UVM Testbench

10 GE MAC CORE
Verification Plan

Author: Advait Madhekar

Table Of Contents

---

1. Introduction

1.1 Purpose

The purpose of this report is to illustrate the plan and infrastructure behind testing a RTL design for a 10 Gigabit MAC design. The report contains design specifications for the RTL design, testbench, and how individual components will communicate with each other to test the RTL.

1.2 Mission Statement

In order to achieve our goal, we will be setting the RTL design under verification, with an external testbench environment programmed in System Verilog. We will be using the different components that UVM has to offer by inheriting from those components.

In order to comply with UVM standards, all programming will be done considering Object Oriented Programming in mind. This means that all components of the testbench will use a class-based structure.

2. Architecture and High Level View

2.1 Clock Signals

In order to synchronize the DUT with the external testbench environment, it is very important to create all the different clocks that the DUT uses. This section outlines those clock cycles and their frequencies.

Name (Testbench)	Name (DUT)	Frequency	Description
wb_clk_tb	wb_clk_i	30 - 156 Mhz	Wishbone Interface Clock.
clk_156m25_tb	clk_156m25	156.25 Mhz	Clock
clk_xgmii_tx_tb	clk_xgmii_rx	156.25 Mhz	Clock for XGMII receive interface in the DUT. The testbench will transmit packets.
clk_xgmii_rx_tb	clk_xgmii_tx	156.25 Mhz	Clock for XGMII transmit interface in the DUT. The testbench will receive packets.

2.2 Registers in DUT

Inside the DUT, there exists a series of registers with different functionalities. These registers hold configuration data that affects the behavior of the DUT in different circumstances. A list of these registers is provided here, as all of them are relevant to our design in the DUT, either relating to the wishbone agent or the config agent.

Register List

Name	Address	Width	Access
Config Register 0	0x00	32	R/W (READ/WRITE)
Interrupt Pending Register	0x08	32	COR (CLEAR ON READ)
Interrupt Status Register	0x0c	32	RO (READ ONLY)
Interrupt Mask Register	0x10	32	R/W (READ/WRITE)
Transmit Octets Count Register	0x80	32	RO (READ ONLY)
Transmit Packets Count Register	0x84	32	RO (READ ONLY)
Receive Octets Count Register	0x90	32	RO (READ ONLY)
Receive Packets Count Register	0x94	32	RO (READ ONLY)

The functionality of each specific register is taken into account when using config and wishbone sequence items to monitor the functionality and validate that the DUT is active and working properly.

2.3 DUT Architecture

The architecture for our DUT is as given below. The DUT contains configuration registers, two state machines, and multiple interfaces. Our DUT is Wishbone compatible, so we will be using a Wishbone agent (Outlined in 2.4 High Level View) to try and monitor statistics from our DUT.

Our testbench will use loopback for the XGMII interface. This means that we won't actually need XGMII TX or RX agents. Instead, the testbench will just use a simple wire to connect the XGMII RX port to the XGMII TX port. Remember that the XGMII TX in the DUT is equivalent to the XGMII RX in the Testbench environment. **IM2** is made with consideration of the testbench environment in mind.

