROR,MULTI_TIME_TWO_ERROR,MULTI_TIME_TWO_ERROR_
PHASE, SINGLE_TIME_TWO_ERROR_PHASE];

extend SINGLE_ERROR vr_enet_seq  {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt;
  };
};

extend NO_ERROR vr_enet_seq  {
  body()@driver.clock is only {
      do NO_PROT_ERR pkt;
  };
};

extend MULTIPLE_ERROR vr_enet_seq  {
  body()@driver.clock is only {
      do MULTI_PROT_ERR_KIND pkt;
  };
};
extend MULTI_TIME_SINGLE_ERROR MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == TX_ER;
           e.err_occurrence == MULTI_TIME;
      };
  };
};
};

extend MULTI_TIME_SINGLE_ERROR PHY vr_enet_seq  {
```

```
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == RX_ER;
           e.err_occurrence == MULTI_TIME;
      };
  };
};
};

extend SINGLE_TIME_SINGLE_ERROR MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == TX_ER;
           e.err_occurrence == SINGLE_TIME;
         };
      };
  };
};

extend SINGLE_TIME_SINGLE_ERROR PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == RX_ER;
           e.err_occurrence == SINGLE_TIME;
         };
      };
  };
};

extend MULTI_TIME_SINGLE_ERROR_PHASE MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == TX_ER;
           e.err_occurrence == MULTI_TIME;
           e.err_phase not in [ABSOLUTE,CARRIER_EXTENSION];
      };
  };
};
};

extend MULTI_TIME_SINGLE_ERROR_PHASE PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == RX_ER;
           e.err_occurrence == MULTI_TIME;
           e.err_phase not in [ABSOLUTE,CARRIER_EXTENSION];
      };
  };
};
};

extend SINGLE_TIME_SINGLE_ERROR_PHASE MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == TX_ER;
           e.err_occurrence == SINGLE_TIME;
           e.err_phase not in [ABSOLUTE,CARRIER_EXTENSION];
         };
      };
```

```
  };
};

extend SINGLE_TIME_SINGLE_ERROR_PHASE PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  SINGLE_PROT_ERR_KIND pkt keeping {
         for each (e) in .protocol_errs {
           e.err_kind == RX_ER;
           e.err_occurrence == SINGLE_TIME;
           e.err_phase not in [ABSOLUTE,CARRIER_EXTENSION];
         };
      };
  };
};

extend SINGLE_TIME_TWO_ERROR MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_ER and e.err_occurrence ==
SINGLE_TIME;
          index == 1 => e.err_kind == TX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
    };
};
};

extend SINGLE_TIME_TWO_ERROR PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_ER and e.err_occurrence ==
SINGLE_TIME;
          index == 1 => e.err_kind == RX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

extend MULTI_TIME_TWO_ERROR MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_ER and e.err_occurrence ==
SINGLE_TIME;
          index == 1 => e.err_kind == TX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

extend MULTI_TIME_TWO_ERROR PHY vr_enet_seq  {
  body()@driver.clock is only {
    do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_ER and e.err_occurrence ==
SINGLE_TIME;
```

```
          index == 1 => e.err_kind == RX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

=============================================
extend SINGLE_TIME_TWO_ERROR_PHASE MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_ER and e.err_occurrence ==
SINGLE_TIME and
                        e.err_phase not in
[IPG,ABSOLUTE,RANDOM,CARRIER_EXTENSION];
          index == 1 => e.err_kind == TX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

extend SINGLE_TIME_TWO_ERROR_PHASE PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_ER and e.err_occurrence ==
SINGLE_TIME and
                        e.err_phase not in
[IPG,ABSOLUTE,RANDOM,CARRIER_EXTENSION];
          index == 1 => e.err_kind == RX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

extend MULTI_TIME_TWO_ERROR_PHASE MAC vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_ER and e.err_occurrence ==
SINGLE_TIME and
                        e.err_phase not in
[IPG,ABSOLUTE,RANDOM,CARRIER_EXTENSION];
          index == 1 => e.err_kind == TX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

extend MULTI_TIME_TWO_ERROR_PHASE PHY vr_enet_seq  {
  body()@driver.clock is only {
      do  MULTI_PROT_ERR_KIND pkt keeping {
        .protocol_errs.size() == 2 and
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_ER and e.err_occurrence ==
SINGLE_TIME and
                        e.err_phase not in
[IPG,ABSOLUTE,RANDOM,CARRIER_EXTENSION];
```

```
          index == 1 => e.err_kind == RX_CARRIER_EXTENSION_ERROR and
e.err_occurrence == SINGLE_TIME;
        };
      };
  };
};

===================================================
               --ILLEGAL SCENARIOS--
===================================================
extend vr_enet_seq_kind :
[TX_RX_IN_IPG,EXTENSION_ERR_IN_ALL,ILLEGAL_ERR_FROM_MAC_PHY,FCI_IN_ALL,HALF
_DUPLEX_ERR_IN_FULL];

extend TX_RX_IN_IPG MAC vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_ER and
e.err_occurrence.reset_soft() and
                        e.err_phase in [IPG,CARRIER_EXTENSION];
        };
      };
  };
};
extend TX_RX_IN_IPG PHY vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_ER and
e.err_occurrence.reset_soft() and
                        e.err_phase in [IPG,CARRIER_EXTENSION];
        };
      };
  };
};

============================================

extend EXTENSION_ERR_IN_ALL MAC vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == TX_CARRIER_EXTENSION_ERROR and
e.err_occurrence.reset_soft() and
                        e.err_phase not in [RANDOM, CARRIER_EXTENSION];
        };
      };
  };
};
extend EXTENSION_ERR_IN_ALL PHY vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == RX_CARRIER_EXTENSION_ERROR and
e.err_occurrence.reset_soft() and
                        e.err_phase not in [RANDOM,CARRIER_EXTENSION];
        };
      };
  };
};

=============================================

extend FCI_IN_ALL PHY vr_enet_seq {
```

```
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind == FALSE_CARRIER_INDICATION and
e.err_occurrence.reset_soft() and
                        e.err_phase not in [RANDOM,IPG];
        };
      };
  };
};

===============================================

extend ILLEGAL_ERR_FROM_MAC_PHY MAC vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind not in
[TX_ER,TX_CARRIER_EXTENSION_ERROR,INVALID_SYNC_HEADER,INVALID_BLOCK_TYPE_FI
ELD];
        };
      };
  };
};




extend ILLEGAL_ERR_FROM_MAC_PHY PHY vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind  in
[TX_ER,TX_CARRIER_EXTENSION_ERROR,INVALID_SYNC_HEADER,INVALID_BLOCK_TYPE_FI
ELD];
        };
      };
  };
};

