VERIFICATION PLAN FOR 10GB ETHERNET MAC CORE

System and Functional Verification using UVM

Abstract

Test plan and user manual created as part of the final project for VLSI410

Vinai Birbal

vdbirbal@uwaterloo.ca

Contents

1 Introduction 2

2 Objective 2

3 Device Under Test (DUT) 3

3.1 DUT Description 3

3.2 DUT Block Diagram 3

3.3 Wishbone Interface 5

4 Description of Verification Environment: UVM Testbench 6

4.1 UVM Methodology 6

4.2 UVM Testbench Components 6

i) Sequence item class 6

ii) Sequence Class 6

iii) Test Class 7

iv) Environment class 7

v) Agent class 7

vi) Sequencer class 8

vii) Driver class 8

viii) Monitor class 8

ix) Scoreboard class 9

x) Virtual Sequence class 9

xi) Virtual Sequencer class 9

xii) Coverage class 9

5 Block Diagram for Verification Environment 10

6 Test Cases 11

7 Functional Coverage 13

8 Summary and Conclusion 14

9 References 15

1 Introduction

This document details the UVM verification environment and test plan developed for a 10GB Ethernet MAC Core. It includes a summary of the functionality of the 10GB Ethernet Core, a description of the UVM test bench, a block diagram for the verification environment, descriptions of the test cases covered, and an evaluation of the coverage achieved.

2 Objective

The primary goal is to design and build a comprehensive UVM verification environment for a 10Gb Ethernet MAC Core. This includes understanding the provided specifications, developing a test plan, and creating UVM testbench source code to validate the design thoroughly. This document is intended to explain the verification environment developed for the 10GB Ethernet MAC Core and to aid in its use and facilitate future extension of the environment if necessary.

3 Device Under Test (DUT)

3.1 DUT Description

The DUT being verified is based on the 10GE MAC Core Specifications Rev 0.8 (Tanguay, 2013) dated 1/19/2013 and made available on OpenCores.Org. It is designed for easy integration with proprietary custom logic and features a POS-L3 like interface for data path and a Wishbone compliant interface for management.

There are three major interfaces associated with the 10gEthernetMAC design:

1. ***PKT_TX*** and ***PKT_RX*** interface
2. ***XGMII_RX*** and ***XGMII_TX*** interface
3. ***WISHBONE*** interface