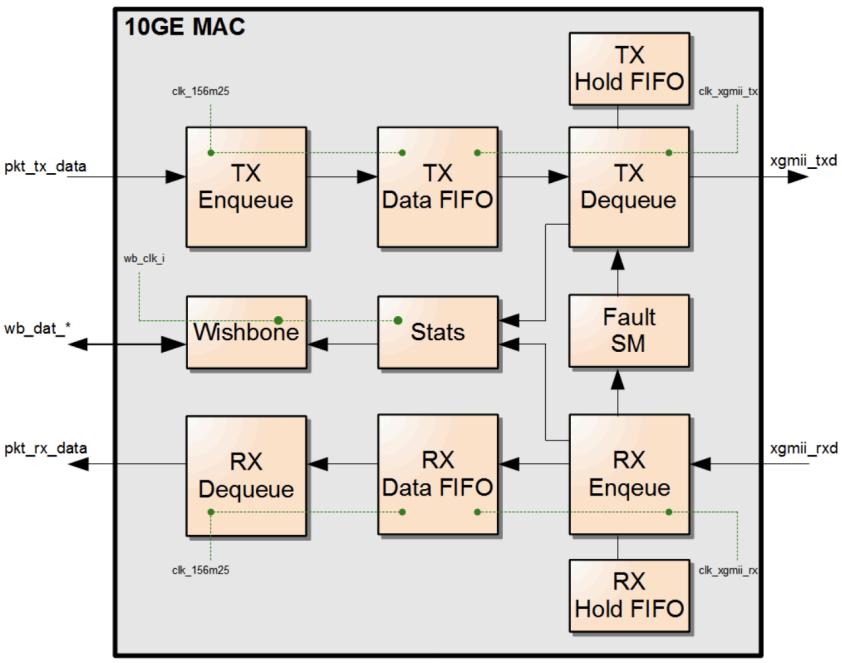
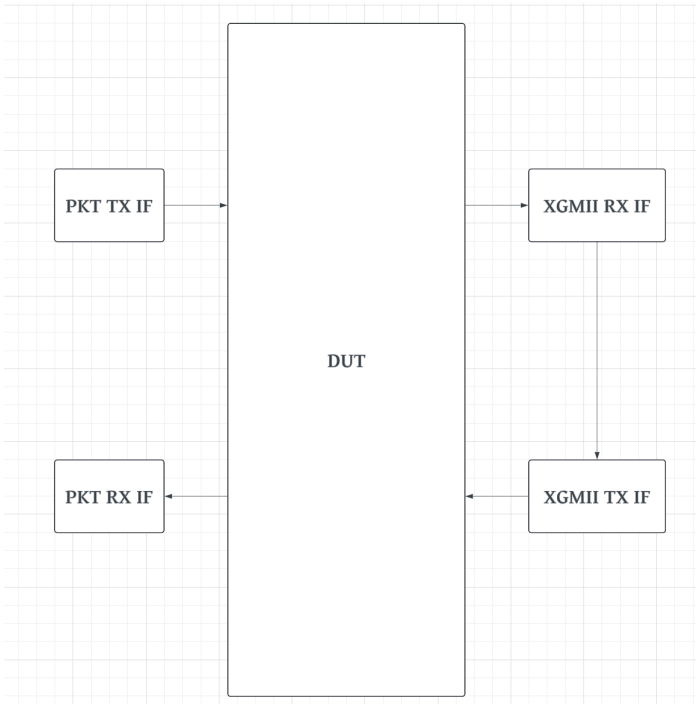


