10GB MAC CORE Verification Plan

Advanced Verification With SystemVerilog OOP Testbench

UNIVERSITY OF CALIFORNIA – SANTA CRUZ

Fall 2014

Authored by: German Andres Mancera

This page intentionally left blank.

SCOPE

The scope of this project is to build a SystemVerilog testbench by following the Objected Oriented Programming (OOP) paradigm. Under the OOP paradigm, the testbench is comprised of different objects which are designed to handle one particular task of the verification flow. These objects and their role will be explained in detailed in this document. The design under test is a 10GE MAC Core whose source code is available online under the LGPL license from OpenCores.org. Even though the design is distributed with a rudimentary test case that allows to run some basic sanity check, such testbench would be extremely difficult to scale and/or maintain. Hence, this project aims to provide a clean verification environment that is easy to understand, maintain and extend by following a complete separation between the RTL domain and the verification domain. This approach has proven to be very helpful while dealing with the verification of complex digital systems such as the ones that are found today in any application.

In the end, the overall idea is to transform a very rudimentary testcase into a fully-fledged SystemVerilog OOP testbench by using all the techniques we have learned throughout this course.

INTRODUCTION

This document starts with an explanation of the 10GE MAC Core design architecture, its main functionality, components, interfaces and register set. The DUT is intentionally designed so that it can be easily integrated with custom logic. It is also designed with a limited feature set to guarantee a small gate footprint. Even though the design features both transmit and receive XGMII (10 Gigabit Media Independent Interface) interfaces, this verification plan will not make any attempt to verify the correctness of these interfaces. The DUT will be connected in loopback mode, i.e., the *xgmii_rx_** interfaces will be directly connected to the corresponding *xgmii_tx_** interfaces.

This document will then move to a detailed explanation of the SystemVerilog OOP testbench. The document will delve into the specifics of each one of the testbench building blocks and will provide details on how they fit into the verification flow. A top level testbench diagram illustrates the existing relationships among these components and also shows the mechanism that is used in order to translate testbench stimulus into RTL signals.

The next section describes the different test cases that have been created and how constrained randomization is used in order to generate different types of stimuli that are meant to exercise different logic inside the DUT.

The next section describes some of the enhancements that are planned for the verification environment. The last section lists all the references and bibliography used throughout this project.

DESIGN ARCHITECTURE

The design under test is a 10GE MAC Core that is designed so that it can be easily integrated with proprietary custom logic. The design features a POS-L3 like interface for the datapath and a Wishbone compliant interface for configuration. A block diagram of the DUT is shown below:

