10GE MAC CORE VERIFICATION PLAN

Advanced Verification with SystemVerilog OOP Testbench

UNIVERSITY OF CALIFORNIA – SANTA CRUZ
FALL 2021

Jesse Palomera
jpalomera@ucdavis.edu

Acknowledgements

First and foremost, I would like to commend the effort and valuable time given by the instructor of the course Benjamin Ting. His guidance and instruction have helped me immensely in understanding SystemVerilog OOP to complete this project. I also thank the University of California, Santa Cruz for providing access, and licenses to Synopsys VCS. I would also like to extend my deepest gratitude to my mentor Phil Morris for all the support and encouragement throughout this year. Phil is the reason why I decided to go and pursue a career in Digital Design and Verification.

Contents

Chapter 1

Project Scope

The objective of the project is to build a SystemVerilog class-based verification environment, otherwise known as the OOP Testbench on a 10Gb Ethernet MAC Core design. We were provided with the source code for the 10GE MAC Core as Verilog RTL along with a rudimentary testbench and simulation environment that we used as a reference to build the OOP Testbench from scratch using what we learned in the course. The 10GE MAC design is an Open-Source design where the RTL is available freely for anyone who wishes to use it. A block diagram of the core is presented in the figure below with clock domains indicated with green dotted lines. There are 3 different interfaces associated with the design: PKT_TX and PKT_RX interface, XGMII_TX and XGMII_RX interface, and Wishbone interface. In this project, we will be dealing only with the TX and RX interface, this is to say that the XGMII interface will be configured in loopback mode and the Wishbone will be disabled.

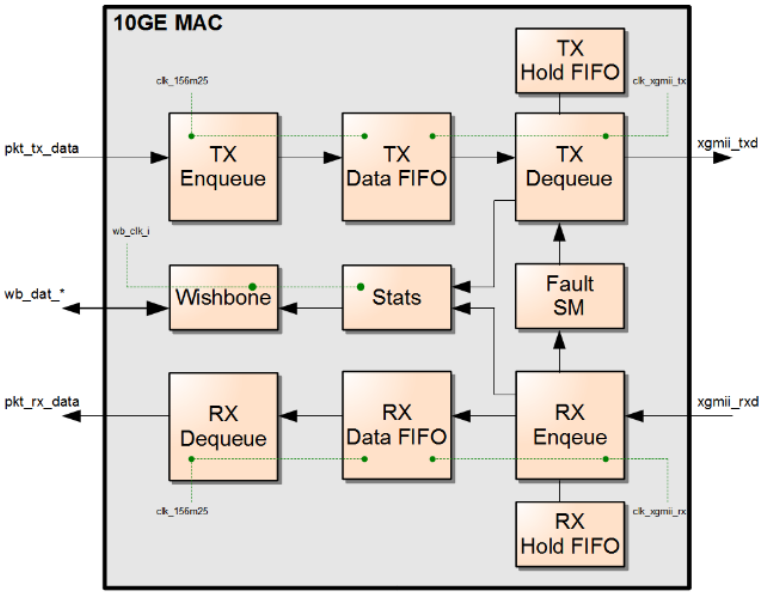

Figure 1: Block Diagram

Chapter 2

Architecture

2.1 Operation

TX Enqueue Engine

The Tx Enqueue Engine receives frames from the user's core logic and stores them in the transmit FIFO along with some additional flags such as SOP and EOP indicators. It also provides FIFO fill status (available signal) to the core.

TX FIFO

The Tx FIFO is 128-entry deep by default with 64-bit of data and 8-bit of status per entry. Since the FIFO can only store 512 bytes of data (64 x 64-bit), the MAC must operate in flow-through mode, meaning that the transmission of a frame on the XGMII interface can (and must) start while the frame is still being written to the FIFO on the enqueue side. The upper bits of the FIFO, bits 71:64, contains status information used by the Tx Dequeue Engine to maintain frame alignment. The format of the FIFO is shown in the next figure:

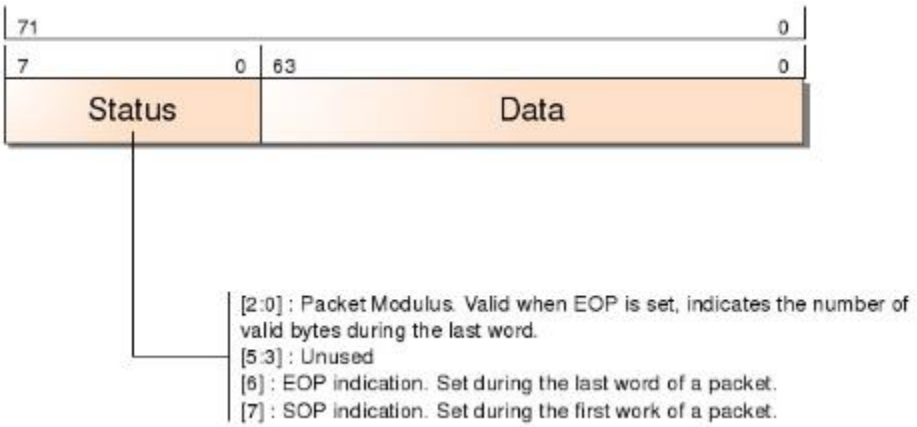

Figure 2: Tx FIFO Format

TX Dequeue Engine

The Tx Dequeue Engine contains two state-machines. The first state-machine reads data from the data FIFO and pads small packets to the minimum 64-byte required for Ethernet. The state-machine writes data to the holding FIFO and the CRC calculation logic. The purpose of the holding FIFO is to compensate for latency in the CRC calculation. The CRC logic operates in 64-bit data at the exception of the last word of the frame. Since the last word may contain 1 to 8 valid bytes, the CRC logic transitions to 8-bit mode at the end of the frame. Up to 8 cycles may be required to complete the calculation, hence the 8-word delay in the holding FIFO.

The Encoding State-Machine reads data from the holding FIFO after inserting the Ethernet preamble. To minimize the logic, this state-machine aligns all data on 64-bit boundary and generates flags indicating when 32-bit alignment is necessary to meet the required IFG. The IFG is calculated based on the accumulated Deficit Idle Count (DIC) and modulus of the current frame. Resulting Flags are passed to the Barrel Shifter which performs the final 32-bit alignment.

The RC Layer monitors link status signals from the Fault State-Machine and inserts remote fault messages when a local fault was detected. The fault signal is also passed to the Encoding State-Machine which stops transmitting packets during faults.

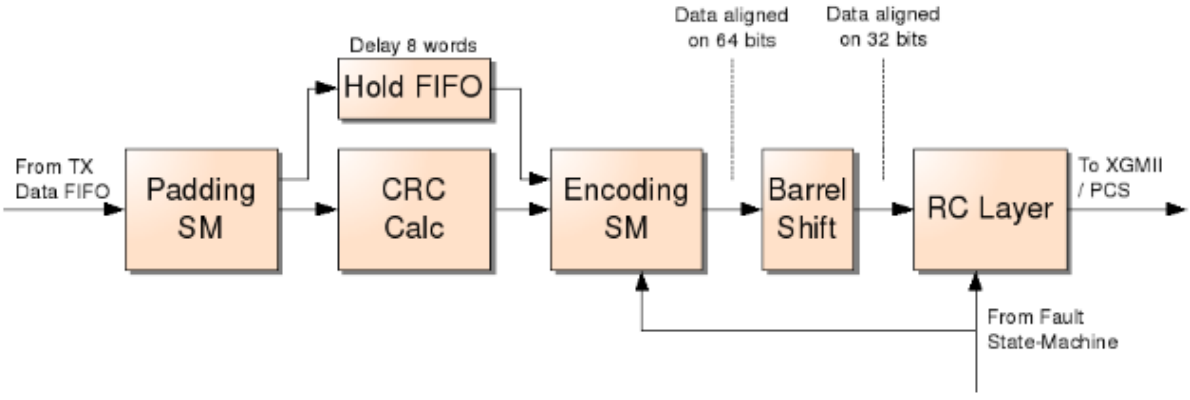

Figure 3: Tx Dequeue Engine

RX Enqueue Engine

In the Rx Enqueue Engine, the RC layer monitors the XGMII interface for fault conditions and pass the status to the Fault State-Machine. The Barrel Shifter looks for the start of frame delimiters on 32-bit boundary and re-aligns the data on 63-bit boundary. This method reduces the complexity of the next stages.