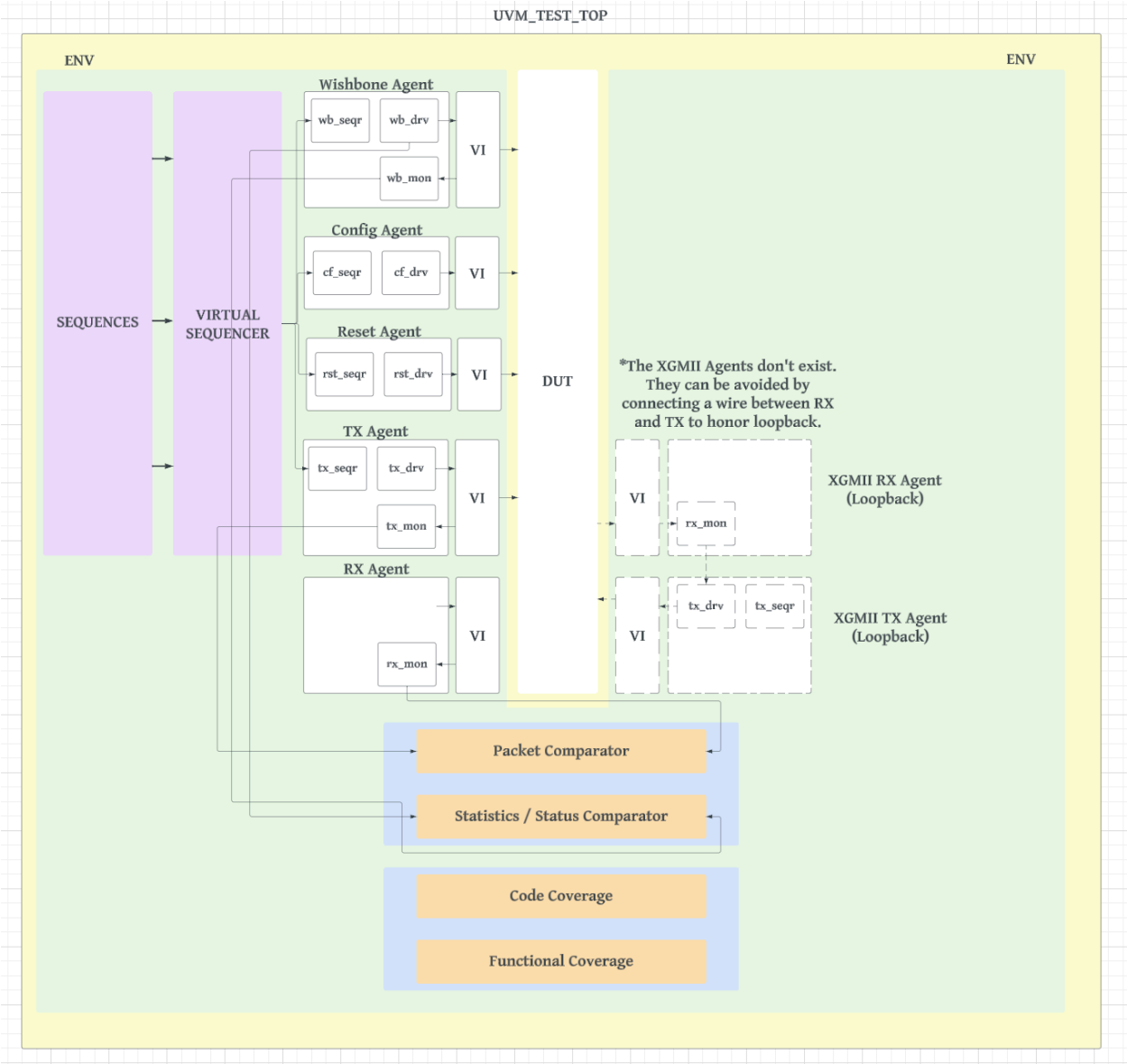
Figure 1: Block Diagram

IM1: 10 Gigabit MAC Core IP Design Specification.



IM2: External Testbench Environment for 10 Gigabit MAC Core IP Design, considering loopback mode.