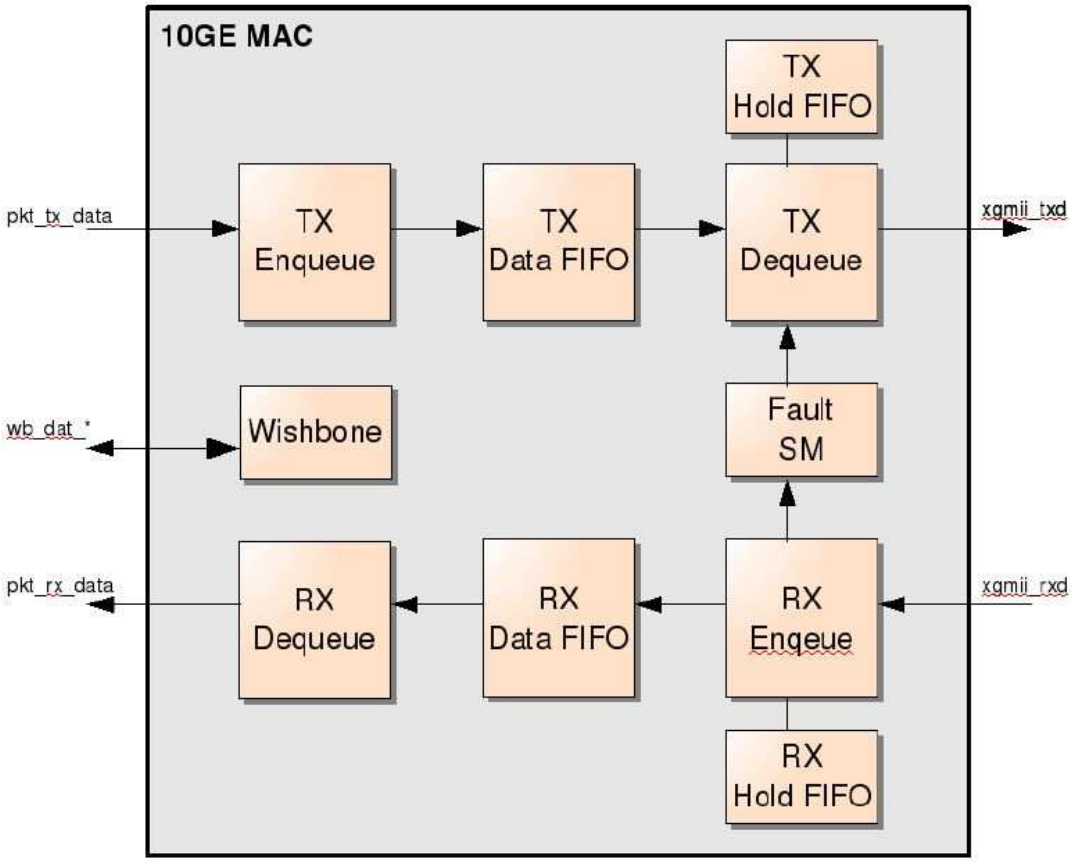

FIGURE 1. DUT BLOCK DIAGRAM

The design has the following interfaces:

- Packet Receive and Transmit interfaces.
- XGMII Receive and Transmit interfaces.
- Management interface.

The DUT receives frames from the user's core logic via the packet transmit interface and forwards data and control characters towards the XAUI/PCS macro. The different signals available in the packet transmit interface provide all the information that is required in order to delimit the packets and get all their data bytes. Table 1 summarizes all the signals available in the interface.

Port Name	Description
pkt_tx_data[63:0]	Transmit data
pkt_tx_val	Transmit valid
pkt_tx_sop	Transmit Start of Packet
pkt_rx_eop	Transmit End of Packet
pkt_tx_mod	Transmit packet length modulus

TABLE 1. PACKET TRANSMIT INTERFACE

The DUT accepts data and control characters from the XAUI/PCS macro and delivers it back to the core logic via the packet receive interface. Just like in the case of the transmit interface, the signals available in the packet receive interface are used to appropriately delimit the packets. Table 2 below summarizes all the signals available in the interface.

Port Name	Description
pkt_rx_ren	Receive read enable
pkt_rx_avail	Receive available
pkt_rx_data[63:0]	Receive data
pkt_rx_val	Receive valid
pkt_rx_sop	Receive Start of Packet
pkt_rx_eop	Receive End of Packet
pkt_rx_mod	Receive packet length modulus

TABLE 2. PACKET RECEIVE INTERFACE

There are also some configuration registers that can be accessed via a management interface. This management interface is Wishbone-compliant and it allows to read and write the following registers:

- Configuration register 0.
- Interrupt Pending Register.
- Interrupt Status Register.
- Interrupt Mask Register.

Table 3 below shows all the different signals that are available in the Wishbone management interface.

Port Name	Description
wb_adr_i[7:0]	Address input
wb_cyc_i	Wishbone cycle
wb_dat_i[31:0]	Wishbone data input
wb_stb_i	Wishbone strobe
wb_we_i	Wishbone write enable
wb_ack_o	Wishbone acknowledgment
wb_dat_o	Wishbone data output
wb_int_p	Wishbone interrupt signal

TABLE 3. MANAGEMENT INTERFACE

VERIFICATION ENVIRONMENT

Verification Methodology

The testbench will be developed in SystemVerilog, and several Object Oriented Programming concepts will be utilized in order to code a robust, yet conceptually simple-to-understand verification environment. Under this methodology, there are different testbench components which specialize in carrying out different tasks. These components are instantiated classes which might or might not be a container class for another component. Other important features of SystemVerilog that are used in the testbench are also explained in the following sections.