3.2 DUT Block Diagram

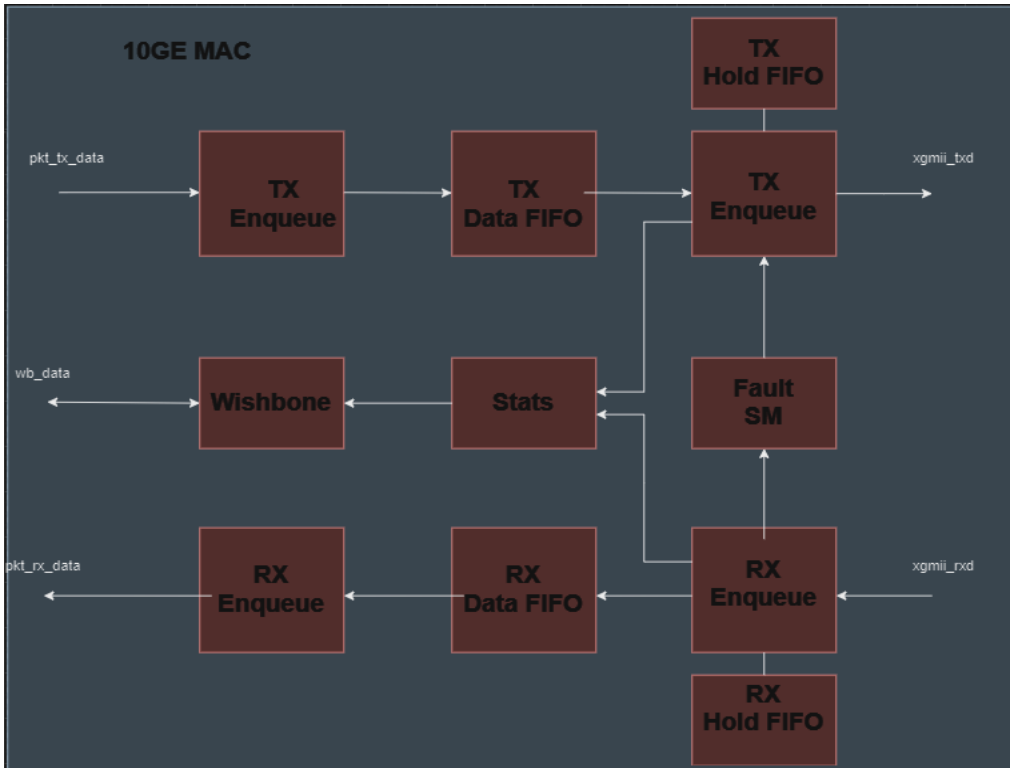


Figure 1- Block Diagram for 10GB Ethernet MAC Core

PKT_TX and PKT_RX interface

The PKT_TX and PKT_RX interface uses a data frame with 64-bits of data and 8 bits of status per entry for the pkt_tx_data outputs and pkt_rx_data inputs as shown in the block diagram above (*Figure 1*). Each TX and RX FIFO has a dept of 64 entries by default meaning it can only store 512 bytes of data. The upper bits of the FIFO contains status information used by the Dequeue and Enqueue engines to maintain frame alignment.

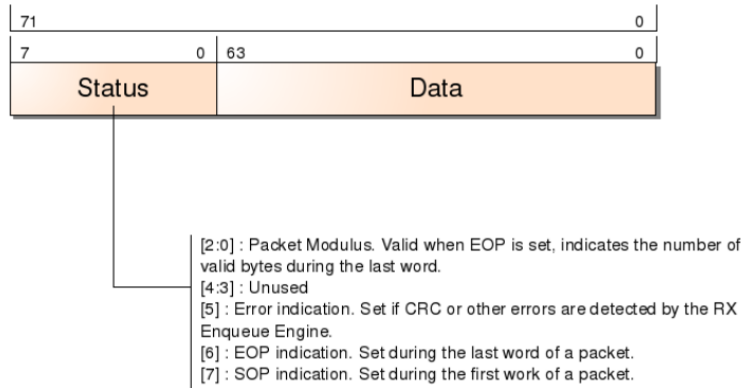


Figure 2- Data Frame for DUT

The following tables summarize the ports used for the packet interfaces.

Port	Direction	Description
pkt_rx_ren	Input	Received Read Enable
pkt_rx_avail	Output	Receive Available
pkt_rx_data [63:0]	Output	Receive Data
pkt_rx_eop	Output	Receive End of Packet
pkt_rx_val	Output	Receive Valid
pkt_rx_sop	Output	Receive Start of Packet
pkt_rx_mod	Output	Receive Packet Length Modulus
pkt_rx_err	Output	Receive Error

Table 1- Ports available in Packet Receive Interface

Port	Direction	Description
pkt_tx_data [63:0]	Input	Transmit Data
pkt_tx_val	Input	Transmit Valid
pkt_tx_sop	Input	Transmit Start of Packet
pkt_tx_eop	Input	Transmit End of Packet
pkt_tx_mod	Input	Transmit Packet Length Modulus
pkt_tx_full	Output	Transmit Packet Full

Table 2- Ports available in Packet Transmit Interface

XGMII_RX and XGMII_TX interface

The 10GE MAC DUT uses a simplified 64-bit XGMII interface clocked on the rising edge only for the xgmii_txd and xgmii_rxd ports as shown in the block diagram above (*figure 2*). The interface includes a 8-bit XGMII transmit control signal with each bit corresponding to a byte on the 64-bit interface. When high, it indicates that the byte is a control byte while when low it indicated that the byte carries data.

3.3 Wishbone Interface

The Wishbone Interface is based on the Wishbone System-on-Chip Interconnect Architecture. Revision B.3. (OpenCores, 2022) IP. It is used to access the registers involved in the management and configuration of the DUT. The registers accessed by the Wishbone Interface are listed below:

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

The following tables summarize the ports used for the Wishbone interface.