2.4 High Level View

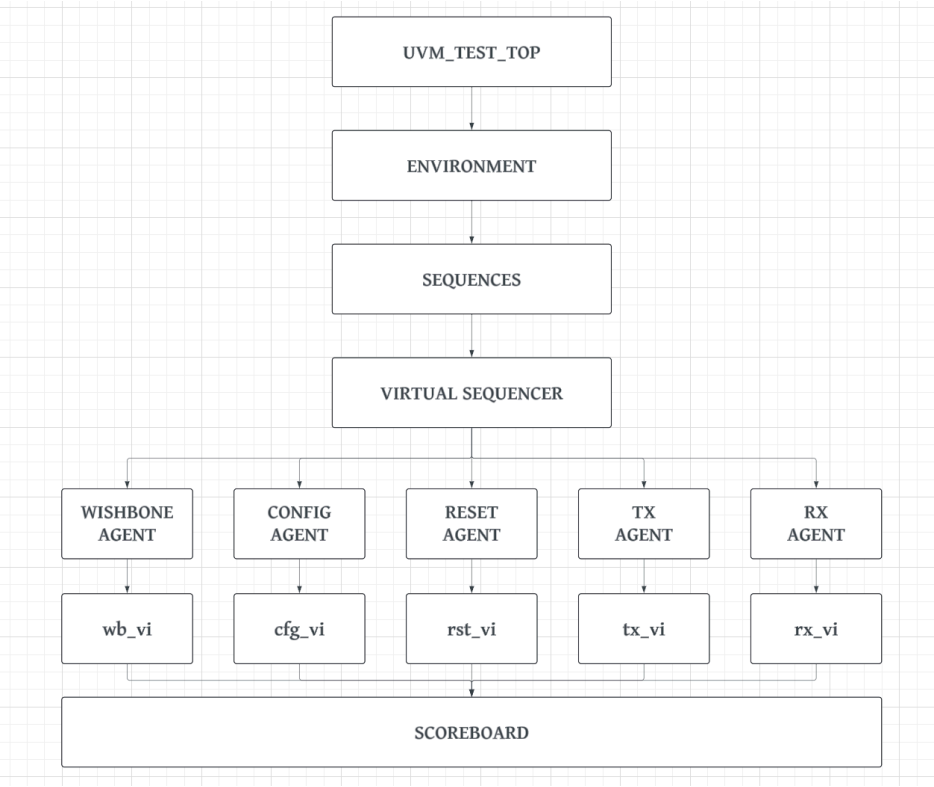


IM3: High Level View of Testbench Architecture and Infrastructure.

Above, we can see a high level view of how our testbench environment will look. All the components given in this view will be built into the testbench.

Quick Note: The XGMII interface is programmed in loopback mode. Therefore, we won't be building the XGMII agents for this testbench. Instead, we will be connecting the receiving XGMII interface to the transmitting XGMII interface with a wire.

2.5 Testbench Topology



IM4: Testbench Topology

Testbench Topology

Environment and Top Testbench Infra:

Name	File Name	Purpose and Description
UVM Test Top	uvm_test_top.sv	Wrap the entire testbench across this class.
Environment	environment.sv	Creating the environment for the testbench.
Virtual Sequencer	virt_seqr.sv	To route sequence items to the appropriate agents.

Agent Infra:

Name	File Name	Purpose and Description
Wishbone Agent	wb_agent.sv	A wishbone wrapper class to wrap the sequencer, driver, and monitor.
Wishbone Sequencer	wb_seqr.sv	Hold onto packet for driver until driver sends API request for item_done() and get_next_item().

Wishbone Driver	wb_drv.sv	Drive the packet through to the Virtual Interface to pin toggle, and then to the DUT.
Wishbone Monitor	wb_mon.sv	Receive Wishbone Packets from Wishbone Interface of DUT, and forward to Scoreboard::Register Comparator.
Config Agent	cfg_agent.sv	A config wrapper class to wrap the sequencer, driver. Can only accept Config Sequence Items. Meant to access Registers in the DUT. Refer to section 2.2 Registers in DUT for more information.
Config Sequencer	cfg_seqr.sv	Hold onto packet for driver until driver sends API request for item_done() and get_next_item().
Config Driver	cfg_drv.sv	Drive the packet through to the Virtual Interface to pin toggle, and then to the DUT.
Reset Agent	rst_agent.sv	A reset wrapper class to wrap the sequencer and driver. Can only accept Reset Sequence Items.
Reset Sequencer	rst_seqr.sv	Hold onto packet for driver until driver sends API request for item_done() and get_next_item().
Reset Driver	rst_drv.sv	Drive the packet through to the Virtual Interface to pin toggle, and then to the DUT.
TX Agent	tx_agent.sv	A TX wrapper class to wrap the sequencer, driver, and monitor. Can only accept TX Sequence Items.
TX Sequencer	tx_seqr.sv	Hold onto packet for driver until driver sends API request for item_done() and get_next_item().
TX Driver	tx_drv.sv	Drive the packet through to the Virtual Interface to pin toggle, and then to the DUT.
TX Monitor	tx_mon.sv	This monitor has two functionalities: 1. Tap into the Virtual Interface to monitor all packets that are entering the DUT, and record them into the Scoreboard. 2. Will check pkt_tx_full output signal from DUT. If true, stop sending additional packets to the Scoreboard. If not, keep sending packets to the Scoreboard. The reason we don't have a connection from Driver to Scoreboard is that we want to make sure that the packets going into the Packet Comparator FIFO in the scoreboard are packets that were successfully inserted into the FIFO of the DUT. In the event they weren't

		inserted properly, we don't want any illegitimate errors.
RX Agent	rx_agent.sv	A RX wrapper class to wrap the monitor.
RX Monitor	rx_mon.sv	Receive RX Packets from Wishbone Interface of DUT, and forward to Scoreboard::Register Comparator.