Program Block

Each one of the testcases is basically a program block. A program block is a feature of SystemVerilog which is used in order to minimize the possibility of a race condition between the design and the testbench by scheduling events so that they happen in different regions of a delta cycle. Namely, RTL events are scheduled to be executed in the active region whereas testbench events are scheduled to be executed in the reactive region.

Clocking Block

Clocking blocks are another feature of SystemVerilog which aids in preventing race conditions. It specifies the input skew and the output skew. In addition to that, clocking blocks are tied to a sampling event, which is generally the design clock. This design uses 4 different clock signals: *clk_156m25*, *clk_xgmii_rx*, *clk_xgmii_tx* and *wb_clk_i*. These free-running blocks are generated in the testbench module and passed onto the interface.

Virtual Interface

Virtual interface is a pointer to a real interface. This obscure entity stems from the fact that RTL input and output ports can only talk in the form of “wiggles”, whereas testbench-generated transactions are crafted at a much higher level of abstraction: Testbench transactions are basically objects of a class. Therefore, we need a way to let the RTL domain talk to the testbench domain. The vehicle that allows for this communication to happen is the virtual interface. The virtual interfaces are shown as yellow diamonds in Figure 1 later in this section.

Classes

A class is a concept that comes directly from Object Oriented Programming. Most of the testbench components are actually classes. Some of these objects are used for the stimulus generation (packet) and some other objects are used for the infrastructure (environment, driver, monitor, scoreboard, and coverage). When it comes to stimulus generation, constrained randomization plays an important role in allowing the testbench to generate stimuli that will trigger different conditions inside the DUT. When it comes to infrastructure, container classes are used in all different places in the testbench. Some objects are contained inside other object as shown in the figure below.

Testbench Components

The following figure shows the architecture of the testbench that will be used in order to validate the correctness of the Design under Test (DUT). A description of each one of the testbench components is provided below.

