The Report Committee for Ling Lin
Certifies that this is the approved version of the following Report:

Applying Universal Verification Methodology on a Universal Asynchronous Receiver/ Transmitter Design

APPROVED BY
SUPERVISING COMMITTEE:

Jacob A. Abraham, Supervisor

Mark McDermott

Applying Universal Verification Methodology on a Universal Asynchronous Receiver/ Transmitter Design

by

Ling Lin

Report

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ENGINEERING

The University of Texas at Austin
May 2020

Dedicated to my family and friends.

Acknowledgments

First and foremost, applauds and thanks to the God, for his showers of blessings throughout my research path to complete the report successfully.

I would like to express the greatest appreciation and sincere gratitude to my committee chair, Professor Jacob A. Abraham, for giving me the opportunity to do research and providing invaluable assistance. His sincerity, dynamism and vision have profoundly inspired me. He has the mindset and the essence of a genius: he constantly and convincingly conveyed a spirit of adventure regarding the education and research domain, and an enthusiasm regarding teaching. Without his support and guidance, this report would not have been possible.

I want to thank my report reader, Dr. Mark McDermott, whose work demonstrated to me that concerns for VLSI and digital verification field. Also, I greatly appreciate him for taking out his precious time to be my report reader which means a lot to me.

Lastly, I am extremely grateful to my parents for their love, caring and sacrifices they made to prepare me for my future. I am very much thankful to my parents for their understanding, love, and continuing encouragement during my research work. Also, I express my thanks to my sister for her support and precious prayers. My special thanks go to my ICS track friends for the enthusiastic interest shown to complete my report successfully.

Abstract

Applying Universal Verification Methodology on a Universal Asynchronous Receiver/ Transmitter Design

Ling Lin, M.S.E.
The University of Texas at Austin, 2020

Supervisor: Jacob A. Abraham

Universal asynchronous receiver/ transmitter (UART) is a computer hardware device that is used to transform user data to parallel and serial forms. The UART protocol is typically integrated in an Integrated Circuit (IC) which is used for serial communication over a computer or a peripheral device. The UART protocol accepts the input data as bytes and transmits the single bits in a sequential way. Usually, there will one or more UART peripherals which are integrated in microcontrollers and related devices which can support synchronous operation. The UART communication method may be done in three different ways: simplex (in one direction only), full duplex (both devices can receive and send data simultaneously) and half duplex (both devices can take turns to receive and transmit data). The structure of the data frame contains a single start bit, followed by next five to nine bits, depends on the code set employed. If there is a parity bit, then it will be placed after all the data bits. The next one or two bits are always logic high to indicate the stop bit(s). This technical report will demonstrate a verification methodology to verify WISHBONE UART IP Core using the Universal Verification Methodology (UVM) process.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

1.1 Verification Methodologies

There is a critical need for practical and implementable solutions to the problem of verifying large and complicated digital designs. These designs require a creative mix of verifications techniques, where the mix changes depending on the design implementation. As we know, functional verification requires a significant amount of effort and time for the overall design and verification lifetime. With the assistance of many techniques and tools, verification methodologies can enable efficient execution towards a bug free design where verification methodologies include the Universal Verification Methodology (UVM), Formal Verification, Dynamic Simulation, Assertion based Verification, Power Aware Simulations, and Coverage methodologies.

1.2 UVM Benefits

UVM aims to improve design efficiency by making it easier to verify the design modules with a standardized representation that can be used with many verification implementations. There are few advantages associated with utilizing UVM as the verification tool [1].

- The UVM methodology is built as many modular components, such as agents, driver, sequencer, etc. This empowers the reuse of these components across unit module or multiple modules as well as across various verification projects.
- Configuration mechanisms in UVM are higher level components in a

hierarchy that can access the lower level component variables which enables the deep hierarchy. For example, the configuration helps access various testbench components based on the verification environment around it, and without thinking about how deep the component is in the testbench hierarchy. Without such a configuration mechanism, the users must access the lower level component variables by using hierarchical paths, which makes reusability difficult.

- The Factory mechanism in UVM aims to enhance scalability and flexibility of the testbench by allowing the users to replace an existing class object by the inherited child class object. In short, the factory mechanism helps them to be overridden in various environments or tests without changing the underlying base code.
- The stimulus is kept separate from the actual testbench hierarchy and the stimulus can be reused across different projects or units due to its reusability property.
- The UVM methodology and base library are supported by any specific simulators because there is no independence on any simulators.
- The stimulus generation has good control over generating various sequences with its randomization on the testbench, virtual sequences, layered sequences, and so on.

1.3 Verification Challenges in Modern Processors

The increasing intricacy of hardware and software designs, especially modern processors, have been posing various challenges in verification. The industry has been putting more and more efforts and money into verification than design, but still, there are examples of numerous prototypes with innovative features that have been withdrawn from projects due to

the inability to come up with a solution to completely validate the design functionality within a reasonable time frame.

