ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 9(5), DOI: 10.17485/ijst/2016/v9i5/87173, February 2016* *ISSN (Online) : 0974-5645*

Design and Verification of Slave Block in Ethernet Management Interface using UVM

K. Jagannadha Naidu[*] and M. Srikanth

VLSI Division, School of Electronics Engineering, VIT University, Vellore - 632014, Tamil Nadu, India;
jagannadhanaidu.k@vit.ac.in, srikanth20k@gmail.com

Abstract

Objective: Verification of the Slave Block in Ethernet Management Interface using UVM. **Methodology:** Management Data Input Output (MDIO) and Management Data Clock (MDC) is a two-wire interface used by Ethernet Station Management Entity to configure as well as read status from various PHY devices connected to it. Universal verification methodology is used to verify integrated designs. Verification of the Slave block in Ethernet Management Interface is done through UVM. **Findings:** Verification environment for the Slave Block in Ethernet Management Interface is built using UVM. 94.44% functional coverage and 97.96 code coverage is achieved. **Applications:** Ethernet protocol is used in the computer communication.

Keywords: Ethernet Management Interface, System Verilog, UVM, Verification

1. Introduction

Nowadays, transistors in a single device are increasing from millions to billions which are integrated on a chip. As hardware designing became more and more complex, Semiconductor industry started building Intellectual Property cores (IP) which are designed by many Semiconductor vendors. These all IP cores are integrated into a system on chip designs for multipurpose applications.

In designing IP core from the specifications, it may slightly deviate from the specifications. If deviations are neglected then small deviations leads to bugs in real time scenarios. So compared to designing, verification also plays a major role in verifying the designs for eliminating bugs. This can prevent the semiconductor company from many losses.

Networking technologies are approaching more and more. That can be a wired network or wireless network. In both the cases speed and interconnections play a vital role. In the wired network, Ethernet is the communication factor for Local area networks and other area networks. It is standardized as IEEE 802.3. The devices communicate through Ethernet by dividing the stream of data into frames. The information stored in frame are source addresses, destination addresses and error checking mechanisms for proper transmission and reception.

Management Data Input and Output interface is specified in the IEEE 802.3 standard, for providing a serial interface to transfer management data between Ethernet Media Access Controller (MAC) and a Physical Layer (PHY). Ethernet management interface entity is the device that handles MDIO and MDC. The Slave Block in the Ethernet Management Interface entity translates the Management Data Input and Output/Management Data Clock transactions to access the Registers in the Physical (PHY) device.

As the bugs are increasing in the complex design, verifying the designs through conventional techniques is time consuming in identifying the bugs. Many methodologies have been developed by Semiconductor companies for verification of the design. They are Open Verification Methodology (OVM), Universal Verification Methodology (UVM)[1], etc.

** Author for correspondence*

These Methodologies are currently used in industry. Universal Verification Methodology is derived from Open Verification Methodology and few other Methodologies. The Universal Verification Methodology has class libraries which brings automation for the verification environment. These class libraries are built using System Verilog[2].

The Hardware Description Language (HDL) simulators and Hardware Verification Language (HVL) verification applications are used to be different. System Verilog integrates this HDL and HVL into Hardware Description Verification Language (HDVL). It combines all the design and verification features like assertion based verification, constrained randomization and coverage.

Design the Slave Block in Ethernet Management interface entity using Verilog Hardware description language. Verifying the design Slave Block in Ethernet Management interface entity using UVM Methodology. To achieve 100% code coverage and functional coverage

2. Universal Verification Methodology

The integrated circuits should comply with the specifications as improper verification of the design leads to disasters in the industry. Many industrial methodologies have been introduced by Semiconductor Vendors. OVM by Mentor and Cadence, eRM by Verisity, VMM by Synopsys and AVM by Mentor graphics.

To make standardization in the methodology Universal Verification Methodology (UVM) has been introduced for verification by Accellera[3] i.e. jointly with the semiconductor companies like Synopys, Mentor, Cadence. This is now current industry standard methodology.