==============================================

extend HALF_DUPLEX_ERR_IN_FULL MAC vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind  in
[TX_CARRIER_EXTENSION_ERROR,CRS_DOWN_INJECT,CRS_DOWN_COLLECT,COLLISION];
        };
      };
  };
};
extend HALF_DUPLEX_ERR_IN_FULL PHY vr_enet_seq {
  body()@driver.clock is only {
      do SINGLE_PROT_ERR_KIND pkt keeping {
        for each (e) in .protocol_errs {
          index == 0 => e.err_kind  in
[RX_CARRIER_EXTENSION_ERROR,CRS_DOWN_INJECT,CRS_DOWN_COLLECT,

FALSE_CARRIER_INDICATION,COLLISION];
        };
      };
  };
};
'>
```

```
---------------------------------------------------
---------------------------------------------------
File name     : test_gmii_collision.e
Title         :
Project       : Gigabit Ethernet eVC
Created On    :
Developers    : eInfochips Ltd
Purpose       :
Description   :This testcase is used to fire normal packets
Assumptions   : none
Limitations   : none
Known Errors  : none
Notes         :
---------------------------------------------------
---------------------------------------------------
Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is
authored by eInfochips and is eInfochips intellectual property, including
the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights,
including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner,
electronic or otherwise, without the express written consent of eInfochips.
---------------------------------------------------
Revision History :
---------------------------------------------------

<'
package vr_enet;
'>
o Some specific sequences:

<'
import vr_enet_ve/tests/vr_enet_gmii_ve_config;

--Always configure kind to MAC_DUT for collision testing.
extend vr_enet_system_ve {
  keep kind == MAC_DUT;
};
extend vr_enet_agent_ve {
  keep check_collision == TRUE;
};

--As collision is possible in HALF duplex mode only.
-- Random data mode is off.
extend vr_enet_agent{
   keep config.duplex_kind == HALF;
   keep config.random_data_mode == FALSE;
   keep config.jam_length == 8;
   keep config.retransmission_enable ==FALSE;
};

--Firing 5 normal ethernet packets from MAC agent.
extend MAC MAIN vr_enet_seq {
  keep count ==30;
  body()@driver.clock is only {
    for i from 1 to count {
      do pkt keeping {
        .data_length == 46 and
        .preamble.preamble_length == 56;
      };
    };
  };
};
```

```
--jam_length is time in cycle.
--TX_EN will go low after jam_length time when COL is asserted.
--If jam_length value is zero then TX_EN will go low before jam bits are
sent.
--If it is 1 then TX_EN will go low after 1 byte of jam.
--This is true when COL is not in preamble.
--Firing collision from PHY agent.

extend PHY MAIN vr_enet_seq {
--Method in bfm will be called depending upon value of this flag
-- If TRUE ,preamble_collision() method will be called upon
-- If FALSE,tx_er_collision() method will be called upon
select_method : bool;

  keep count ==30;
  body()@driver.clock is only {
    for i from 1 to count {
    do pkt keeping {
      .directed_prot_errs.size()==1 and
      .preamble.preamble_length == 56 and
      for each (e) in .directed_prot_errs {
        e.err_kind == COLLISION and
        e.timing == i-1 and
        e.err_phase == ABSOLUTE;
      };
    };
    };
  };

  mid_do(s:any_sequence_item) is also {
     for each (e) in pkt.directed_prot_errs {
       if(e.err_kind == COLLISION) {

         gen select_method ;

          if(select_method == TRUE) {
            start driver.as_a(GMII_FAMILY
vr_enet_driver).parent_agent.as_a(ACTIVE
vr_enet_agent).bfm.as_a(GMII_FAMILY PHY
vr_enet_bfm).preamble_collision(e.timing);
          }
          else {
            start driver.as_a(GMII_FAMILY
vr_enet_driver).parent_agent.as_a(ACTIVE
vr_enet_agent).bfm.as_a(GMII_FAMILY PHY
vr_enet_bfm).tx_er_collision(e.timing);
          };
      };--if
    };--for
  };--mid_do()
};

extend vr_enet_bfm {

  !jam_length : uint;
  !collision_timing : uint;

};

--They are used for only setting flags on which particular sets of errors
will be expected by VE
extend PHY GMII_FAMILY vr_enet_bfm {
  preamble_collision(d : uint) @clk is {

     collision_timing = d+1;
```

```
     gen jam_length keeping {it < parent_agent.config.jam_length/2};

     wait rise(gmii_smp.sig_GMII_RX_DV$);

     preamble_collision_expect_error();

     wait [jam_length]*cycle;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     sync
@sys.vr_enet_gmii_ve_env.system_ves[0].system.active_mac_agents[0].bfm.pack
et_started;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     release gmii_smp.sig_GMII_TX_EN ;
  };

   tx_er_collision(d : uint) @clk is {

     collision_timing = d+1;

     wait rise(gmii_smp.sig_GMII_COL$);

     trans_continues_expect_error();

     if(d < 7) {
       wait [11-(d+1)]*cycle;
     }
     else {
       wait [4]*cycle;
     };
     force gmii_smp.sig_GMII_TX_EN$ = 1;
     wait [1]*cycle;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     sync
@sys.vr_enet_gmii_ve_env.system_ves[0].system.active_mac_agents[0].bfm.pack
et_started;
     release gmii_smp.sig_GMII_TX_EN ;

 };
}; //extend



extend GMII_FAMILY vr_enet_bfm {

  preamble_collision_expect_error() is {

    if collision_timing <=7 {
      if(collision_timing < 7 and jam_length == 0) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET025");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET010");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET005");
      }
       else if (collision_timing + jam_length <= 7) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET026");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET010");
```

```
get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET005");
      }
      else if (collision_timing == 7 and jam_length != 0) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET029");
      }
      else {
        if(collision_timing + jam_length > 7 ) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET026");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
        }
        else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");
        };
      };
    }
    else {
      if jam_length==0 {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET028");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET004");
      }
      else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET029");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");

      };

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET004");
    };

  };

  trans_continues_expect_error() is {

    if(collision_timing < 7) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET027");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
```

```
    }
    else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET030");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
    };

  };
};
'>
```

```
---------------------------------------------------
```

Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is authored by eInfochips and is eInfochips intellectual property, including the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights, including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner, electronic or otherwise, without the express written consent of eInfochips.

```
---------------------------------------------------
```

```
---------------------------------------------------
---------------------------------------------------
File name    : test_gmii_collision.e
Title        :
Project      : Gigabit Ethernet eVC
Created On   :
Developers   : eInfochips Ltd
Purpose      :
Description  :This testcase is used to fire normal packets
Assumptions  : none
Limitations  : none
Known Errors : none
Notes        :
---------------------------------------------------
---------------------------------------------------
```

Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is authored by eInfochips and is eInfochips intellectual property, including the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights, including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner, electronic or otherwise, without the express written consent of eInfochips.