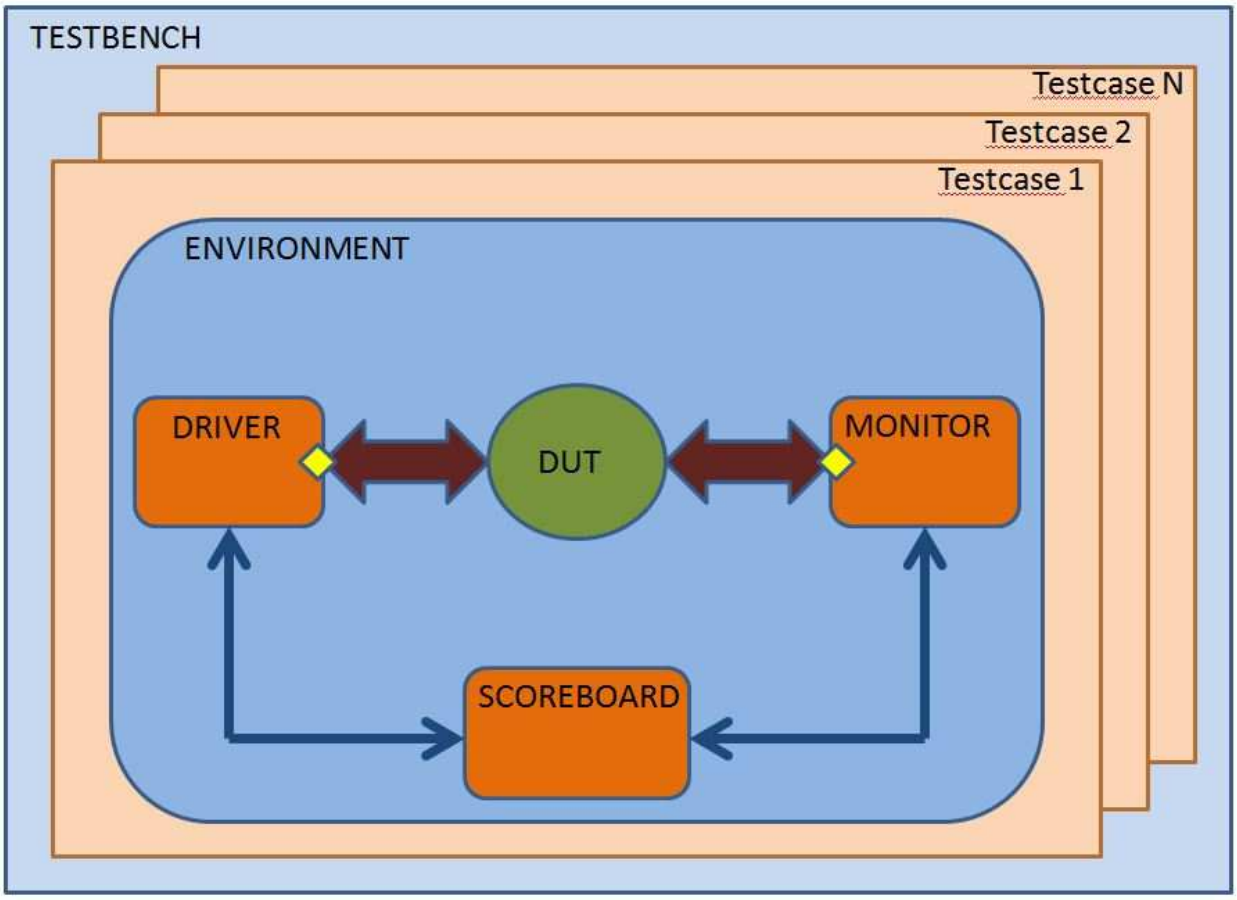

FIGURE 2. TESTBENCH ARCHITECTURE

- ***Driver:*** The driver class is used as one of the infrastructure pieces in the testbench. The driver is aware of the interface protocol and is in charge of taking the packet data items and driving them into the RTL input ports. Part of the interface protocol awareness is being able to correctly drive each one of the signals listed on Table 1. The driver needs to be able to talk to the design through its interface. This communication is only made possible by the virtual interface.

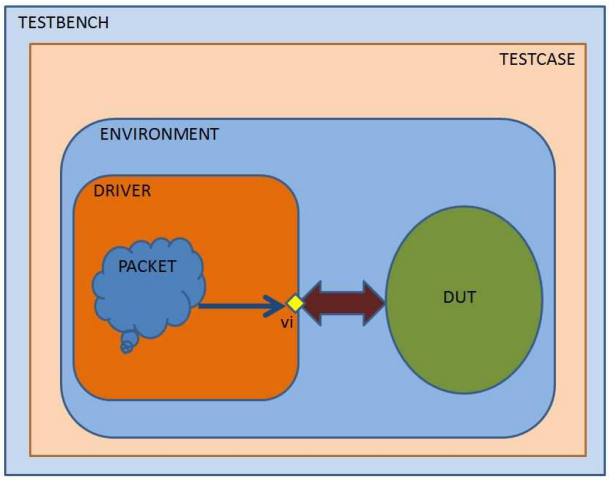

FIGURE 3. COMMUNICATION BETWEEN THE DRIVER AND THE DUT

- ***Monitor:*** The monitor class is also used as one of the infrastructure pieces in the testbench. This class is similar to the driver class in the sense that is also aware of the interface protocol and is able to monitor activity on the output ports of the DUT. While checking for any activity in the interface, the monitor captures packets based on the signals listed on Table 2. The monitor class uses a virtual interface in order to be able to communicate with the design through its output interface.