Virtual Interface Infra:

Name	File Name	Purpose and Description
Wishbone Virtual Interface	wb_virt_int.sv	Pin Toggles Wishbone Sequence Items to signals for DUT, and signals to sequence items for Agent's Monitor.
Config Virtual Interface	cfg_virt_int.sv	Pin Toggles Config Sequence Items to signals for DUT.
Reset Virtual Interface	rst_virt_int.sv	Pin Toggles Reset Sequence Items to signals for DUT.
TX Virtual Interface	tx_virt_int.sv	Pin Toggles TX Sequence Items to signals for DUT, and signals to sequence items for Agent's Monitor.
RX Virtual Interface	rx_virt_int.sv	Converts signals from DUT to sequence items for Agent's Monitor.

Validation and Assertion Infra:

Name	File Name	Purpose and Description
Scoreboard	sb_wrap.sv	Wrapper Class.
Scoreboard::Packet Comparator	sb_pckt_comp.sv	Packet Comparator FIFO in Scoreboard
Scoreboard::Statistics Comparator	sb_stat_comp.sv	Statistics/Status/Configuration Data Comparator FIFO in Scoreboard
Coverage	cov_wrap.sv	Wrapper Class.
Coverage::Code Coverage	cov_code_cov.sv	Code Coverage Monitoring
Coverage::Functional Coverage	cov_funct_cov.sv	Functional Coverage Monitoring

3. Component and Object Descriptions

3.1 Testbench Top View

We will be building a uvm_test_top class that will define the behavior of our test cases throughout the whole simulation. Because we're dealing with multiple agents, we will be implementing a virtual sequencer to take care of all coordination efforts.

1. uvm_test_top derived from uvm_test
    a. Handles all the testbench behavior.
2. environment derived from uvm_env
    a. Handles all environment specifications.
3. uvm_virtual_sequencer derived from uvm_sequencer
    a. Handles all routing of sequence items across multiple agents.

3.2 XGMII Packet Item

Just like any testbench, we will be constructing sequence items that will be used to send to the DUT. This section describes the sequence item for the XGMII interface. The sequence item will contain multiple fields that will be useful to send to the DUT. The fields that each sequence item will collect are represented below.

Name (Testbench)	Name (DUT)	Type	Size (In Bits)	Purpose
transmitted_data	pkt_tx_data	Array	64 Bits	To transmit specific data to the DUT.
transmit_valid	pkt_tx_val	Binary	1 Bit	To identify valid data to the 10GE MAC.
sop	pkt_tx_sop	Binary	1 Bit	To identify the start of the packet.
eop	pkt_tx_eop	Binary	1 Bit	To identify the end of the packet.
data_length_modulus	pkt_tx_mod	Array	3 Bits	To identify the length of packet, during End of Packet. Used to identify valid bytes during the last word of the packet. In the testbench environment, this will allow the monitor to make a decision on what bytes are actually valid, and what bytes aren't.

The signals that we're tracking here are either the inputs or outputs for the DUT. Our goal is to track the right signals that can be used by the Agent's driver and monitor, and the scoreboard.

3.3 XGMII Packet Sequence

The packet sequence is responsible for the generation of our packets. The sequence's duty is to generate the sequence items using randomized data. The class that will be implemented for the packet sequence will be called packet_sequence, and will be derived from the packet class.

The packet_sequence class will pull a value "num_packets" from the uvm_config_db, using the uvm_config_db::get() function. This value will help it determine how many sequence items to generate. It will then generate the required packets.