It has features like constrained random verification, coverage driven verification, reconfigurable, flexible environment, Verification Intellectual Property (VIP) reusability, Transaction Level Communication (TLM), Layered stimulus and Register layer[4]. As it developed from OVM, it has object oriented design and some methodology, these all can be applied to any UVM project. The UVM has built-in class libraries[5] which provide the building blocks for developing verification objects, components in System Verilog. It provides open source library from Accellera which is compatible with any EDA simulator which supports System Verilog. The other features of UVM are reusability and portability. This is portable with any EDA simulator that supports System Verilog.

2.1 UVM Classes

The classes are derived from inbuilt class libraries[6] which have been shown in the Figure 1. For generating data items a sequence class must be derived from “uvm_sequence_item” and routing the sequences, the sequencer class must be derived from “uvm_sequencer”, similarly the driver, monitor and other components are derived from their respective class libraries.

Each class has its own useful methods and the user decides which class to use for verifying the design of the DUT. There are class libraries for the objects and components which is shown in the Figure 1.

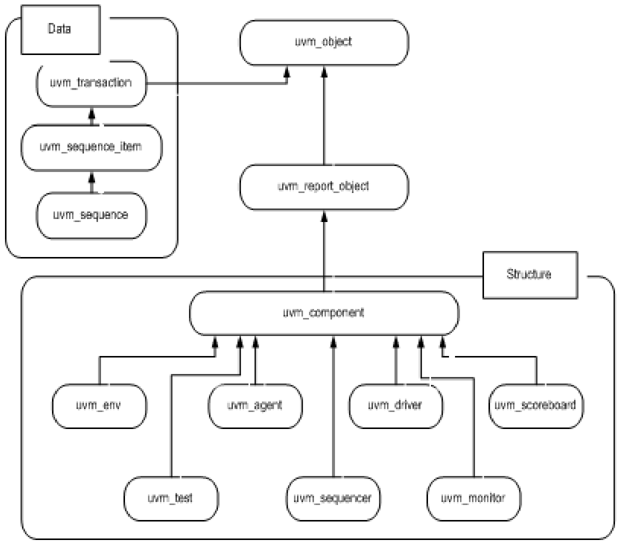

Figure 1. UVM class library.

The UVM verification environment comprises of many objects and components. Objects are Transaction, Sequence_item and Sequence. Components are Environment, Test case, Sequencer, Monitor, Driver, Agent and Scoreboard[7].

2.1.1Agent

It has three components which are Sequencer, Driver and Monitor.

2.1.2 Sequence item

In this the transactions are created. The variables, constraints for sequences are declared in this class.

2.1.3 Sequencer

It arranges the sequence items sequentially in particular order.

2.1.4 Driver

The drivers convert the packet data inside the sequence items into transactions i.e. pin level. These pin level transactions are sent to the DUT through the interface.

2.1.5 Monitor

Through the same interface it takes the transactions and observes pin level activities. These pin level transactions are observed and these observations are converted into sequence items. These sequence items are sent to the components like scoreboard through analysis ports for analysing the test cases.

2.1.6 Environment

It has the collection of all blocks agents, scoreboards, config.

2.1.7 Test Case

In this all test case sequences are called. Each test case checks for each feature in the design or works according to the user defined code.

2.2 UVM Phases

For consistent order of test cases execution the UVM uses these phases during the simulation. The important phases of the UVM are:

2.2.1 Build Phase

This is the first phase of the simulation. When the test cases are simulated this phase is executed. In the build phase, test case component hierarchy are constructed through top to down. During this phase, components are constructed by the factory. In the Environment, components like agents and scoreboard[8] are built and similarly in the Agent, components like sequencer, driver and monitor are built. This is declared as "function void build phase" which means all the components are built in zero simulation time.

2.2.2 Connect Phase

After execution of build phase, this phase starts for connection of components. In this phase, various components of the class are connected. It follows the order as agent is connected to driver and the driver to the sequencer. The monitor is connected to analysis ports. This also is declared as "function void connect phase" which means all the components are connected within zero simulation time.

2.2.3 Run Phase:

After the connection of the components, the run phase is executed. This is the main phase for executing test cases. In this the simulation code starts. This is declared as "task run phase" which means it takes some simulation time.