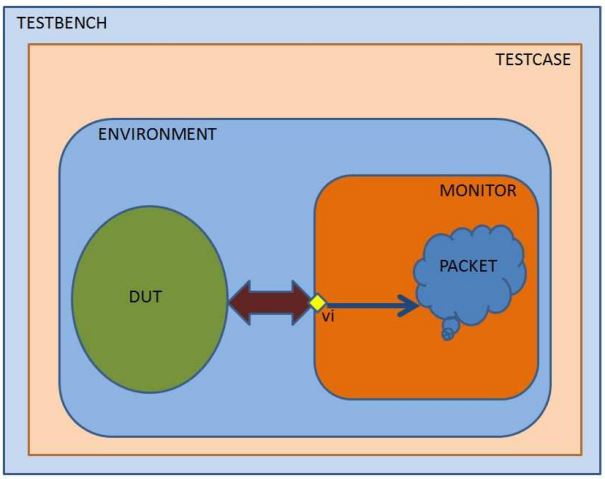

FIGURE 4. COMMUNICATION BETWEEN THE MONITOR AND THE DUT

- ***Scoreboard.***
  This testbench component receives transaction from both the driver and the monitor via mailboxes. The expected transaction is placed in a mailbox by the driver. Likewise, the actual transaction is placed in another mailbox by the monitor. The scoreboard will then proceed to compare these 2 transactions and flag any mismatch with an appropriate error so that the test case fails.

- ***Environment.***
  The environment class is a container class that instantiates the driver, the monitor, and the scoreboard. It is a top level class that controls some of the tasks that are performed by the other testbench components. The environment class is responsible for calling some tasks inside the driver and the monitor to drive and collect packets respectively.

- ***Testcases.***
  Each testcase is a program block that contains a packet-derived class that either relaxes or tightens the constraints that are built around the base packet class. This further constraining allows for more directed tests cases that verify specific DUT behaviors in the presence of certain types of stimuli. Each one of the testcases that have been developed is explained in the following section.

- ***Packet.***
  The packet class is the transaction or data item that is used by the testbench in order to create randomized stimulus that targets the design. Even though the DUT is not entirely Ethernet compliant, a decision has been made to model some of the class members so that they resemble the Ethernet frame header. This allows for the verification to be put in the context of a real world design. The Ethernet-like frame that has been used as a reference for the packet class creation is shown in Figure 5 below.

FIGURE 5. HIGH-LEVEL PACKET CLASS

TESTCASES

Loopback – Bringup Test

This testcase is used in order to bring up the verification environment and it is similar to the one that came with the along with the design. All the header fields of the packet are heavily constrained (MAC destination address, MAC source address, Ether Type) and the payload also has a predefined pattern. The payload size is also constrained to be between 45B and 54B and the interpacket gap is set to a fixed value. All these constraints are put in place to make sure that all the TB components are working as expected and that there are no bugs in their implementation.

Packet with missing EOP

The interface protocol states that the valid signal should be asserted for the duration of the packet. Furthermore, the EOP signal must be asserted for one clock cycle during the last word of the packet. The transmit packet length modulus signal is valid only on EOP and it indicates how many valid bytes are present in the transmit data. This is an error case and the design is expected to recover gracefully as soon as it sees an SOP from the next packet.

Packet with missing SOP

Similar to the missing EOP case, the interface protocol also states that the SOP signal valid signal must be asserted during the first word of the packet. This is also an error case and the design is expected to recover gracefully as soon as it receives the SOP from the next packet.

Undersized packet

According to the functional specification, the TX dequeue engine will pad small packets to the minimum 64B-size required for Ethernet. This testcase will send packets whose length is smaller than 64B in order to verify that they are treated appropriately and don't get dropped by the design.

Oversized packet

Just like in the case of the undersized packet, the design should be able to handle packets that exceed the maximum packet size defined in the standard. This testcase will send packets whose size is greater than 1518B in order to verify that they are treated appropriately and don't get dropped by the design.

Zero IPG packet

In this testcase, the interpacket gap will be constrained so that the design is hit by back-to-back packets on the transmit interface. The packet size should also be properly constrained here to prevent any FIFO overrun. This condition will typically happen when the interpacket gap is very small and the packet size is also very small.

TESTBENCH ADD-ONS

Coverage

The coverage class is instantiated inside the scoreboard, where some covergroups are created with their corresponding bins. This class is used in order to get functional coverage information, which is a measure of the functional correctness of the design.