```
---------------------------------------------------
Revision History :
---------------------------------------------------
```

```
<'
package vr_enet;
'>
```

o Some specific sequences:

```
<'
import vr_enet_ve/tests/vr_enet_gmii_ve_config;

--always configure kind to MAC_DUT for collision testing.
extend vr_enet_system_ve {
  keep kind == MAC_DUT;
};
extend vr_enet_agent_ve {
  keep check_collision == TRUE;
};

--As collision is possible in HALF duplex mode only.
--Random data mode is off.
extend vr_enet_agent{
   keep config.duplex_kind == HALF;
   keep config.random_data_mode == FALSE;
   keep config.jam_length == 8;
   keep config.retransmission_enable ==FALSE;
};

--Firing 5 normal ethernet packets from MAC agent.
extend MAC MAIN vr_enet_seq {
  keep count ==30;
  body()@driver.clock is only {
    for i from 1 to count {
      do pkt keeping {
        .data_length == 46 and
        .preamble.preamble_length == 56;
      };
    };
  };
};


--jam_length is time in cycle.
--TX_EN will go low after jam_length time when COL is asserted.
--If jam_length value is zero then TX_EN will go low before jam bits are
sent.
--If it is 1 then TX_EN will go low after 1 byte of jam.
--This is true when COL is not in preamble.
--Firing collision from PHY agent.
extend PHY MAIN vr_enet_seq {


--Method in bfm will be called depending upon value of this flag
-- If TRUE ,preamble_collision() method will be called upon
-- If FALSE,tx_er_collision() method will be called upon
  select_method : bool;

  keep count ==30;
  body()@driver.clock is only {
    for i from 1 to count {
    do pkt keeping {
      .directed_prot_errs.size()==1 and
      .preamble.preamble_length == 56 and
      for each (e) in .directed_prot_errs {
        e.err_kind == COLLISION and
        e.timing == i-1 and
        e.err_phase == ABSOLUTE;
      };
    };
    };
  };
```

```
  mid_do(s:any_sequence_item) is also {
     for each (e) in pkt.directed_prot_errs {
       if(e.err_kind == COLLISION) {

         gen select_method ;

          if(select_method == TRUE) {
            start driver.as_a(GMII_FAMILY
vr_enet_driver).parent_agent.as_a(ACTIVE
vr_enet_agent).bfm.as_a(GMII_FAMILY PHY
vr_enet_bfm).preamble_collision(e.timing);
          }
          else {
            start driver.as_a(GMII_FAMILY
vr_enet_driver).parent_agent.as_a(ACTIVE
vr_enet_agent).bfm.as_a(GMII_FAMILY PHY
vr_enet_bfm).tx_er_collision(e.timing);
          };
      };--if
    };--for
  };--mid_do()
};

extend vr_enet_bfm {

  !jam_length : uint;
  !collision_timing : uint;

};

--They are used for only setting flags on which particular sets of errors
will be expected by VE
extend PHY GMII_FAMILY vr_enet_bfm {
  preamble_collision(d : uint) @clk is {

     collision_timing = d+1;
     gen jam_length keeping {it < parent_agent.config.jam_length/2};

     wait rise(gmii_smp.sig_GMII_RX_DV$);

     preamble_collision_expect_error();

     wait [jam_length]*cycle;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     sync
@sys.vr_enet_gmii_ve_env.system_ves[0].system.active_mac_agents[0].bfm.pack
et_started;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     release gmii_smp.sig_GMII_TX_EN ;
  };

   tx_er_collision(d : uint) @clk is {

     collision_timing = d+1;

     wait rise(gmii_smp.sig_GMII_COL$);

     trans_continues_expect_error();

     if(d < 7) {
       wait [11-(d+1)]*cycle;
     }
     else {
       wait [4]*cycle;
```

```
     };
     force gmii_smp.sig_GMII_TX_EN$ = 1;
     wait [1]*cycle;
     force gmii_smp.sig_GMII_TX_EN$ = 0;
     sync
@sys.vr_enet_gmii_ve_env.system_ves[0].system.active_mac_agents[0].bfm.pack
et_started;
     release gmii_smp.sig_GMII_TX_EN ;

 };
}; //extend



extend  GMII_FAMILY vr_enet_bfm {

  preamble_collision_expect_error() is {

    if collision_timing <=7 {
      if(collision_timing < 7 and jam_length == 0) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET025");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET010");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET005");
      }
       else if (collision_timing + jam_length <= 7) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET026");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET010");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET005");
      }
      else if (collision_timing == 7 and jam_length != 0) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET029");
      }
      else {
        if(collision_timing + jam_length > 7 ) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET026");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
        }
        else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");
        };
      };
    }
    else {
      if jam_length==0 {
```

```
get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET028");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET004");
      }
      else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET029");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");

      };

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET004");
    };

  };

  trans_continues_expect_error() is {

    if(collision_timing < 7) {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET027");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
    }
    else {

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET030");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET002");

get_enclosing_unit(vr_enet_system_ve).mac_active_agents[0].error_logger.exp
ected_errors.add("ERR_ENET001");
    };

  };
};
'>
```

Copyright (c) 2000-2003 eInfochips. - All rights reserved. This software is authored by eInfochips and is eInfochips intellectual property, including the copyrights in all countries in the world. This
software is provided under a license to use only with all other rights, including ownership rights, being retained by eInfochips.
This file may not be distributed, copied, or reproduced in any manner, electronic or otherwise, without the express written consent of eInfochips.

Appendix B Log Report

Normal test case log report:

```
Initializing Specman Elite (4.3.4)  -  Linked on Sun Aug 29 21:15:26 2004

      1. break on error
Breakpoint already exists:   1. break on error
Loading
/home/manishr/manish/test_cases/rgmii/test_rgmii_normal/test_rgmii_normal.
e
...
read...parse...update...patch...h code...code...clean...
Doing setup ...
191 checks were modified.
Generating the test using seed 1974560108...
[0] ENET_0 A_RGMII_MAC_0 MAC: Checking DUT signal connectivity in
RGMII_MAC_0
enet_evc_top.rgmii_0
[0] ENET_0 A_RGMII_PHY_0 PHY: Checking DUT signal connectivity in
RGMII_PHY_0
enet_evc_top.rgmii_0

-------------- ENET_0: vr_enet_env-@0
Verisity Ethernet eVC  -  version 2.0
No of ACTIVE MAC AGENTS  : 1
No of PASSIVE MAC AGENTS : 0
No of ACTIVE PHY AGENTS  : 1
No of PASSIVE PHY AGENTS : 0
-------------- E path: sys.vr_enet_rgmii_env