Setting the value:

```
static int num_packets = 10;
uvm_config_db#(int)::set(this, "*", "num_packets", num_packets);
```

Getting the value:

```
static int num_packets;

virtual task body();
  uvm_config_db#(int)::get(this, "*", "num_packets", num_packets);
endtask
```

This code doesn't show the actual process for using the num_packets, but will set num_packets equal to what the test case has intended.

3.4 Wishbone Item

The wishbone interface in the DUT is meant to allow us to access the statistics registers in the DUT. This will give us different information about the behavior of the DUT during the whole simulation.

The wishbone item allows us to create a base layout for the wishbone packet. The description for this sequence item is as follows.

Name (Testbench)	Name (DUT)	Type	Size (In Bits)	Purpose
wishbone_addr	wb_adr_i	Array	8 Bits	Address Input. Bits 0 and 1 are not used. All accesses to the core must be 32 bits long.
wishbone_cycle	wb_cyc_i	Binary	1 Bit	Wishbone Cycle
wishbone_data	wb_dat_i	Binary	32 Bit	Wishbone Data to be Inputted into Wishbone FIFO.
wishbone_strobe	wb_stb_i	Binary	1 Bit	Wishbone Strobe.

wishbone_write	wb_we_i	Array	3 Bits	Wishbone Write Enable.

3.5 Wishbone Sequence

The wishbone sequence's task is to generate wishbone sequence items given to the members as described in section 3.4. The wishbone_sequence class will inherit from uvm_sequence. Within the sequence class, there will be multiple tasks to define the pre_body(), body(), and post_body() behavior of the class object in the simulation environment. The actual wishbone sequence item member values will be completely randomized.

The goal is to be able to make read and write wishbone sequence items.

3.6 Config Item

The configuration sequence item is used to configure the DUT. You can find a list of registers in 2.2 Registers in DUT.

The point of this item is to configure the DUT based on specific rules within the config registers from before, and monitor the behavior of the DUT over time.

3.7 Config Sequence

The config sequence will be incharge of creating the sequence items, and sending them over to the sequencer in the Config Agent.

3.8 Reset Item

The purpose of having reset functionality is to reset all the pins and the fifo within the DUT. Specifically, we'll be using the reset pins within the DUT. In order to make sure that everything is reset properly, we'll need to assert/deassert the different pins within 2-cycles of each other. Therefore, the descriptions below for **Type** are asserted/deasserted at different times, within different sequence items, at different cycles.

Name (Testbench)	Name (DUT)	Type	Description
wishbone_rst	wb_rst_i	UVM_ACTIVE_LOW	This is Active High. It must be deasserted synchronous to wb_clk_tb to reset the wishbone interface.
xgmii_tx_rst	reset_xgmii_tx_n	UVM_ACTIVE_HIGH	This is Active Low. It must be asserted synchronous to clk_xgmii_tx_tb to reset the xgmii tx sequence.
xgmii_rx_rst	reset_xgmii_rx_n	UVM_ACTIVE_HIGH	This is Active Low. It must be asserted synchronous to clk_xgmii_rx_tb to reset the xgmii rx sequence.
156m25_rst	reset_156m25_n	UVM_ACTIVE_HIGH	This is Active Low. It must be asserted synchronous to clk_156m25_tb to reset the clock

			domain core functionality.

3.9 Reset Sequence

Using this sequence allows the testbench to reset both the receive and transmit FIFOs within the DUT. The reset_sequence class will inherit from the reset sequence item class. To ensure that the reset sequence item that is driven to the DUT has done its job, it's important to assert the signals within 2 cycles of each other. This is a requirement for the DUT.

4. Agent Descriptions

4.1 TX Agent

The TX Agent is a wrapper class for the different components that are built within it. These components include a sequencer, driver, monitor, virtual interface, and a TLM analysis port. It is responsible for taking packets from the XGMII Packet Sequence and sending them to both the scoreboard and the DUT.

The sequencer within the TX Agent will request a packet from the sequence, and will hold onto the packet until the driver requests it. Once the driver calls the two API calls (get_next_item() and item_done()), the sequencer will keep sending the sequence items it receives from the sequence to the driver. The sequence will generate a new item on the fly. The driver is responsible for sending the sequence item to both the scoreboard and the DUT.