Assertions

SystemVerilog assertions were formally included in the IEEE 1800 standard. They allow checking of end-to-end behaviors. Based on the functional specification, some temporal assertions can be added to the interface in order to verify SOP/EOP correct framing on the packet receive interface.

Regression Script

A top level regression script will be provided (possibly leveraged from the one that was provided by the instructor in Lab 2). This script will allow to run a full regression and will also report on the results.

Wishbone interface

The wishbone interface clock and reset signals will be properly wired from the testbench so that this interface is functional. This will allow the testbench to have access to the design registers and make some reads and writes either during initialization time (after reset is deasserted) of at the end of the test.

REFERENCES

- 10GE MAC Core Functional Specification – OpenCores.org
  http://opencores.org/project,xge_mac

- Wishbone System-on-Chip Interconnect Architecture. Revision B.3.
  http://cdn.opencores.org/downloads/wbspec_b3.pdf

- Advanced Verification using SystemVerilog OOP Testbench. Benjamin Ting.
  pp. 1-549. July 11, 2014

- OOP Testbench Workbook. Benjamin Ting
  pp. 1-214. July 11, 2014

APPENDIX: POTENTIAL BUGS IN DESIGN

Potential Bug 1: Under certain conditions, the design is unable to gracefully handle packets with a missing EOP flag.

Under certain conditions and for some packet sizes, the design is not able to gracefully handle packets that are received with the appropriate EOP delimiter on the last word. The packet in error is correctly processed and the *pkt_rx_err* signal is correctly asserted. The transmission of the subsequent packet is also performed correctly. However, there is a transition period in between the error and non-error packets where the design does not drive correctly the packet receive interface signals. This transition period is shown in red in figure 6 below.