The Decoding State-Machine has the complex task of delimiting frames and detecting invalid frames such as fragments and runts. As it decodes the data, it writes a copy to the holding FIFO and CRC logic. The purpose of the holding FIFO is to compensate for latency in the CRC calculation. The CRC logic operates in 64-bit data at the exception of the last word of the frame. Since the last word may contain 1 to 8 valid bytes, the CRC logic transitions to 8-bit mode at the end of the frame. Up to 8 cycles may be required to complete the calculation, hence the 8-word delay in the holding FIFO.

As data emerges from the holding FIFO, it is written to the Rx Data FIFO along with SOP, EOP, and other flags. If the CRC logic reports an error during that time, an error flag is set in the next FIFO entry.

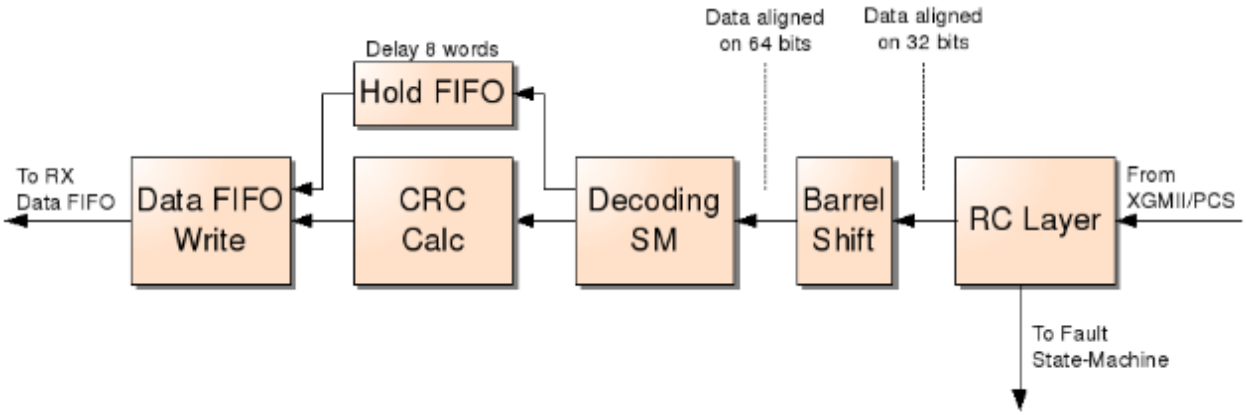

Figure 4: Rx Enqueue Engine

RX FIFO

The Rx FIFO is 128-entry deep by default with 64-bit of data and 8-bit of status per entry. Since the FIFO can only store 512 bytes of data (64 x 64-bit), the MAC must operate in flow-through mode, meaning that the transmission of a frame on the packet interface can (and must) start while the frame is still being written to the FIFO on the enqueue side. The upper bits of the FIFO, bits 71:64, contains status information used by the Tx Dequeue Engine to maintain frame alignment. Also included is an error flag used to propagate CRC error or any other conditions to the Dequeue Engine. The format of the FIFO is shown in the next figure:

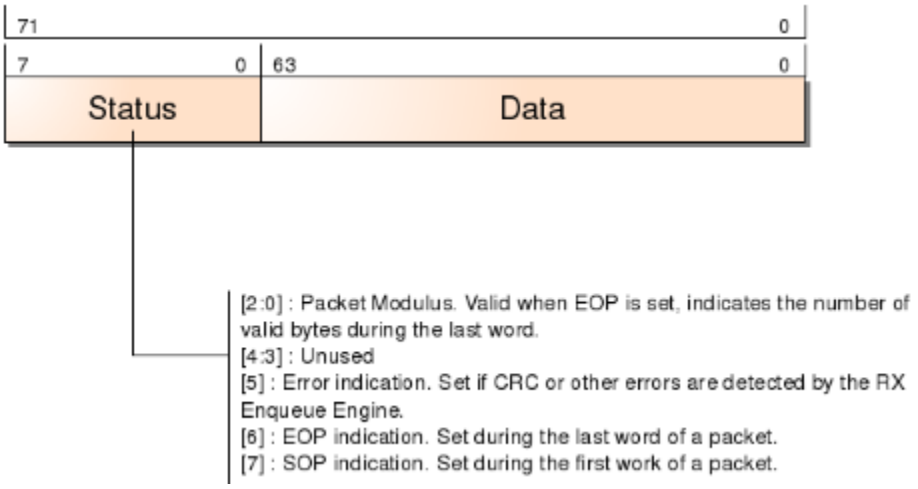

Figure 5: Rx FIFO Format

RX Dequeue Engine

The Rx Dequeue Engine interfaces with the user logic. Its primary function is to convert the internal status from the Rx FIFO into meaningful signals on the PKT_RX interface.

2.2 Clocks

The figure below describes the various clocks required by the 10GE MAC Core.

Name	Frequency	Description
wb_clk_i	30 - 156Mhz	Wishbone interface clock.
clk_156m25	156.25Mhz	Clock for transmit and receive packet interfaces towards user's core logic. "pkt_tx_*" and "pkt_rx_*" signal are timed to this clock.
clk_xgmii_rx	156.25Mhz	Clock for XGMII receive interface between 10GE MAC core and XAUI/PCS macro.
clk_xgmii_tx	156.25Mhz	Clock for XGMII transmit interface between 10GE MAC core and XAUI/PCS macro.

Figure 6: 10GE MAC Clocks

2.3 Resets

The figure below describes the various resets required by the 10GE MAC Core. The user must ensure that each reset is synchronously de-asserted with its corresponding clock. To ensure that transmit and receive FIFOs are initialized correctly, "reset_156m25_n", "reset_xgmii_rx_n" and "reset_xgmii_tx_n" must be de-asserted within 2-cycles of each other.

Name	Description
wb_rst_i	Wishbone interface reset. Active high. Must be de-asserted synchronous to wb_clk_i.
reset_156m25_n	Core packet interfaces clock domain reset. Active low. Must be de-asserted synchronous to clk_156m25.
reset_xgmii_rx_n	XGMII receive clock domain reset. Active low. Must be de-asserted synchronous to clk_xgmii_rx.
reset_xgmii_tx_n	XGMII transmit clock domain reset. Active low. Must be de-asserted synchronous to clk_xgmii_tx.

Figure 7: 10GE MAC Resets

2.4 I/O Ports

This section specifies the 10GE MAC Core IO ports that we will be dealing with in this project.

Packet Receive Interface

This interface is used to transfer received packets to the FPGA/ASIC core logic.

Port	Direction	Description
pkt_rx_ren	Input	Receive Read Enable: This signal should only be asserted when a packet is available in the receive FIFO. When asserted, the 10GE MAC Core will begin packet transfer on the next cycle. Signal should remain asserted until EOP becomes valid.
pkt_rx_avail	Output	Receive Available: Indicates that a packet is available for reading in receive FIFO.
pkt_rx_data [63:0]	Output	Receive Data: Little-endian format. First byte of packet will appear on pkt_rx_data [7:0].
pkt_rx_eop	Output	Receive End of Packet: Asserted when the last word of a packet is read from receive FIFO.
pkt_rx_val	Output	Receive Valid: Indicates that valid data is present on the bus. This signal is typically asserted one cycle after pkt_rx_ren was asserted unless FIFO underflow occurs.
pkt_rx_sop	Output	Receive Start of Packet: Indicates that the first word of a frame is present on the bus.
pkt_rx_mod [2:0]	Output	Receive Packet Length Modulus: Valid during EOP. Indicates valid bytes during last word. Little Endian mode: 0: pkt_rx_data [63:0] is valid 1: pkt_rx_data [7:0] is valid 2: pkt_rx_data [15:0] is valid 3: pkt_rx_data [23:0] is valid 4: pkt_rx_data [31:0] is valid 5: pkt_rx_data [39:0] is valid 6: pkt_rx_data [47:0] is valid 7: pkt_rx_data [55:0] is valid Big Endian mode: 0: pkt_rx_data [63:0] is valid 1: pkt_rx_data [63:56] is valid 2: pkt_rx_data [63:48] is valid 3: pkt_rx_data [63:40] is valid 4: pkt_rx_data [63:32] is valid

		5: pkt_rx_data [63:24] is valid 6: pkt_rx_data [63:16] is valid 7: pkt_rx_data [63:8] is valid
pkt_rx_err	Output	Receive Error: When asserted during a transfer, indicates that current packet is bad and should be discarded by users' logic. This signal is most likely asserted as the result of a CRC error. A frame of invalid size will also cause this signal to be asserted.