Before sending the sequence item through to the DUT, the driver needs to pin toggle to sequence item. It will do this by sending the sequence item through a virtual interface, which will allow the DUT (made in modules) to read the signals in the sequence item.

The TX Agent also contains a monitor. The monitor has two functionalities. It needs to tap into the virtual interface and send any packets from there to the Scoreboard. It also needs to monitor the signal "pkt_tx_full" from the DUT, and ensure that if this signal ever goes high, to not send any more packets to the Scoreboard.

4.2 RX Agent

Unlike the TX Agent, the RX Agent does not contain a driver and a sequencer. It only contains a monitor and a TLM analysis port. Due to this, the RX Agent is categorized as a passive agent.

The RX Agent's job is to receive any signals from the DUT at every clock posedge, convert them to object format using a virtual interface, and check the object for idle data. If the object does contain idle data, the object will be thrown out. Otherwise, it will be sent to the scoreboard and coverage for validation.

The RX Agent will deem a packet invalid by taking a look at xgmii_txc[7:0] signal. If any bit within this signal is low, the packet will be deemed valid. If not, it is an invalid packet.

xgmii_txc[7:0] is a control signal outputted from the DUT. It shows if any bytes in the xgmii_txd[63:0] are valid, or idle. The location of the bit in xgmii_txc directly correlates to the byte within xgmii_txd, and can be used to determine the validity of the object.

4.3 Wishbone Agent

The wishbone agent is an active agent, containing a sequencer, driver, monitor, pass through port, and a virtual interface. The wishbone agent will work just like the RX and TX agents, and will request a wishbone_item from the wishbone_sequence. Once it has the item, it will wait till the driver is done with its current item, and then send it to the driver. The driver will send the item to the scoreboard, coverage, and will toggle pin the item to make it suitable for the DUT. The monitor in the wishbone agent will enable tracking and validation of received wishbone packets.

4.4 Config Agent

The config agent is a passive agent, which contains a sequencer, driver, and a virtual interface. This agent is responsible for config sequence items. It's job is to edit the configuration registers or read them depending on the access of each configuration register.

4.5 Reset Agent

The reset agent will contain a sequencer and a driver. The sequencer will request reset_items from the reset_sequence, and will hold onto the item for the driver. The driver can request the next item with the built in API call infrastructure. This enables the sequencer to send the item it's holding on to the driver, and request the next sequence_item using the seq.start() function. All of this infrastructure will be built into the testbench.

5. Validators Description

5.1 Scoreboard (Assertions)

The scoreboard is responsible for comparing packets that are transmitted to and received from the DUT. The driver in each agent that uses the scoreboard is responsible for sending their own sequence_item data to the scoreboard. The scoreboard will hold that data in a queue. Once a packet has been received by the monitor, the corresponding transmitted packet in the scoreboard will be popped from the FIFO, and will be checked against the data in the received packet. If the data matches, there should be no problems.

The scoreboard is also responsible for comparing the registers in the DUT to how they should be. For example, if only one packet was transmitted, then we should be able to tell that from our own scoreboard.

5.2 Code Coverage

Code coverage is important to identify untested and under-tested parts of our DUT. Checking code coverage allows DV engineers to identify points in the DUT that haven't been covered, and could contain bugs and errors.

5.2 Functional Coverage

Functional coverage allows us to measure what functionalities or features have been exercised by the testcases. In our case, this is useful as we need to measure the effectiveness of our test cases in a constrained random verification environment.

6. Test Case Descriptions

Test General Types

Test Type ID	Test Type Name	Directory	Type Description
1	Loopback	loopback	Loopback Test Cases to ensure packets being transmitted from the TX interface are being received from the RX interface.
2	Directed	directed	Test cases meant to test specific protocols within the system.
3	Constraint Random	random	Randomized Test Cases.
4	Configuration	config	Test cases meant to test behavior DUT based on different configuration settings. Also meant to monitor different configuration settings and statistics based on packets transmitted.
5	Flow Control	flow_control	Monitor DUT's behavior during various flow control mechanisms.