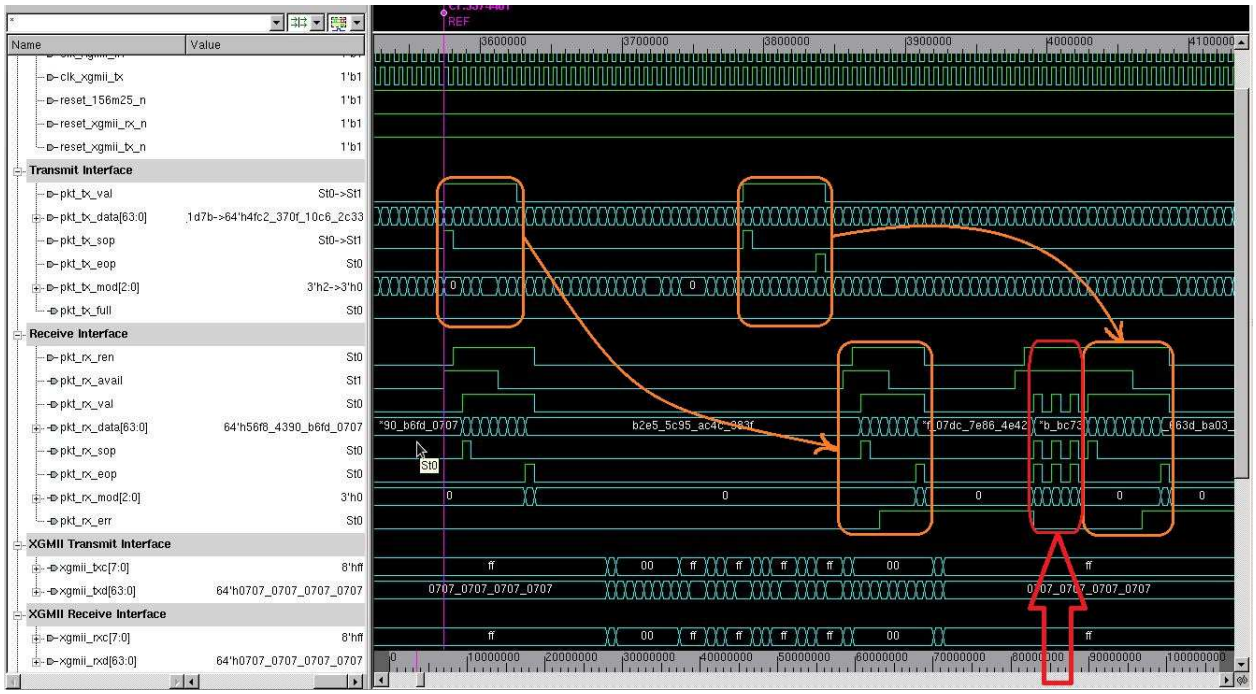

FIGURE 6. DESIGN NOT HANDLING CORRECTLY A PACKET WITH MISSING EOP

As seen in the figure, the design is sending 3 single-cycle packets (i.e., SOP and EOP are asserted in the same clock cycle). The pkt_rx_err signal is never driven back to 1'b1, so the core logic that connects to the packet receive interface would have interpreted this as 3 valid short packets.

Potential Bug 2: The design does not pad certain undersized packets with zeros as mentioned in the Functional Specification.

For certain packet sizes that fall into the undersize packet category, the design is not passing the the packets with zeros. While explaining the functionality of the TX Dequeue Engine, the functional specification says that "the state-machine reads data from the data FIFO and pads small packets to the minimum 64-byte required for Ethernet". Figure 7 shows a case where the design receives a packet of size 53 bytes. The packet is transmitted over 7 clock cycles. The *pkt_tx_mod* signal is driven to 8'h5 on the EOP cycle to indicate that only *pkt_tx_data[63:24]* is valid, hence the last word should be 40'hc915_35ac_ed. The remaining 3 bytes (24'h9a_c599) should be discarded by the design.