Port	Direction	Description
wb_adr_i [7:0]	Input	Address input
wb_cyc_i	Input	Wishbone cycle
wb_data_i [31:0]	Input	Wishbone data input
wb_stb_i	Input	Wishbone strobe
wb_we_i	Input	Wishbone write enable
wb_ack_o	Output	Wishbone acknowledge
wb_dat_o [31:0]	Output	Wishbone data output
wb_int_o	Output	Wishbone Interrupt

4 Description of Verification Environment: UVM Testbench

4.1 UVM Methodology

Universal Verification Methodology (UVM) is an IEEE-standard, object-oriented verification framework designed to ensure the functional correctness of RTL designs using SystemVerilog. Combining elements from OVM, VMM, and TLM, UVM offers a comprehensive set of base classes for creating modular and reusable testbenches. Its architecture supports horizontal reuse across different projects and vertical reuse at various integration levels within the same project. UVM's standardized approach, leveraging object-oriented principles, enhances collaboration, reduces errors, and accelerates development by enabling the reuse of pre-verified components, making it an effective tool for verifying complex digital designs.

4.2 UVM Testbench Components

The following components were used in the UVM testbench:

i) Sequence item class

The Sequence Item class is derived from `**uvm_sequence_item** and represents the data that will be sent to the DUT. It is a data carrier and can include fields that represent the inputs to the DUT. It contains the transaction data and methods for randomization and printing. Each transaction is an instance of a sequence item. These sequence items are sent to sequencer which then forwards it to the driver. The following sequence item classes are used in the verification environment:

- Packet – Main sequence items that contain the data being sent and received by the DUT.
- reset_sequence_item – Generates resets for the DUT as part of the verification environment.
- wishbone_sequence_item- Used to configure the DUT by setting values to the appropriate registers.

ii) Sequence Class

The sequence class is derived from `**uvm_sequence** class and generates the sequence items used in the verification environment. It defines the body of the sequence, where sequence items are created, randomized, and sent to the driver via the sequencer. The following sequence classes are used in this testbench and are contained in the Virtual Sequence Class:

- packet_sequence – Generates the ethernet packets that are transmitted to the sequencer in the TX agent.
- reset_sequence- Generates the reset _item that are transmitted to the sequencer in the reset agent.
- wishbone_sequence- Generates the wishbone_sequence items that are transmitted to the sequencer in the wishbone agent.

iii) Test Class

The test class derives from `**uvm_test** and is the top element in the UVM component hierarchy. It is responsible for configuring the testbench environment and running the sequences during the simulation. There is one main test class used in this testbench with other test classes extended from it to test different functionality of the DUT:

- test_base- This is where the environment is created, and the non-virtual sequences started.

iv) Environment class

The Environment class derives from `**uvm_env** and instantiates the necessary agents and other components required for the verification environment. It contains the agents that facilitate the transfer of the sequence items generated in the sequence classes to the DUT and the transfer of sequence items produced by the DUT to the scoreboard. The scoreboard also exists within the environment class.

v) Agent class

The Agent class derives from `**uvm_agent** and contains the sequencer, driver, and monitor. It serves as a container for these components and handles their configuration and connections. It also coordinates these components to ensure proper communication between the DUT and the testbench. There are four agents that are used in this testbench:

- Reset Agent- Contains the reset sequencer and reset driver. Facilitates the transmission of the reset sequence items from the reset sequence class to the reset interface on the DUT.
- TX Agent- Contains the tx sequencer, tx driver and tx monitor. Facilitates the transmission of the packet sequence items from the sequence class to the tx interface on the DUT.
- RX Agent- Contains the rx monitor. Facilitates the transfer of packet sequence items from the rx interface on the DUT to the scoreboard.

- Wishbone Agent- Contains the wishbone sequencer, wishbone driver and wishbone monitor. Facilitates the transfer of wishbone sequence items from the wishbone sequence to the wishbone port on the DUT.

vi) Sequencer class

The Sequencer class derives from `**uvm_sequencer** and manages the flow of sequence items to the driver. It is contained within the agent and connects the sequence class to the driver class. It provides services to facilitate the transfer of sequence items from the sequence class to the driver class. There are three sequencer classes used in this testbench:

- tx_sequencer- Exists within the tx_agent and connects the packet class to the tx_driver.
- reset_sequencer- Exists within the reset_agent and connects the reset_sequence to the reset_driver.
- wishbone_sequencer- Exists within the wishbone agent and connects the wishbone_sequence to the wishbone_driver.