All sequence drivers:
----------------------
       driver               sent       pending        current
  0.  vr_enet_driver-@1      0          0              -
  1.  vr_enet_driver-@2      0          0              -

Starting the test ...
Running the test ...
[0] ENET_0: Checking DUT reset signal connectivity in environment ENET_0
[0] ENET_0: Checking signal RESET  Environment name ENET_0 with value
reset
[0] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) Starting MAIN vr_enet_seq-@3
[0] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) Starting MAIN vr_enet_seq-@4
Running should now be initiated from the simulator side
To complete waveform setup, execute in simulator prompt the command file
sn_wave_simvision.sv
Please load the file: 'sn_wave_test_rgmii_normal.sv' into the viewer after
the
simulation is completed.
Doing garbage collection: current size is 85337136 bytes ...
Done - new size is 66636012 bytes.
[0] ENET_0: Reset was asserted
[0] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@3 quit
[0] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@4 quit
[4] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) rerunning drvr 0
[4] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) Starting MAIN vr_enet_seq-@19
[4] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) rerunning drvr 1
[4] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) Starting MAIN vr_enet_seq-@20
[100] ENET_0: Reset was deasserted
```

```
[104] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_SPEED_MODE_MISMATCH:
Detected speed mode from the bus is SPEED_MODE_10MBPS
while expected speed mode is SPEED_MODE_1GBPS
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed

[104] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_DUPLEX_STATUS_MISMATCH:
Detected duplex status from the bus is HALF
while expected duplex status is FULL
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates duplex status

[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@20: Executing
default
body() method: doing 2 sequences
[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@21 created
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@19: Executing
default
body() method: doing 2 sequences
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@22 created
[108] ENET_0 A_RGMII_PHY_0 PHY: 0 error validation rules violated...
[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@23 created
[108] ENET_0 A_RGMII_MAC_0 MAC: 0 error validation rules violated...
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@24 created
[196] ENET_0 A_RGMII_PHY_0 PHY BFM : Started sending packet #0 ETHERNET
INJECT
vr_enet_packet-@23
[196] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #0 ETHERNET
INJECT
vr_enet_packet-@24
[212] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Started collecting packet #0
ETHERNET COLLECT vr_enet_packet-@25
[212] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #0
ETHERNET COLLECT vr_enet_packet-@26
[772] ENET_0 A_RGMII_PHY_0 PHY BFM : Finished sending packet #0 UID:
0xc8100000 ETHERNET INJECT vr_enet_packet-@23
[772] ENET_0 A_RGMII_PHY_0 PHY BFM :
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :VLAN_TAG
    Duplex mode             :FULL
    Packet number           :0
    Length of data          :20
    Start time              :196
    End_time                :772
    IPG in bit time         :96
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0xda5efe190392
    Tag Protocol ID         :33024
    User Priority           :0
    Canonical Indicator     :0
    VLAN Identifier         :2
    Length/Type value       :20
    Pad Size                :22
    Actual crc              :0x9fcb09e5
============================================
[772] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@23 sent by drvr 1
[772] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #0 UID:
0x48000000 ETHERNET INJECT vr_enet_packet-@24
[772] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
```

```
    Duplex mode             :FULL
    Packet number           :0
    Length of data          :20
    Start time              :196
    End_time                :772
    IPG in bit time         :96
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0x243b421a5646
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x22d11e1e
============================================
[772] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_tx_ok_get is updated to
the
value : 1
[772] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  brd_frm_tx_ok_get is
updated to
the value : 1
[772] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  multi_frm_tx_ok_get is
updated
to the value : 1
[772] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@24 sent by drvr 0
[772] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@21 done
[772] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@27 created
[772] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@22 done
[772] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@28 created
[772] ENET_0 A_RGMII_PHY_0 PHY: 0 error validation rules violated...
[772] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@29 created
[772] ENET_0 A_RGMII_MAC_0 MAC: 0 error validation rules violated...
[772] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@30 created


-------------------------------------------------------

   *** Dut warning at time 776
        Checked at line 96 in @vr_enet_rgmii_checker
        In MONITOR GMII_FAMILY
vr_enet_rgmii_layer-@6.sn___phy_status_in_ipg_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor.GMII_FAMILY'RGMII'rgmii
_monitor):

ERR_ENET044_PHY_RGMII_DUPLICATION_MISMATCH:
Data is not duplicated at positive and negative cycles.
Data on positive edge RXD[3:0] is 0x05
while data on negative edge RXD[7:4] is 0x0e
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed


-------------------------------------------------------
Will continue execution (check effect is WARNING)

[788] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Finished collecting packet #0
UID:
0xc8100000 ETHERNET COLLECT vr_enet_packet-@25
[788] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR:
==========Packet Information=============
    Packet type             :ETHERNET_802_3
    Tag kind                :VLAN_TAG
    Duplex mode             :FULL
    Packet number           :0
    Length of data          :20
    Start time              :212
    End_time                :788
```

```
    IPG in bit time         :104
    Preamble size in bits  :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0xda5efe190392
    Tag Protocol ID         :33024
    User Priority           :0
    Canonical Indicator     :0
    VLAN Identifier         :2
    Length/Type value       :20
    Pad Size                :22
    Actual crc              :0x9fcb09e5
============================================
[788] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #0
UID:
0x48000000 ETHERNET COLLECT vr_enet_packet-@26
[788] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :FULL
    Packet number           :0
    Length of data          :20
    Start time              :212
    End_time                :788
    IPG in bit time         :104
    Preamble size in bits  :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0x243b421a5646
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x22d11e1e
============================================
[788] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_rcd_ok_get is updated
to
the value : 1
[788] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  brd_frm_rcd_ok_get is
updated
to the value : 1
[868] ENET_0 A_RGMII_PHY_0 PHY BFM : Started sending packet #1 ETHERNET
INJECT
vr_enet_packet-@29
[868] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #1 ETHERNET
INJECT
vr_enet_packet-@30
[884] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Started collecting packet #1
ETHERNET COLLECT vr_enet_packet-@31
[884] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #1
ETHERNET COLLECT vr_enet_packet-@32
[1444] ENET_0 A_RGMII_PHY_0 PHY BFM : Finished sending packet #1 UID:
0xc8100001 ETHERNET INJECT vr_enet_packet-@29
[1444] ENET_0 A_RGMII_PHY_0 PHY BFM :
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :FULL
    Packet number           :1
    Length of data          :20
    Start time              :868
    End_time                :1444
    IPG in bit time         :96
    Preamble size in bits  :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
```