2.2.4 Report Phase

From the monitor component, the data is collected through the analysis ports for analysing and reporting purpose. This is useful for further processing the data for storing in database. The following Figure 2 shows the partial phases.

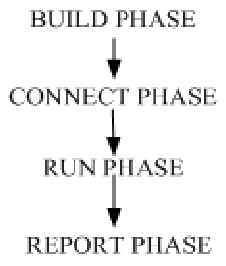

Figure 2. Partial phases.

2.3 UVM Macros

The UVM macros are useful for many classes, variables. With these macros, reporting, registering in factory and some more can be done. Few of them are listed below:

2.3.1 uvm_object_utils

This macro is used with classes derived from uvm_object. This also registers the new class in factory.

2.3.2 uvm_component_utils

This macro registers the new class type in the factory. It is usually used when deriving new classes from uvm_component.

2.3.3 uvm_field_int

This macro is used for registering a transaction variables in the UVM factory and implements operations.

2.3.4 uvm_info

This is a very useful macro to print messages from the UVM environment during simulation time.

2.3.5 uvm_error

This macro is responsible for sending messages with an error tag to the output log. This is immune to verbosity. This can be controlled by a message ID or severity.

2.3.6 uvm_fatal

This macro is responsible for stopping simulations. This can be controlled by a message ID or severity.

2.3.7 uvm_warning

This macro is responsible for displaying warning messages. This is set by verbosity from the environment.e, there are many more phases.

3. Design of Slave Block in Ethernet Management Interface

3.1 Slave Block

The Slave Block in the Ethernet Management Interface entity translates the Management Data Input and Output/ Management Data Clock transactions to access the Registers in the Physical device. There are 31 Physical devices in each Physical device there are 32 registers and any register can be accessed. While translating the Management Data Input and Output/Management Data Clock transactions if the Physical (PHY) address of the transaction matches with the device own address (CFG_ PHY_ADDR) then only the Physical device responds. The ratio of the MDC period to "CLK" period is assumed to be greater than or equal to 4.

The Master can send any type of transactions i.e. with preamble or without, the Slave Block should capable of understanding MDIO/MDC transactions with or without the preamble from Master. The Figure 3 shows the slave block and Figure 4 shows the host block.

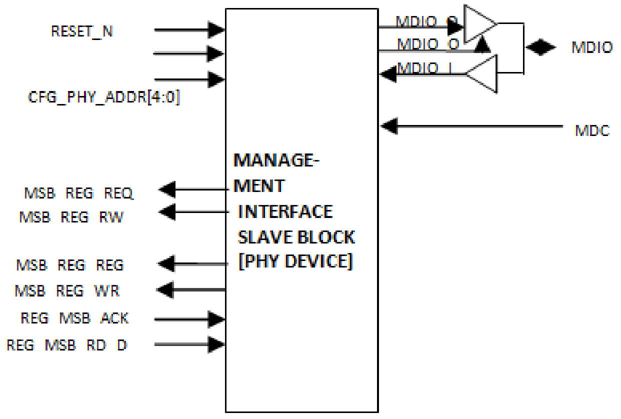

Figure 3. Slave Block.

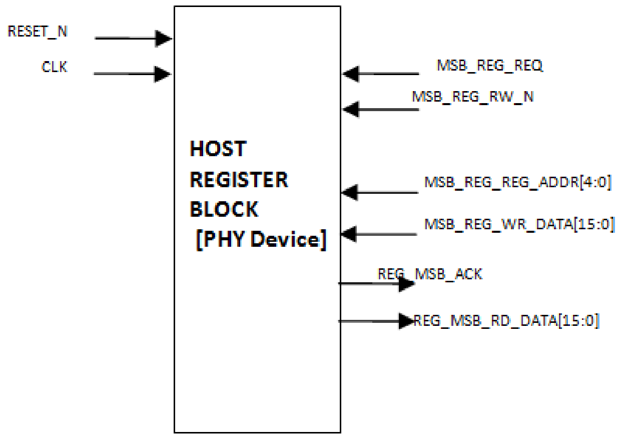

Figure 4. Host block.