vii) Driver class

The Driver class derives from `**uvm_driver** and is responsible for converting sequence items into DUT signal transactions. It exists within the agent class and drives the DUT signals based on sequence items. It contains a virtual interface that converts the sequence items received from the sequencer to a signal that can be transmitted to the DUT. The following drivers are used in this testbench:

- tx_driver- Exists within the tx_agent and drives the pkt_tx_data interface on the DUT.
- reset_driver- Exists within the reset_agent and drives the reset interface on the DUT.
- wishbone_driver- Exists within the wishbone_agent and drives the wishbone interface on the DUT.

viii) Monitor class

The Monitor class is derived from `**uvm_monitor** is responsible for observing the DUT's signals without driving them. It captures the activity on these signals and potentially forwards the observed data to other components like the Scoreboard. It exists within the agent class, contains a virtual interface and is used to receive responses from the DUT. The virtual interface is used to convert an RTL signal from the DUT to a sequence item that can be used in the UVM verification environment. The following monitors exists in this testbench:

- tx_monitor- Located within the tx_agent. Receives signals from the DUT which are transmitted as sequence items to the scoreboard class.
- rx_monitor- Located within the rx_agent. Receives signals from the DUT which are transmitted as sequence items to the scoreboard class.
- wishbone_monitor-Located within the wishbone agent. Receives signals from the DUT which are transmitted as sequence items to the scoreboard class.

ix) Scoreboard class

The Scoreboard class is derived from `**uvm_scoreboard** and is responsible for comparing the actual results observed by the Monitor against the expected results. It determines if the DUT behaviour matches the expected functionality and reports mismatches. It exists within the environment class and receives sequence items from the monitor class.

x) Virtual Sequence class

The Virtual Sequence class extends from `**uvm_sequence**. It manages multiple sequences, ensuring they start and stop in a coordinated manner. The virtual sequence class contains the various sequence classes used in the UVM verification environment. These are:

- packet_sequence
- wishbone_seqence
- reset_sequence

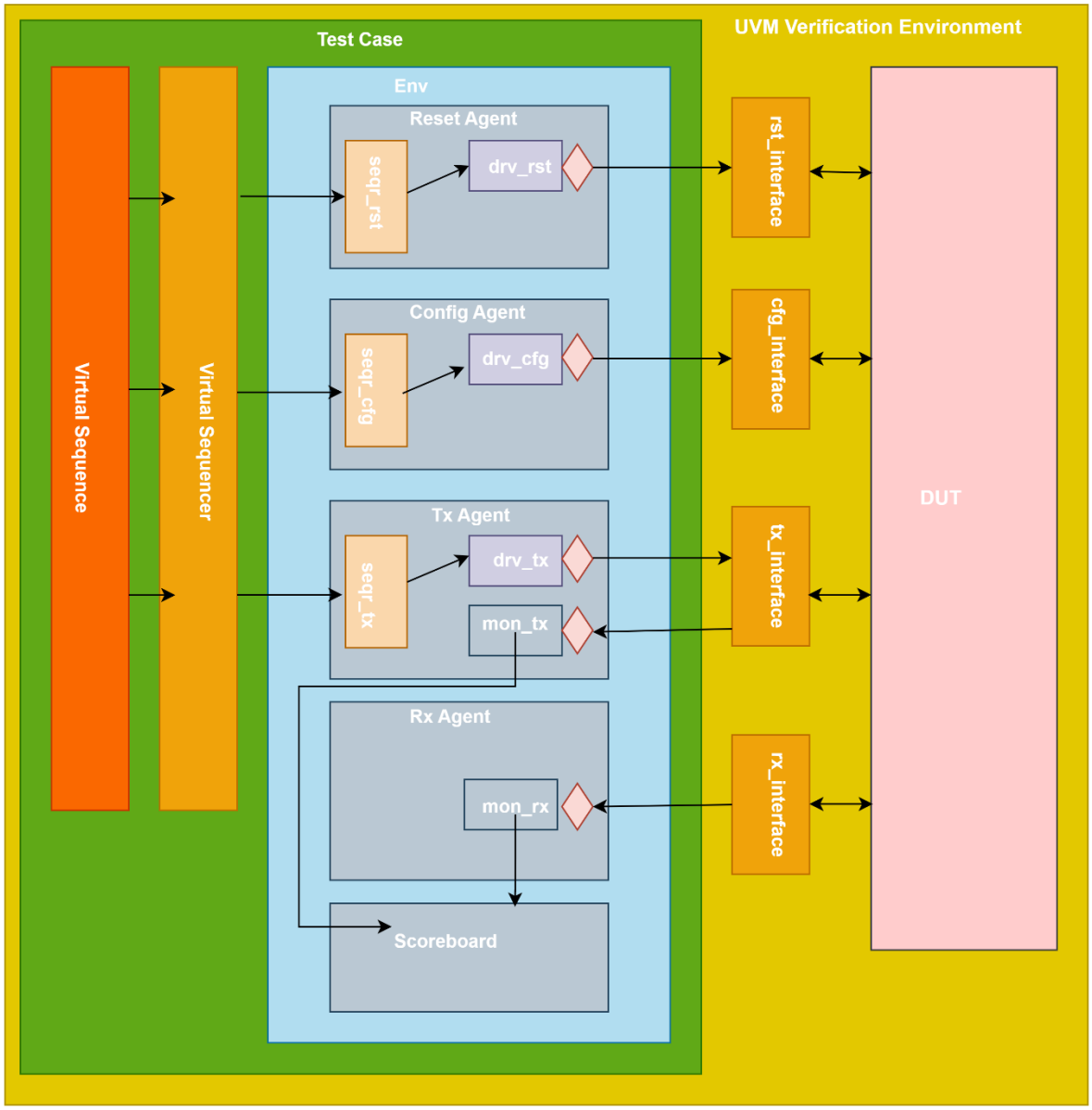
xi) Virtual Sequencer class

The Virtual Sequencer class extends from `**uvm_sequencer** and contains references to the sequencers for the different agents. It connects the sequence classes located in the virtual sequence to the appropriate sequencers located in the agent classes. The connection is done in the environment class.

xii) Coverage class

The Coverage class derives from `**uvm_subscriber** and collects coverage data during the simulation. It includes cover groups to capture functional and code coverage metrics, ensuring comprehensive testing. **Due to time constraints the Coverage Class is still awaiting implementation as it was optional.**

5 Block Diagram for Verification Environment



6 Test Cases

The following testcases, listed in the table below, were created using the 10GB Ethernet MAC Core specification document.

Test	Specification	Test Description	Status
Loopback	Normal sized Packets are transmitted to and from the DUT correctly. In Loopback the packet sent to the TX port of the should be the same as that received from the RX port.	The DUT is connected in loopback mode with the xgmii_txd port connected to the xgmii_rxd port. It is then initialized and reset. A normal sized packet is then sent on the TX interface. After a delay a packet is received on the RX interface. The packet length and payload for the packets transmitted on the TX interface and received from the RX interface are then compared in the scoreboard. No interrupts should be generated in the Interrupt Pending Register.	Implemented
Small Packet (<64 bytes)	The transmitter adds PAD bytes to packets less than 64 bytes to meet the requirement of ethernet.	The DUT is connected in loopback mode as above. It is then initialized and reset. A small packet is then sent on the TX interface. After a delay a packet is received on the RX interface. The packet length and payload for the packets received from the RX interface are then compared in the scoreboard against the expected packets generated from the TX interface with padding. No interrupts should be generated in the Interrupt Pending Register.	Implemented
Large Packet (>1514 bytes)	Lage packets are packets with a payload greater than the standard 1500 bytes. Large packets should be transmitted successfully without errors. Packets are tested for payload sizes < 3,000 bytes.	The DUT is connected in loopback mode as above. It is then initialized and reset. A large packet is then sent on the TX interface. After a delay a packet is received on the RX interface. The packet length and payload for the packets received from the RX interface are then compared in the scoreboard against the packets from the TX interface to ensure they are the same.	Implemented