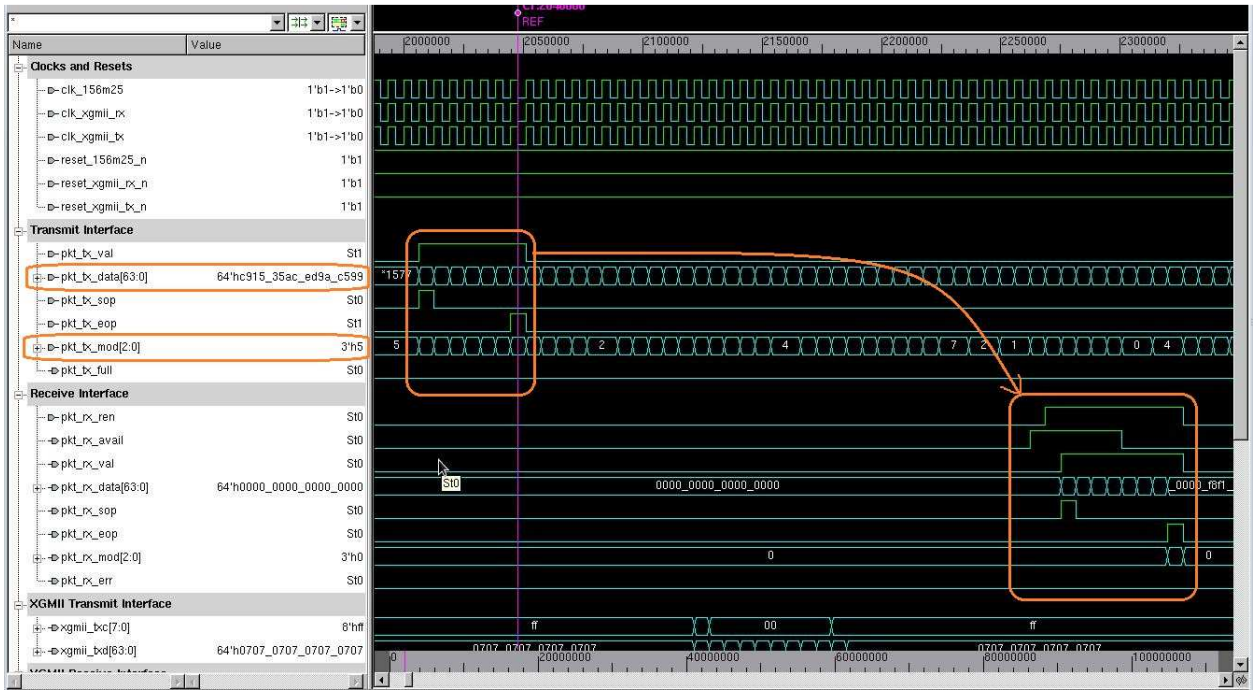

FIGURE 7. UNDERSIZE PACKET RECEIVED BY THE DESIGN

As can be seen in figure 8 below, the transmission of the same packet over the packet receive interface takes 8 cycles. This happens because of the padding that the design is supposed to perform for undersized packets. Notice how the *pkt_rx_data* signal is incorrectly driven on the 7th clock cycle. The design is driving the 7th word of the packet as 64'hc915_35ac_ed9a_c599. The 3 LSBs of this word are incorrect and should have been driven to zero. Hence the 7th word of the packet should have been of the form 64'hc915_35ac_ed00_0000.

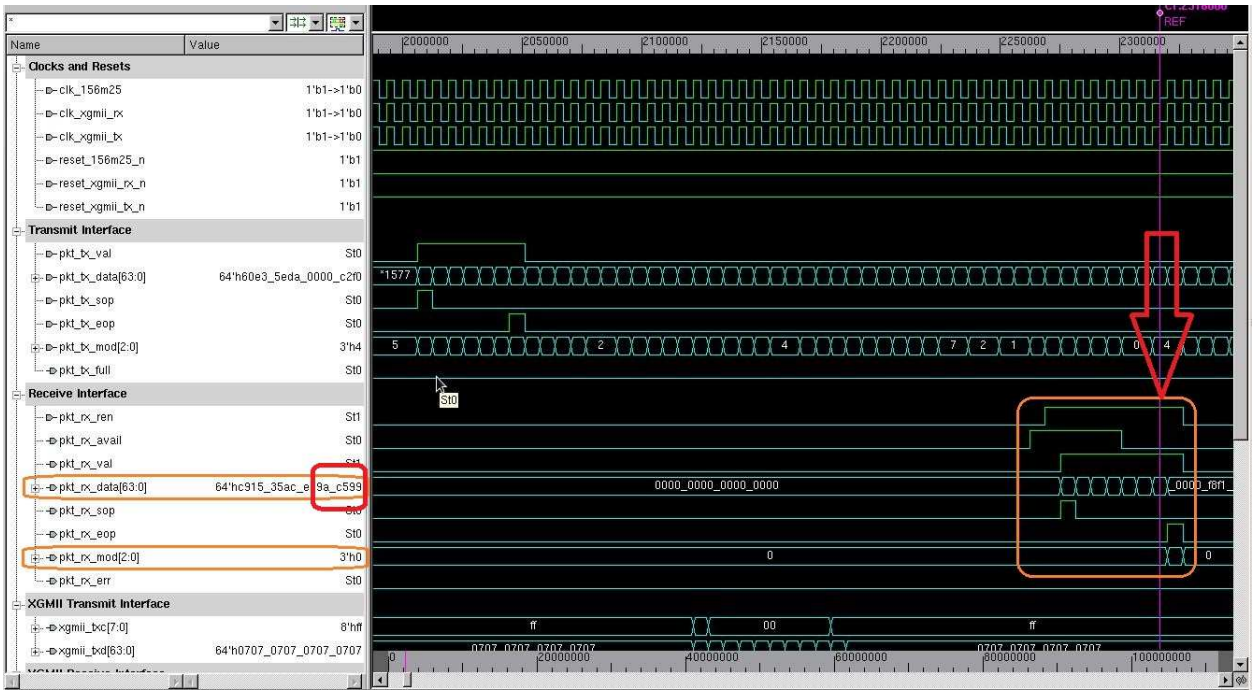

FIGURE 8. UNDERSIZE PACKET TRANSMITTED BY THE DESIGN

The garbage bytes of the 7th word as marked in red in Figure 8 above. Since the design was always configured in loopback mode, it is possible that this bug resides in the upstream logic that drives the XGMII interface. A detailed look at the XGMII transmit interface and its control signals would be required to further triage this issue.