```
    Src. Address           :0x1497b445d806
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0xe021ebd2
============================================
[1444] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@29 sent by drvr 1
[1444] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #1 UID:
0x48000001 ETHERNET INJECT vr_enet_packet-@30
[1444] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :FULL
    Packet number          :1
    Length of data         :20
    Start time             :868
    End_time               :1444
    IPG in bit time        :96
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0x54c1e2508d68
    Src. Address           :0x4edb4a8239d4
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0x87a37b48
============================================
[1444] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_tx_ok_get is updated
to
the value : 2
[1444] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@30 sent by drvr 0
[1444] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@27 done
[1444] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@28 done


-------------------------------------------------------

   *** Dut warning at time 1448
        Checked at line 96 in @vr_enet_rgmii_checker
        In MONITOR GMII_FAMILY
vr_enet_rgmii_layer-@6.sn___phy_status_in_ipg_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor.GMII_FAMILY'RGMII'rgmii
_monitor):

ERR_ENET044_PHY_RGMII_DUPLICATION_MISMATCH:
Data is not duplicated at positive and negative cycles.
Data on positive edge RXD[3:0] is 0x02
while data on negative edge RXD[7:4] is 0x0d
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed


-------------------------------------------------------
Will continue execution (check effect is WARNING)

[1460] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Finished collecting packet #1
UID:
0xc8100001 ETHERNET COLLECT vr_enet_packet-@31
[1460] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR:
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :FULL
    Packet number          :1
    Length of data         :20
    Start time             :884
    End_time               :1460
```

```
    IPG in bit time        :96
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0xffffffffffff
    Src. Address           :0x1497b445d806
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0xe021ebd2
============================================
[1460] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #1
UID:
0x48000001 ETHERNET COLLECT vr_enet_packet-@32
[1460] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :FULL
    Packet number          :1
    Length of data         :20
    Start time             :884
    End_time               :1460
    IPG in bit time        :96
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0x54c1e2508d68
    Src. Address           :0x4edb4a8239d4
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0x87a37b48
============================================
[1460] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_rcd_ok_get is updated
to
the value : 2
[1460] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  brd_frm_rcd_ok_get is
updated
to the value : 2
[4644] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@20 ended
[4644] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@19 ended
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test ...
[4644] ENET_0 A_RGMII_MAC_0 MAC SCOREBOARD:

Total number of errors are 0

+++++++++++++++++++++++++++++++++++++++++++++++++


[4644] ENET_0 A_RGMII_PHY_0 PHY SCOREBOARD:

Total number of errors are 0

+++++++++++++++++++++++++++++++++++++++++++++++++


Checking is complete - 0 DUT errors, 2 DUT warnings.
***************************************************************
Finished an Ethernet test with stop condition
OBJ_MECH_IDLE_CYCLES_TOGETHER
***************************************************************
eVC name:                   ENET_0
Test time:                  4644
***************************************************************
[4644] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Number of packets collected
----------------------------------------------------------------------
```

```
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
----------------------------------------------------------------------
[4644] ENET_0 A_RGMII_MAC_0 MAC BFM : Number of packets injected
----------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
----------------------------------------------------------------------
[4644] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Number of packets collected
---------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
----------------------------------------------------------------------
[4644] ENET_0 A_RGMII_PHY_0 PHY BFM : Number of packets injected
----------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
----------------------------------------------------------------------
Wrote 1 cover_struct to test_rgmii_normal_1974560108.ecov




All sequence drivers:
----------------------
       driver               sent       pending        current
  0.  vr_enet_driver-@1      2           0              -
  1.  vr_enet_driver-@2      2           0              -


**** Specman - finishing session:
config run -exit_on == normal_stop (or all); exiting...
```

Collision test case log report:

```
Initializing Specman Elite (4.3.4)  -  Linked on Sun Aug 29 21:15:26 2004
```

```
      1. break on error
Breakpoint already exists:   1. break on error
Loading
/home/manishr/manish/test_cases/rgmii/test_rgmii_normal/test_rgmii_normal.
e
...
read...parse...update...patch...h code...code...clean...
Doing setup ...
191 checks were modified.
Generating the test using seed 1110103410...
[0] ENET_0 A_RGMII_MAC_0 MAC: Checking DUT signal connectivity in
RGMII_MAC_0
enet_evc_top.rgmii_0
[0] ENET_0 A_RGMII_PHY_0 PHY: Checking DUT signal connectivity in
RGMII_PHY_0
enet_evc_top.rgmii_0

-------------- ENET_0: vr_enet_env-@0
Verisity Ethernet eVC  -  version 2.0
No of ACTIVE MAC AGENTS  : 1
No of PASSIVE MAC AGENTS : 0
No of ACTIVE PHY AGENTS  : 1
No of PASSIVE PHY AGENTS : 0
-------------- E path: sys.vr_enet_rgmii_env


All sequence drivers:
----------------------
       driver               sent       pending        current
  0.  vr_enet_driver-@1      0           0              -
  1.  vr_enet_driver-@2      0           0              -

Starting the test ...
Running the test ...
[0] ENET_0: Checking DUT reset signal connectivity in environment ENET_0
[0] ENET_0: Checking signal RESET  Environment name ENET_0 with value
reset
[0] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) Starting MAIN vr_enet_seq-@3
[0] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) Starting MAIN vr_enet_seq-@4
Running should now be initiated from the simulator side
To complete waveform setup, execute in simulator prompt the command file
sn_wave_simvision.sv
Please load the file: 'sn_wave_test_rgmii_normal.sv' into the viewer after
the
simulation is completed.
Doing garbage collection: current size is 85531036 bytes ...
Done - new size is 66671040 bytes.
[0] ENET_0: Reset was asserted
[0] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@3 quit
[0] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@4 quit
[4] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) rerunning drvr 0
[4] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) Starting MAIN vr_enet_seq-@19
[4] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) rerunning drvr 1
[4] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) Starting MAIN vr_enet_seq-@20
[100] ENET_0: Reset was deasserted
[104] ENET_0 A_RGMII_MAC_0 MAC:

PHY_RGMII_LINK_STATUS_MISMATCH:
Link status indicated as down by RGMII PHY.
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates link status

[104] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_SPEED_MODE_MISMATCH:
Detected speed mode from the bus is SPEED_MODE_10MBPS
```

```
while expected speed mode is SPEED_MODE_100MBPS
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed

[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@20: Executing
default
body() method: doing 2 sequences
[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@21 created
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@19: Executing
default
body() method: doing 2 sequences
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@22 created
[108] ENET_0 A_RGMII_PHY_0 PHY: 0 error validation rules violated...
[108] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@23 created
[108] ENET_0 A_RGMII_MAC_0 MAC: 0 error validation rules violated...
[108] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@24 created
[108] ENET_0 A_RGMII_PHY_0 PHY BFM : Injecting RX_ER
vr_enet_directed_prot_err
[112] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_SPEED_MODE_MISMATCH:
Detected speed mode from the bus is SPEED_MODE_1GBPS
while expected speed mode is SPEED_MODE_100MBPS
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed

[112] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_DUPLEX_STATUS_MISMATCH:
Detected duplex status from the bus is FULL
while expected duplex status is HALF
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates duplex status

[120] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_SPEED_MODE_MISMATCH:
Detected speed mode from the bus is SPEED_MODE_1GBPS
while expected speed mode is SPEED_MODE_100MBPS
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates RXC clock speed

[120] ENET_0 A_RGMII_MAC_0 MAC:
PHY_RGMII_DUPLEX_STATUS_MISMATCH:
Detected duplex status from the bus is FULL
while expected duplex status is HALF
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates duplex status



-------------------------------------------------------

   *** Dut warning at time 124
        Checked at line 152 in @vr_enet_rgmii_checker
        In vr_enet_monitor-@16.sn___phy_mii_rx_er_chk_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor):

ERR_ENET031_PHY_MII_FALSE_CRS_INDICATION:
RX_ER signal has been asserted when RX_DV signal was de-asserted by MII
PHY,
without indicating False Carrier Indication.
Specs(IEEE-802.3,2000): 22.2.2.8

-------------------------------------------------------
Will continue execution (check effect is WARNING)



-------------------------------------------------------

   *** Dut warning at time 132
        Checked at line 152 in @vr_enet_rgmii_checker
```

```
        In vr_enet_monitor-@16.sn___phy_mii_rx_er_chk_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor):

ERR_ENET031_PHY_MII_FALSE_CRS_INDICATION:
RX_ER signal has been asserted when RX_DV signal was de-asserted by MII
PHY,
without indicating False Carrier Indication.
Specs(IEEE-802.3,2000): 22.2.2.8

-------------------------------------------------------
Will continue execution (check effect is WARNING)

[292] ENET_0 A_RGMII_PHY_0 PHY BFM : Started sending packet #0 ETHERNET
INJECT
vr_enet_packet-@23
[292] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #0 ETHERNET
INJECT
vr_enet_packet-@24
[308] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Started collecting packet #0
ETHERNET COLLECT vr_enet_packet-@25
[308] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #0
ETHERNET COLLECT vr_enet_packet-@26
[316] ENET_0 A_RGMII_PHY_0 PHY: Collision is detected on the bus
[484] ENET_0 A_RGMII_PHY_0 PHY BFM : Finished sending packet #0 UID:
0xc8100000 ETHERNET INJECT vr_enet_packet-@23
[484] ENET_0 A_RGMII_PHY_0 PHY BFM :
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :HALF
    Packet number          :0
    Length of data         :20
    Start time             :292
    End_time               :484
    IPG in bit time        :96
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0xffffffffffff
    Src. Address           :0xf4d6e69b2c41
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0x4a12a3b
==========Packet Error Information==========
      Error type        : RX_ER
      Error phase       : IPG
      Start time        : 0
      Error length      : 2
      -------------------
============================================
[484] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@23 sent by drvr 1
[484] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #0 UID:
0x48000000 ETHERNET INJECT vr_enet_packet-@24
[484] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :HALF
    Packet number          :0
    Length of data         :20
    Start time             :292
    End_time               :484
    IPG in bit time        :96
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0xffffffffffff
```

```
    Src. Address           :0xee4e3f148e91
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0x36e8a1ae
==========Packet Error Information==========
      Error type        : TX_ER
      Error phase       : DATA
      Start time        : 0
      Error length      : 2
      ------------------
============================================
[484] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@21 done
[484] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@27 created
[484] ENET_0 A_RGMII_PHY_0 PHY: 0 error validation rules violated...
[484] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@28 created
[484] ENET_0 A_RGMII_PHY_0 PHY BFM : Injecting RX_ER
vr_enet_directed_prot_err
[488] ENET_0 A_RGMII_MAC_0 MAC:

PHY_RGMII_LINK_STATUS_MISMATCH:
Link status indicated as down by RGMII PHY.
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates link status

[496] ENET_0 A_RGMII_MAC_0 MAC:

PHY_RGMII_LINK_STATUS_MISMATCH:
Link status indicated as down by RGMII PHY.
Specs(HP-RGMII,Version2.0 Specs): Table-4 Indicates link status



-------------------------------------------------------


   *** Dut warning at time 500
        Checked at line 152 in @vr_enet_rgmii_checker
        In vr_enet_monitor-@16.sn___phy_mii_rx_er_chk_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor):

ERR_ENET031_PHY_MII_FALSE_CRS_INDICATION:
RX_ER signal has been asserted when RX_DV signal was de-asserted by MII
PHY,
without indicating False Carrier Indication.
Specs(IEEE-802.3,2000): 22.2.2.8

-------------------------------------------------------
Will continue execution (check effect is WARNING)

[500] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The Source address
could
not be unpacked due to lack of bits
[500] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The Length/Type
field
could not be unpacked due to lack of bits
[500] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The CRC could not be
unpacked due to lack of bits
[500] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Finished collecting packet #0
CRC:
0xc49b92d9 ETHERNET COLLECT vr_enet_packet-@25
[500] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR:
==========Packet Information=============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :HALF
    Packet number          :0
    Length of data         :0
```

```
    Calculated CRC          :0x9d64c953
    Start time              :308
    End_time                :500
    IPG in bit time         :0
    Preamble size in bits  :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0x000000000000
    Src. Address            :0x000000000000
    Length/Type value       :0
    Pad Size                :0
    Actual crc              :0x0
==========Packet Error Information==========
      Error type        : CRC_ERROR
      -------------------
      Error type        : SHORT_FRAME_ERROR
      -------------------
      Error type        : COLLISION
      Error phase       : ABSOLUTE
      Start time        : 0
      -------------------
============================================
[500] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The Source address
could
not be unpacked due to lack of bits
[500] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The Length/Type
field
could not be unpacked due to lack of bits
[500] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The CRC could not be
unpacked due to lack of bits
[500] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #0
CRC:
0xc49b92d9 ETHERNET COLLECT vr_enet_packet-@26
[500] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :0
    Length of data          :0
    Calculated CRC          :0x9d64c953
    Start time              :308
    End_time                :500
    IPG in bit time         :0
    Preamble size in bits  :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0x000000000000
    Src. Address            :0x000000000000
    Length/Type value       :0
    Pad Size                :0
    Actual crc              :0x0
==========Packet Error Information==========
      Error type        : CRC_ERROR
      -------------------
      Error type        : SHORT_FRAME_ERROR
      -------------------
      Error type        : COLLISION
      Error phase       : ABSOLUTE
      Start time        : 0
      -------------------
============================================
[500] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_chk_seq_err_get is
updated
to the value : 1
```