3.2 Host Block

The Host Register block consists of 32 registers. Each register width is 16-bit which can be read and write by the Slave Block. Once MSB_REG_REQ is asserted in a clock cycle then the Host register block acknowledges and completes the accesses by asserting REG_MSB_ACK for one clock cycle at cycle no. N (where N is the cycle time of the read/write transactions) and returns read data along with acknowledge for read access and writes data into the corresponding register for write access.

4. Verification Methodology

The Verification plan for a Slave Block in Ethernet

Management Interface is described in this section. The driver component will act as the Master for the Slave Block.

4.1 Verification Plan

This defines what all features in the design have to be tested and metric coverage for that feature. It has been divided as Features, Input generation, Checking and Coverage scenarios. The completion of the verification plan depends on these parts. If all are covered then Verification is successfully completed.

4.1.1 Features

This list gives the complete features of the design to be verified. Check whether transaction is with preamble or without preamble. Single write and read transactions. Multiple write and read transactions. Response when read transaction is done. Depth of the host data. Generation of Acknowledgements from the host.

4.1.2 Input Generation

Serially send 64 bits of transaction through MDIO with preamble. Serially send 32 bits of transaction through MDIO without preamble. Randomize Reg addresses and Phy addresses with constraints. Randomize Data with Reg address and Phy address as constraint. Randomize Read and write operations. Corrupt the serial transmission of 64 bits transaction and similarly for 32 bits i.e. without preamble.

4.1.3 Checker

Use a memory in Monitor where it stores while writing the data. When reading it compares the data of corresponding address.

4.1.4 Coverage

Appropriate Cross groups, Cross points, Bins are built for the functional coverage of the design.

4.2 Verification Environment

This verification environment gives the purpose of test bench and working of objects and components. The environment for the Slave Block mainly comprises of objects and components. Objects are Transaction, Sequence_item and Sequence. Components are Environment, Test case, Sequencer, Monitor, Driver, Agent and Scoreboard. The agent comprises of drivers, sequence, monitor. The verification environment is shown in the Figure 5.

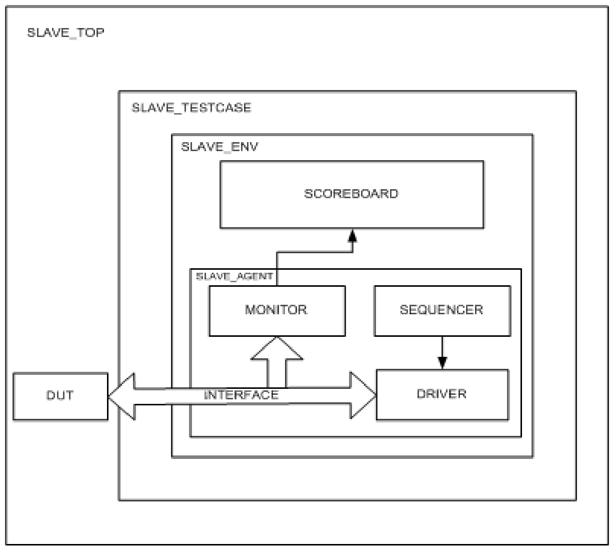

Figure 5. Verification enivironment.

Each one of the objects and components individually are described below:

4.2.1 Transaction

This class is extended from the class library "uvm_sequence_item". The user defined class is registered in the factory. In this class, random variables are created and other variables for monitor. In this constraints are also added for sequences and have post_randomize methods.

4.2.2 Sequence

This class is extended from the class library "uvm_sequence". The user defined class is registered in the factory. This generates series of transaction which in turn are routed to the sequencer.

4.2.3 Sequencer

This class is extended from the class library "uvm_sequencer". The user defined class is registered in the factory. This class controls the transaction flow from the sequences. This routed sequences are sent to driver.