Packet Transmit Interface

This interface accepts packets to the FPGA/ASIC core logic.

Port	Direction	Description
pkt_tx_data [63:0]	Input	Transmit Data: Little-endian format. First byte of packet must appear on pkt_tx_data [7:0].
pkt_tx_val	Input	Transmit Valid: This signal must be asserted for each valid data transfer to the 10GE MAC.
pkt_tx_sop	Input	Transmit Start of Packet: This signal must be asserted during the first word of a packet.
pkt_tx_eop	Input	Transmit End of Packet: This line must be asserted during the last word of a packet.
pkt_tx_mod [2:0]	Input	Transmit Packet Length Modulus: Valid during EOP. Indicates valid bytes during last word. Little Endian mode: 0: pkt_tx_data [63:0] is valid 1: pkt_tx_data [7:0] is valid 2: pkt_tx_data [15:0] is valid 3: pkt_tx_data [23:0] is valid 4: pkt_tx_data [31:0] is valid 5: pkt_tx_data [39:0] is valid 6: pkt_tx_data [47:0] is valid 7: pkt_tx_data [55:0] is valid Big Endian mode: 0: pkt_tx_data [63:0] is valid 1: pkt_tx_data [63:56] is valid 2: pkt_tx_data [63:48] is valid 3: pkt_tx_data [63:40] is valid 4: pkt_tx_data [63:32] is valid 5: pkt_tx_data [63:24] is valid 6: pkt_tx_data [63:16] is valid 7: pkt_tx_data [63:8] is valid

pkt_tx_full	Output	Transmit Full: This signal indicates that transmit FIFO is nearing full and transfers should be suspended at the end of the packet. Transfer of next packet can begin as soon as this signal is de-asserted.

Chapter 3

Verification Environment

3.1 SystemVerilog for Verification

The design of ICs in industry today have become sophisticated which forced designers to steer away from a self-developed testbench in Verilog or with Verilog and C++. Since the complexity of designs have exceptionally increased, the endeavors to verify a design have dominated the process which led to the development of verification tools and methodologies. SystemVerilog is an object-oriented programming (OOP) language that supports hardware verification. Amongst many features, it provides the ability of using a layered testbench with a constrained-random stimulus. This allowed the self-developed “jack of all trades” testbench to convert into each component in SystemVerilog OOP testbench becoming a “specialist” in the verification environment. Figure 8 shows a standard OOP testbench architecture that contains various components like program block, clocking block, interface, packet generator, driver, environment, monitor, scoreboard, coverage, and a top-level module that instantiates the Design Under Test (DUT), interfaces, and clocks. The table below describes each component of the OOP testbench.