```
-------------------------------------------------------

   *** Dut warning at time 500
        Checked at line 41 in @vr_enet_data_check
        In vr_enet_monitor-@15.check_tx_packet() (unit:
sys.vr_enet_rgmii_env.active_phy_agents[0].monitor):

ERR_ENET001_MAC_BAD_CRC:
MAC has transmitted a packet with BAD CRC.
Specs(IEEE-802.3,2000): 3.4

-------------------------------------------------------
Will continue execution (check effect is WARNING)



-------------------------------------------------------

   *** Dut warning at time 500
        Checked at line 46 in @vr_enet_data_check
        In vr_enet_monitor-@15.check_tx_packet() (unit:
sys.vr_enet_rgmii_env.active_phy_agents[0].monitor):

ERR_ENET002_MAC_SHORT_FRAME:
MAC has transmitted a packet with SHORT FRAME.
Specs(IEEE-802.3,2000): 4.2.3.3

-------------------------------------------------------
Will continue execution (check effect is WARNING)



-------------------------------------------------------

   *** Dut warning at time 508
        Checked at line 152 in @vr_enet_rgmii_checker
        In vr_enet_monitor-@16.sn___phy_mii_rx_er_chk_eval() (unit:
sys.vr_enet_rgmii_env.active_mac_agents[0].monitor):

ERR_ENET031_PHY_MII_FALSE_CRS_INDICATION:
RX_ER signal has been asserted when RX_DV signal was de-asserted by MII
PHY,
without indicating False Carrier Indication.
Specs(IEEE-802.3,2000): 22.2.2.8

-------------------------------------------------------
Will continue execution (check effect is WARNING)

[684] ENET_0 A_RGMII_PHY_0 PHY BFM : Started sending packet #1 ETHERNET
INJECT
vr_enet_packet-@28
[684] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #0 ETHERNET
INJECT
vr_enet_packet-@24
[700] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Started collecting packet #1
ETHERNET COLLECT vr_enet_packet-@29
[700] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #1
ETHERNET COLLECT vr_enet_packet-@30
[708] ENET_0 A_RGMII_PHY_0 PHY: Collision is detected on the bus
[876] ENET_0 A_RGMII_PHY_0 PHY BFM : Finished sending packet #1 UID:
0xc8100001 ETHERNET INJECT vr_enet_packet-@28
[876] ENET_0 A_RGMII_PHY_0 PHY BFM :
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
```

```
    Duplex mode             :HALF
    Packet number           :1
    Length of data          :20
    Start time              :684
    End_time                :876
    IPG in bit time         :96
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xbd55fb832377
    Src. Address            :0x7826a4b48198
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x728911e4
==========Packet Error Information==========
      Error type        : RX_ER
      Error phase       : IPG
      Start time        : 0
      Error length      : 2
      -------------------
============================================
[876] ENET_0 A_RGMII_PHY_0 PHY: SEQ(2) vr_enet_packet-@28 sent by drvr 1
[876] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #0 UID:
0x48000000 ETHERNET INJECT vr_enet_packet-@24
[876] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :0
    Length of data          :20
    Start time              :684
    End_time                :876
    IPG in bit time         :96
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0xee4e3f148e91
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x36e8a1ae
==========Packet Error Information==========
      Error type        : TX_ER
      Error phase       : DATA
      Start time        : 0
      Error length      : 2
      -------------------
============================================
[876] ENET_0 A_RGMII_PHY_0 PHY: SEQ(1) NORMAL_PHY vr_enet_seq-@27 done
[892] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The Source address
could
not be unpacked due to lack of bits
[892] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The Length/Type
field
could not be unpacked due to lack of bits
[892] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: WARNING! : The CRC could not be
unpacked due to lack of bits
[892] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Finished collecting packet #1
CRC:
0xc49b92d9 ETHERNET COLLECT vr_enet_packet-@29
[892] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR:
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :1
```

```
    Length of data         :0
    Calculated CRC         :0x9d64c953
    Start time             :700
    End_time               :892
    IPG in bit time        :0
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0x000000000000
    Src. Address           :0x000000000000
    Length/Type value      :0
    Pad Size               :0
    Actual crc             :0x0
==========Packet Error Information==========
      Error type        : CRC_ERROR
      -------------------
      Error type        : SHORT_FRAME_ERROR
      -------------------
      Error type        : COLLISION
      Error phase       : ABSOLUTE
      Start time        : 0
      -------------------
============================================
[892] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The Source address
could
not be unpacked due to lack of bits
[892] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The Length/Type
field
could not be unpacked due to lack of bits
[892] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: WARNING! : The CRC could not be
unpacked due to lack of bits
[892] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #1
CRC:
0xc49b92d9 ETHERNET COLLECT vr_enet_packet-@30
[892] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :HALF
    Packet number          :1
    Length of data         :0
    Calculated CRC         :0x9d64c953
    Start time             :700
    End_time               :892
    IPG in bit time        :0
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0x000000000000
    Src. Address           :0x000000000000
    Length/Type value      :0
    Pad Size               :0
    Actual crc             :0x0
==========Packet Error Information==========
      Error type        : CRC_ERROR
      -------------------
      Error type        : SHORT_FRAME_ERROR
      -------------------
      Error type        : COLLISION
      Error phase       : ABSOLUTE
      Start time        : 0
      -------------------
============================================
[892] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_chk_seq_err_get is
updated
to the value : 2
```

```
-------------------------------------------------------

   *** Dut warning at time 892
        Checked at line 41 in @vr_enet_data_check
        In vr_enet_monitor-@15.check_tx_packet() (unit:
sys.vr_enet_rgmii_env.active_phy_agents[0].monitor):

ERR_ENET001_MAC_BAD_CRC:
MAC has transmitted a packet with BAD CRC.
Specs(IEEE-802.3,2000): 3.4

-------------------------------------------------------
Will continue execution (check effect is WARNING)



-------------------------------------------------------

   *** Dut warning at time 892
        Checked at line 46 in @vr_enet_data_check
        In vr_enet_monitor-@15.check_tx_packet() (unit:
sys.vr_enet_rgmii_env.active_phy_agents[0].monitor):

ERR_ENET002_MAC_SHORT_FRAME:
MAC has transmitted a packet with SHORT FRAME.
Specs(IEEE-802.3,2000): 4.2.3.3

-------------------------------------------------------
Will continue execution (check effect is WARNING)