		No interrupts should be generated in the Interrupt Pending Register.	
Remote Fault (TX/RX)	Transmission of packets from DUT is stopped during remote faults and an error flag is set in the next FIFO entry. The interrupt is cleared on read.	Invalid packet is transmitted to DUT. Scoreboard observes stop in transmission of packets. The Interrupt Pending Register is read to ensure that the Remote Fault Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.
RX fragment error	The RX Fragment interrupt is triggered when a frame with a start of frame delimiter and without an end of frame delimiter is received. The interrupt is cleared on read.	A packet is transmitted with a start of frame delimiter but no end of frame delimiter, followed by another packet. The Interrupt Pending Register is read to ensure that the RX Fragment Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.
RX Data Underflow	The RX Data FIFO Underflow interrupt is asserted when the "pkt_rx_ren" signal is asserted, and no data is available in the receive FIFO. Interrupt is cleared on read.	The "pkt_rx_ren" signal is asserted when the RX FIFO is empty (before any packets are sent). The Interrupt Pending Register is read to ensure that the RX Data Underflow Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.
Rx Data Overflow	The RX Data FIFO Overflow interrupt is asserted when the RX FIFO is full, and it receives another packet.	The "pkt_rx_ren" signal is de-asserted, and packets sent to the DUT until the RX FIFO overflows. The Interrupt Pending Register is read to ensure that the RX Data Overflow Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.
TX Data Underflow	The TX Data FIFO Underflow interrupt is asserted when the "pkt_tx_valid" signal is asserted, and no data is available in the transmit FIFO. Interrupt is cleared on read.	The "pkt_tx_valid" signal is asserted when the TX FIFO is empty (before any packets are sent). The Interrupt Pending Register is read to ensure that the TX Data Underflow Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.

TX Data Overflow	The TX Data FIFO Overflow interrupt is asserted when “pkt_tx_valid” is asserted and the TX FIFO is full. The pkt_tx_full signal should be asserted when the transmit FIFO is nearing full.	The “pkt_tx_valid” signal is de-asserted, and packets sent to the DUT until the TX FIFO overflows. The Interrupt Pending Register is read to ensure that the TX Data Overflow Interrupt is asserted. The register is then read again to ensure the interrupt is de-asserted.	Not implemented as yet.

7 Functional Coverage

Functional coverage is a measure of which design features have been verified by tests. Functional coverage should be close to 100% to ensure that the DUT meets the predefined specifications. Coverage data is collected in the Coverage Class to be displayed. This verification environment is test bench based and verification is done by testing the important aspects of the operation of the core IP and ensuring that it operates as per the specifications.

8 Summary and Conclusion

A UVM Verification Environment for a 10GB Ethernet MAC Core was successfully developed. In this project, UVM significantly streamlined the verification process by providing a standardized framework and reusable components. The structured approach facilitated better organization, readability, and debugging of the testbench, ultimately leading to more efficient and effective verification.

One of the primary challenges I faced was transitioning from the rudimentary testbench to a well-structured UVM environment. The initial setup required a steep learning curve, especially in understanding the existing spaghetti-like testbench code and reorganizing it into modular components.

If I were to build the next version of this UVM testbench, I would focus on enhancing the coverage metrics and incorporating more advanced verification features, such as functional coverage using Covergroups and SystemVerilog Assertions (SVA). This would also involve implementing components for the Wishbone interface and verifying interrupt like faults and overflow errors which were not implemented thus far due to a shortage of time.

The UVM environment is designed to be scalable and reusable. The modular design and use of standard UVM components ensure that the testbench can be easily adapted to other projects or extended with additional features as needed. This scalability makes this verification environment an asset for future verification tasks. Thus, if I had to start a new project that was similar to this I would try to reuse as much of the verification environment as possible while making necessary changes for the aspects unique to the new project.

In conclusion, this project has provided invaluable hands-on experience in designing and building a UVM verification environment for a complex digital design. Through this process, I have gained a deeper appreciation for the power and flexibility of UVM, as well as the importance of a well-structured and systematic verification approach. The knowledge and skills acquired during this project will undoubtedly be beneficial in my future endeavours in the field of hardware verification.

9 References

OpenCores. (2022, September 7). *Wishbone System-on-Chip Interconnect Architecture. Revision B.3.* Retrieved from OpenCores.Org: www.silicore.net/wishbone.htm.

Tanguay, A. (2013, 1 19). *OpenCores.Org.* Retrieved from https://opencores.org/projects/xge_mac

Ting, B. (2017). *Advanced Verification using SystemVerilog OOP Testbench*.

Ting, B. (2017). *OOP Testbench Workbook*.