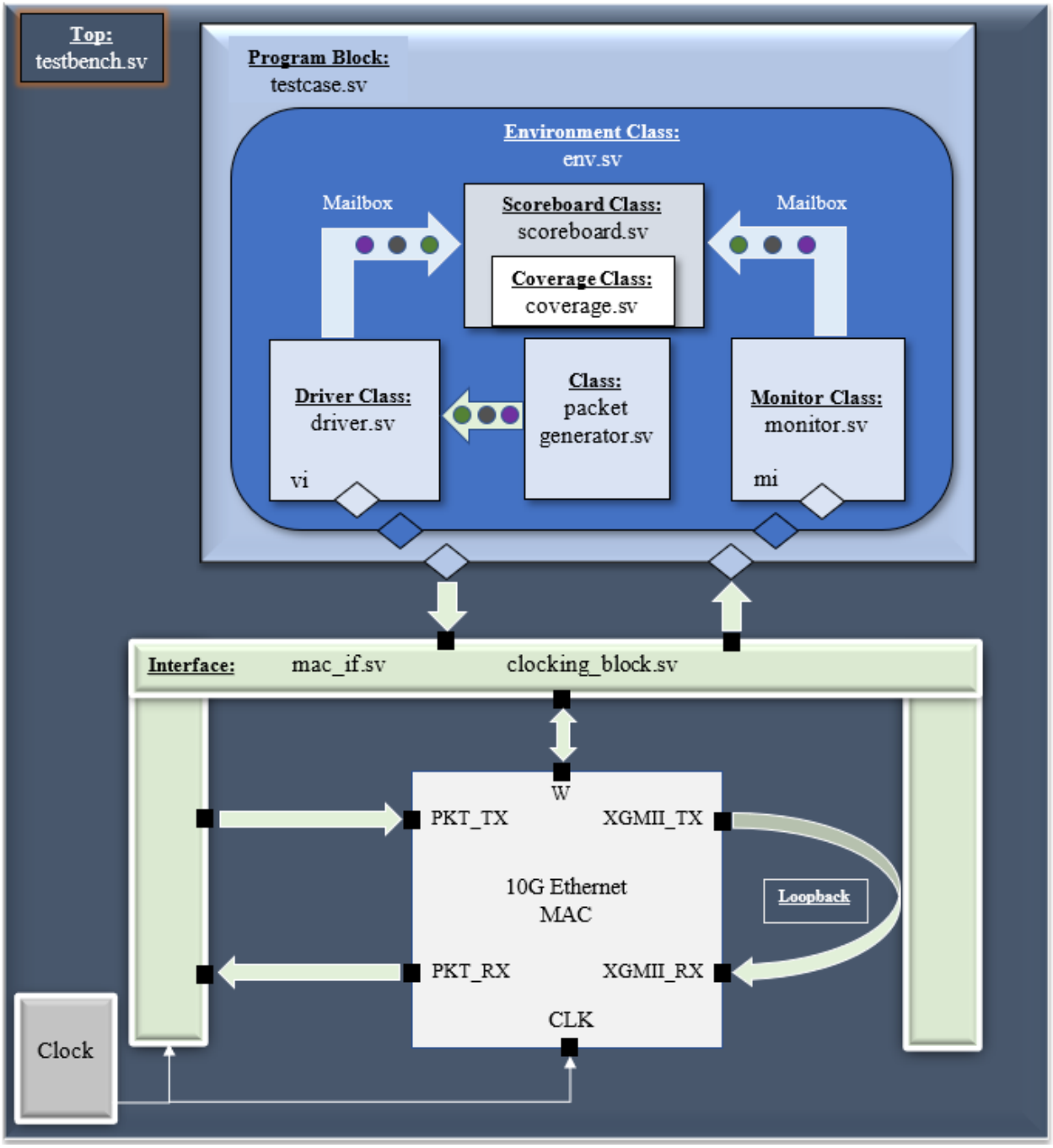

Figure 8: Standard OOP Testbench Architecture

Component	Description
Program Block	The program block separates the testcase and testbench, it is a border between RTL code and testbench. Program Block is used to minimize race conditions by scheduling events so that they happen in different delta cycles.
Clocking Block	The clocking block separates the functional behavior of the design from its clocking behavior. It assembles signals that are synchronous to a particular clock and makes their time explicit.
Interface	The interface is simply a bundle of wire. It is just like a module which can be instantiated and can also instantiate other interfaces.
Driver Class	The driver component has the logic that is driven to the DUT, and sends data received by the packet generator via mailbox. The driver talks to the design through its interface, this communication is made possible via virtual interface, represented by the diamond shapes. The driver also sends the transaction driven to the scoreboard via mailbox.
Monitor Class	The monitor component collects and monitors the transaction collected on the output of the DUT and forwards the collected transactions to the scoreboard via mailbox.
Scoreboard Class	The scoreboard component checks if the design is working as expected by comparing the transactions driven into the DUT by what was collected by the monitor.
Environment Class	Driver, Monitor, and Scoreboard are the components of the environment, and the environment connects them. It is also responsible of calling tasks inside the driver, monitor, and coverage to drive, collect packets and collect coverage.