[2924] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #0 ETHERNET
INJECT vr_enet_packet-@24
[2940] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #2
ETHERNET COLLECT vr_enet_packet-@31
[3276] ENET_0 A_RGMII_MAC_0 MAC BFM : Injecting TX_ER
vr_enet_directed_prot_err
[4076] ENET_0 A_RGMII_PHY_0 PHY: SEQ(0) MAIN vr_enet_seq-@20 ended
[4076] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #0 UID:
0x48000000 ETHERNET INJECT vr_enet_packet-@24
[4076] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type            :ETHERNET_802_3
    Tag kind               :UNTAGGED
    Duplex mode            :HALF
    Packet number          :0
    Length of data         :20
    Start time             :2924
    End_time               :4076
    IPG in bit time        :1016
    Preamble size in bits  :56
    SFD                    :1 0 1 0 1 0 1 1
    Dest. Address          :0xffffffffffff
    Src. Address           :0xee4e3f148e91
    Length/Type value      :20
    Pad Size               :26
    Actual crc             :0x36e8a1ae
==========Packet Error Information==========
      Error type        : TX_ER
      Error phase       : DATA
      Start time        : 0
      Error length      : 2
      -------------------
============================================
```

```
[4076] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_tx_ok_get is updated
to
the value : 1
[4076] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  brd_frm_tx_ok_get is
updated
to the value : 1
[4076] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  multi_frm_tx_ok_get is
updated
to the value : 1
[4076] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  multi_coll_frm_get is
updated
to the value : 1
[4076] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@24 sent by drvr 0
[4076] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@22 done
[4076] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@32 created
[4076] ENET_0 A_RGMII_MAC_0 MAC: 0 error validation rules violated...
[4076] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@33 created
[4092] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #2
UID:
0x48000000 ETHERNET COLLECT vr_enet_packet-@31
[4092] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :2
    Length of data          :20
    Start time              :2940
    End_time                :4092
    IPG in bit time         :1024
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0xffffffffffff
    Src. Address            :0xee4e3f148e91
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x36e8a1ae
==========Packet Error Information==========
      Error type         : TX_ER
      Error phase        : DATA
      Start time         : 0
      Error length       : 2
      -------------------
============================================
[4260] ENET_0 A_RGMII_MAC_0 MAC BFM : Started sending packet #1 ETHERNET
INJECT vr_enet_packet-@33


-------------------------------------------------------

   *** Dut warning at time 4276
        Checked at line 40 in @vr_enet_mii_checker
        In vr_enet_monitor-@15 (unit:
sys.vr_enet_rgmii_env.active_phy_agents[0].monitor):

mii_ipg_chk_btn_tx_pkts: ERR_ENET017_MAC_MII_IPG_TOO_SHORT:
Inter packet gap (IPG) between two Tx packets is shorter than 96 bit time.
Specs(IEEE-802.3,2000): 4.2.3.2.2 & 22.2.3.1

-------------------------------------------------------
Will continue execution (check effect is WARNING)

[4276] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Started collecting packet #3
ETHERNET COLLECT vr_enet_packet-@34
```

```
[4612] ENET_0 A_RGMII_MAC_0 MAC BFM : Injecting TX_ER
vr_enet_directed_prot_err
[5412] ENET_0 A_RGMII_MAC_0 MAC BFM : Finished sending packet #1 UID:
0x48000001 ETHERNET INJECT vr_enet_packet-@33
[5412] ENET_0 A_RGMII_MAC_0 MAC BFM :
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :1
    Length of data          :20
    Start time              :4260
    End_time                :5412
    IPG in bit time         :96
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0x18957307c120
    Src. Address            :0xac2b44ac6769
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x8755daef
==========Packet Error Information==========
      Error type         : TX_ER
      Error phase        : DATA
      Start time         : 0
      Error length       : 2
      -------------------
============================================
[5412] ENET_0 A_RGMII_MAC_0 MAC: MAC attribute  frm_tx_ok_get is updated
to
the value : 2
[5412] ENET_0 A_RGMII_MAC_0 MAC: SEQ(2) vr_enet_packet-@33 sent by drvr 0
[5412] ENET_0 A_RGMII_MAC_0 MAC: SEQ(1) NORMAL_MAC vr_enet_seq-@32 done
[5428] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Finished collecting packet #3
UID:
0x48000001 ETHERNET COLLECT vr_enet_packet-@34
[5428] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR:
==========Packet Information==============
    Packet type             :ETHERNET_802_3
    Tag kind                :UNTAGGED
    Duplex mode             :HALF
    Packet number           :3
    Length of data          :20
    Start time              :4276
    End_time                :5428
    IPG in bit time         :92
    Preamble size in bits   :56
    SFD                     :1 0 1 0 1 0 1 1
    Dest. Address           :0x18957307c120
    Src. Address            :0xac2b44ac6769
    Length/Type value       :20
    Pad Size                :26
    Actual crc              :0x8755daef
==========Packet Error Information==========
      Error type         : TX_ER
      Error phase        : DATA
      Start time         : 0
      Error length       : 2
      -------------------
============================================
[8612] ENET_0 A_RGMII_MAC_0 MAC: SEQ(0) MAIN vr_enet_seq-@19 ended
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test ...
[8612] ENET_0 A_RGMII_MAC_0 MAC SCOREBOARD:
```

```
Total number of errors are 0

++++++++++++++++++++++++++++++++++++++++++++++++++


[8612] ENET_0 A_RGMII_PHY_0 PHY SCOREBOARD:

Total number of errors are 0

++++++++++++++++++++++++++++++++++++++++++++++++++


Checking is complete - 0 DUT errors, 9 DUT warnings.
****************************************************************
Finished an Ethernet test with stop condition
OBJ_MECH_IDLE_CYCLES_TOGETHER
****************************************************************
eVC name:                    ENET_0
Test time:                   8612
****************************************************************
[8612] ENET_0 A_RGMII_MAC_0 MAC RX MONITOR: Number of packets collected
----------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
-----------------------------------------------------------------------
[8612] ENET_0 A_RGMII_MAC_0 MAC BFM : Number of packets injected
-----------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
-----------------------------------------------------------------------
[8612] ENET_0 A_RGMII_PHY_0 PHY TX MONITOR: Number of packets collected
----------------------------------------------------------------------
4==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
-----------------------------------------------------------------------
[8612] ENET_0 A_RGMII_PHY_0 PHY BFM : Number of packets injected
-----------------------------------------------------------------------
2==> ETHERNET_802_3 type packets
0==> ETHERNET_PAUSE type packets
0==> ETHERNET_VII type packets
0==> ETHERNET_MAGIC type packets
0==> ETHERNET_JUMBO type packets
0==> ETHERNET_SNAP type packets
0==> RANDOM_DATA type packets
-----------------------------------------------------------------------
Wrote 1 cover_struct to test_rgmii_normal_1110103410.ecov
```

```
All sequence drivers:
----------------------
       driver               sent       pending        current
  0.  vr_enet_driver-@1      2           0              -
  1.  vr_enet_driver-@2      2           0              -


**** Specman - finishing session:
config run -exit_on == normal_stop (or all); exiting...
```

--