Loopback Tests

Test ID	Test Name	Test Description + Completion
1	loopback_basic	Packets transmitted through the TX agent and interface. Packets should be successfully received on the RX agent and interface. Scoreboard FIFO should be empty at the end of simulation.

Directed Tests

Test ID	Test Name	Test Description + Completion
1	directed_pkt_len_walk	Packet length walking Send all length packets (between 64 bytes to 1518 bytes) to DUT. All packets sent in incremental length.
2	directed_alt_big_small	Alternating big/small packets. Can specify ranges for big or small packet length types.
3	directed_big	All Big Packets.
4	directed_small	All Small Packets.
5	directed_rx_pkt_zero	Try to read from DUT's FIFO when read_enable is on, rx_pkt_avail = 0. Should give invalid data. *Rx_data_underflow may get set based on the Interrupt Pending Register.
6	directed_rx_pkt_err	Corrupt a single byte within the Loopback XGMII. May need to corrupt a single byte on the XGMII Loopback between TX and RX. *Not possible to achieve this test/signal high if this cannot be done.
7	directed_drain_fifo	Try to read from DUT's FIFO when read_enable is on, but FIFO is full. Keep reading from FIFO at every clock posedge till invalid data. Ensure that data is the same as what was inserted by the TX agent.
8	directed_rx_fragment_err	Try to receive a < 64 byte packet with CRC error. *Not possible due to lack of availability of information pertaining to the TX CRC enable bit in the config register.
9	directed_disable_tx	Switch off TX Enable bit in Configuration Register 0 and try to send packets from the testbench to the DUT. All packets should be stuck in the TX FIFO. But cannot go out to the XGMII side. Eventually, you'll get the TX_full signal on the packet TX interface. After, switch on TX Enable bit and look for packets on RX Monitor.
10	directed_pause	If dealing with multiple 10GE MACs, a pause frame is

		initiated when a certain specified threshold is crossed for the number of packets sent to any of the devices (50%, 80%) *Unable to test this pause frame as we don't have access to multiple 10GE MAC Cores.

Constraint Randomized Tests

Test ID	Test Name	Test Description + Completion
1	random_len_payload	Send randomized length packets with randomized payload.

Configuration Tests

Test ID	Test Name	Test Description + Completion
1	config_stats	Send TX packets and monitor statistics from the wishbone interface using Scoreboard's Register Comparator.

Flow Control Tests

Test ID	Test Name	Test Description + Completion
1	fc_ren_half	Will send more packets than are being read for a good amount of time. Check if packets kept in DUT's FIFO are only valid packets, by comparing read packets to Scoreboard's Packet Comparator. read_enable working at 50%
2	fc_ren_off	Try to read from DUT's FIFO when read_enable is off, and DUT's fifo has some randomized packets. Should give idle, invalid data.

7. Simulation and Regression

There is a python script available in ~/UVM-Project/Advait_Madhekar/scripts/reg_scr.py that allows the engineer to run all the test cases at once to see if anything fails, or if everything passes. It will run all the test cases one at a time in separate environments.

8. Conclusion

8.1 Documentation

Documentation located in /doc/verification_plan.pdf

8.2 Shortcomings

This testbench does have several shortcomings.

1. All the Scoreboard connections should be connected to an intermodulation monitor.
2. This testbench is currently not scalable enough to test individual components in the DUT. Other than knowing that there is an error in the design, it's impossible to know where that error has occurred within the design.
3. The test cases given in this plan are not nearly as comprehensive as they can be.
4. There is no predictor within the testbench to predict the output value from configurations. For example, if 10 packets are transmitted from testbench to DUT, there's no way to tell that there are 10 packets transmitted within our own testbench. So, there's no way to validate the configuration register data. Note: sequence items generated are not equivalent to sequence items driven to DUT.

I aim to fix these shortcomings in the long-term.

8.3 Summary

Overall, this comprehensive testbench should put the DUT through heavy testing.