Coverage Class	There are two types of coverage: 1. Code Coverage: Measures how good the code is covered during simulation. It is an add-on to language of Verilog, and it is generated automatically by the simulator by running commands. 2. Functional Coverage: This is the heart and soul of the RTL design. It is user-defined coverpoints and covergroups that check the functionality of the design and cover corner cases
Testbench	This is the top-level module that contains the instantiation of the Clocks, DUT, Testcase program block, and instantiation of Interfaces.

Packet Class

The packet class contains data members for the fields of the Tx frame. The members are:

1. Valid (tx_val)
2. Start of Packet (tx_sop)
3. Modulo (tx_mod)
4. Data (tx_data)
5. End of Packet (tx_eop)
6. Inter-Packet Gap (IPG)

Val	SOP	Modulo	Data	EOP
(1 bit)	(1 bit)	(3 bits)	[7:0] data []	(1 bit)

Figure 9: Tx Frame

3.2 Testcases

Each of the following test cases are contained in a program block each in their own SystemVerilog file. Each program block declares and creates the environment class and applies constraints to the stimulus required for the test case. The conditions for each of the test cases are that the XGMII interfaces are connected in loopback.

1. Loopback
    - I will be driving constrained-random legal frames into the DUT to verify that it can be transmitted and received through the DUT logic. Since the XGMII interfaces are connected in loopback so the packet transmitted and received should be equal. This test will print whether it has passed and compare if the number of frames received matches the number transmitted.
2. Missing EOP
    - I will be driving constrained-random legal frames into the DUT while omitting the End of Packet signal at the end of the packet. The Transmit Interface states this signal must be asserted during the last word of a packet. Furthermore, the Packet Length Modulus is valid during EOP to indicate valid bytes during the last word. This is to verify that the DUT can detect the missing EOP signal and indicate that the current packet is bad and should be discarded by user's logic by driving the signal "pkt_rx_err" high.
3. Missing SOP
    - Like the Missing EOP test case the DUT must be able to detect the missing SOP signal. The Transmit interface specifications states this signal must be asserted during the first word of a packet, so the DUT must not transmit the packet.

4. Undersized Packet
    - I will be driving constrained-random frames into the DUT with the data constrained to less than the minimum 64 bytes. This is to verify that the DUT can detect an undersized packet and pad it up to the minimum legal size (64-byte) required for Ethernet. This packet should be transmitted and received with the packet padded with 0's.
5. Oversized Packet
    - Like the Undersized Packet test case, I will constrain the data to be larger than 1500 bytes and the DUT must be able to transmit an oversized packet. The packet transmitted and received should match. The DUT should assert the Tx FULL signal to indicate that the FIFO is nearing full.
6. Zero IPG Packet
    - I will be driving constrained-random frames into the DUT with the IPG constrained to 0. This is to verify that the DUT can transmit and receive packets without clock signal delays between frames. There should not be an error.

Conclusion

I have successfully designed and created an OOP testbench from scratch using SystemVerilog for a 10GE MAC Core provided in RTL. I was able to effectively use the rudimentary testbench provided as a guide to implement an OOP testbench. It is apparent how useful the OOP testbench is for providing specialists to perform a specific task to minimize the risk of race conditions. I found this course to be exceptional being able to learn, understand, implement the following topics covered:

- Concurrent Processes
- Constraints/Overriding
- Randomization
- Classes/Container Classes
- Inheritance/Polymorphism/Shallow Copy
- Advanced Data Structures

Potential Bugs

Undersized Packet: The 10G Ethernet MAC Core specification states that the Tx Dequeue reads data from the holding FIFO, and pads small packets to the minimum 64-byte required for Ethernet. However, as this test was performed, I found that the design only pads small packets up to 60-bytes, ultimately failing the testcase.

Oversized Packet: When sending large number of bytes into the design, the specifications state when the FIFO is nearing full, the PKT TX FULL signal should be asserted by the design to alert the user to stop sending packets at the end of the current packet. Once the signal is de-asserted it is okay to start sending packets. However, when applying tests, the signal never was asserted resulting in PKT RX ERR signal to be asserted.