4.2.4 Driver

This class is extended from the class library "uvm_driver". The user defined class is registered in the factory. The routed sequences are aligned in proper order through the sequencer. This driver class directly takes the sequences from the sequencer through port. These sequences are interfaced to the DUT. Then the output from the DUT is monitored through same interface. The driver class is built in the agent with build phase and in the environment sequencer and driver is connected in the connect phase.

4.2.5 Monitor

This class is extended from the class library "uvm_monitor". The user defined class is registered in the factory. This samples the signals from the interface. One monitor for displaying the transactions. It collects the transactions and displays accordingly. The other monitor is used for checking the transactions and for functional coverage. For checking a memory element is created to store the received data. It checks with the memory and read data from the DUT and displays error messages if any error occurs. Functional coverage is achieved through creation of cover groups, cover points and bins for the variables. In a simple meaning the bin checks whether it is filled or not. If it is filled then it is covered.

4.2.6 Agent

This class is extended from the class library "uvm_agent". The user defined class is registered in the factory. In this sequencer, driver, monitor are build in the build phase. In the connect phase of the agent class, driver is connected to sequencer through port and exports.

4.2.7 Environment

This class is extended from the class library "uvm_env". The user defined class is registered in the factory. This class builds the verification environment. It builds agents, scoreboard. In the build phase agent, scoreboard is created. In the connect phase the agent is connected to the scoreboard with the ports.

4.2.8 Test Cases

This class is extended from the class library "uvm_test". The user defined class is registered in the factory. In this class, sequence is randomized through the randomize function. The sequence is started here through environment, agent and sequencer.

5. Results

The verification environment is built accordingly using UVM. Different combinations of test cases are generated. All these test cases are with preamble and without preamble. Few of the test cases are constraint randomized so that the constraints are added. All the test cases reports are merged to one single report. The following table shows the functional and code coverage metrics for the DUT. The simulation result of simple read and write transactions is shown in Figure 6. From the above Table 1, code coverage did not reach 100% as some of the variables in the design are fixed at one value which means toggling did not happen for the few variables. The functional coverage achieved is 94.44% covering the features.

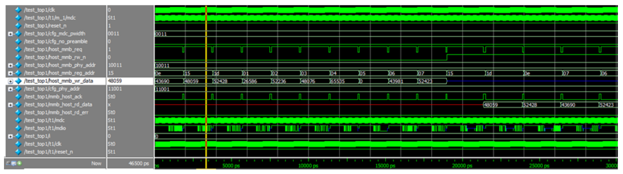

Figure 6. Simple read and write transactions.

Table 1. Coverage metrics

Coverage	Achieved
Code Coverage	97.96%
Functional Coverage	94.44%

6. Conclusion

The main objective of this project was to design the Slave Block in Ethernet Management Interface entity and develop the Verification Environment using UVM Methodology. As in the previous chapters the design part and verification of the design are explained. Through conventional test bench the design was checked. Then through the Verification Environment, the total design is verified.

7. References

1. Mentor Graphics, Verification Academy. UVM Cookbook. Mentor Graphics; 2012. p. 1-569.
2. Spears C. System Verilog for Verification: A guide to learning the testbench language features. 2nd ed. Springer: Business Media LLC; 2007.
3. Accellera Universal Verification Methodology (UVM) 1.1 User's Guide. Accellera; 2011. p. 1–300.
4. Bergeron J. Writing testbenches using System Verilog. USA; Springer Business Media LLC: 2006.

5. Yun YN, Kim JB, Kim ND, Min B. Beyond UVM for practical Soc verification. International SoC Design Conference (ISOCC); Jeju. 2011 Nov 17-18. p. 158–62.
6. Siddharth R, Singh V. Review on UVM concepts for functional verification. International Journal of Electrical, Electrical and Data Communications. 2014 Mar; 2(3):101–7.
7. Priyadharson ASM, Joshua SV, Thilip Kumar C. PLC – HMI and Ethernet based monitoring and control of mimo system in a petrochemical industry. Indian Journal of Science and Technology, 2015 Oct; 8(27):1–5.
8. Rosenberg S, Meade KA. A practical guide to adopting the Universal Verification Methodology (UVM). San Jose CA, USA: Cadence Design Systems Inc; 2010.