Even though years and years of research have been done in the field of modern processors, the exponential growth in the complexity of modern processor design over past few years has continued to bring gigantic challenges into the field of verification. There are various optimizations to improve the performance or lower the power consumption for these modern processors such as branch prediction, pipelining, out-of-order execution, instruction set architecture, and so on. All these optimizations and implementation varieties have created extra corner cases for the existing design, which requires additional time and human efforts to ensure the verification for the design is fully covered, and to guarantee the safety and correctness of the design.

The goal is to verify the correctness of each module in the design and make sure all the modules can communicate with each other properly. Sometimes, the verification plan is relatively similar from one generation to another generation. Therefore, it is beneficial if we can reuse the verification plans and codes. This is why UVM was invented and developed.

With growing implementation of UVM methodology in the verification industry, it should be obvious that the advantages of UVM outweighs any drawbacks. However, there are few drawbacks for it. For example, UVM methodology is still under development, and has many cases of overheads that can sometimes cause simulation to appear slow or probably could even cause bugs in the design. Also, for anyone new to the UVM methodology, the learning curve to understand all the implementation details and the library could be challenging and time-consuming; especially, hardware

engineers must learn object-oriented programming (OOP) since the UVM methodology is OOP based. In short, UVM is useful but extremely complicated which requires several experts in the field to write UVM verification components and testbenches in order to proliferate to team.

There are few things which could potentially slow down the verification process.

- Either the design specification is incomplete during the verification, or the design under test is not fully available which would extend the process time.
- It requires re-use of the previous code but cannot assess the previous code reusability due to the limited resources.
- It requires more verification engineers than design engineers due to the complexity of the design. Usually, the verification code is 3~5 times larger than the design code. Thus, it needs more time and effort on debugging.
- It is complicated to master all the details of UVM and the learning curve for the UVM methodology is steep. So, only UVM experts are good at it.
- It is difficult for designers to grant assistance to the team because they are probably not familiar with methodology and language.
- Understanding another person’s code is a challenging task.
- Lastly, the verification document never stays synchronized with the code because it requires constant updates which cab causes mismatches between the design and verification documents.

These issues described above are based on reality and will occur in every design, which can slow down the verification process. Therefore, it is important to keep in mind that the verification

process will take longer time than one thinks it should take.

1.4 UART Verification using UVM: Motivation and Related Work

The WISHBONE compatible Universal Asynchronous Receiver/Transmitter (UART) peripheral delivers an interface between the WISHBONE system bus (described in the next Chapter) and an external communication channel. This UART contains a receiver and a transmitter pair which is used to convert the outgoing and incoming data into the serial binary data stream. As we know, UART is one of the most common and easily used serial communication techniques. Today, UART is being used in many electronic applications like RFID based applications, Bluetooth modules, global system for mobile communications, and so on. The computers we have, and internal modems contain UART modules as well in order to manage the serial ports. Therefore, it is essential to ensure the functional correctness for UART modules in the design. The WISHBONE peripheral was studied during my graduate coursework, which sparked my interest to do verification for the UART module inside the WISHBONE peripheral.

1.5 Summary and Related Work

This technical report studied the Universal Verification Methodology for the WISHBONE UART protocol. The UART provides serial communication capabilities, which allow communication with external systems. The key feature of the design is the WISHBONE INTERFACE with multiple bit selectable data bus modes. The WISHBONE UART protocol is verified using the

UVM methodology. The test bench is written with constrained randomization in order to obtain the maximum functional coverage.

Chapter 2

UART Protocol

This chapter elaborates the details of the WISHBONE protocol and gives the details of the UVM methodology in the protocol in order to verify the functional correctness.

2.1 UART Definition

Universal asynchronous receiver/ transmitter (UART) is a hardware computer module used for converting user data into parallel and serial types. An UART protocol is typically integrated in an Integrated Circuit (IC) which is used for serial communication over a computer bus or to/from a peripheral device. The UART protocol accepts the input data as a byte and transmits the single bits in a sequential fashion. Usually, there will one or more UART peripherals which are integrated in microcontrollers, and the related devices can support synchronous operation. The UART communication method may be done in three ways: simplex (in one direction only), full duplex (both devices can receive and send data simultaneously) and half duplex (both devices can take turns to receive and transmit data). The structure of data framing contains a single start bit, followed by next five to nine bits, depends on the code set employed. If there is a parity bit, then it will be placed after all the data bits. The next one or two bits are always logic high to indicate the stop bit(s).

2.2 WISHBONE Definition

The WISHBONE System-On-Chip (SOC) interconnect architecture for IP cores is an open source hardware computer bus which aims to let the integrated circuit parts communicate with each other, enabling the connection of various cores to each other within a chip. Currently, the WISHBONE bus protocol is used in many designs to create the logical interface between IP cores in both academic and industry projects. The WISHBONE bus interconnect itself is not an IP core, it is an interface to all cores that require interfacing to other cores inside an FPGA or ASIC chip. The WISHBONE protocol allows the design engineer to combine multiple hardware core modules together written in VHDL, Verilog, or some other hardware description language. The WISHBONE is intended as a logical bus which does not specify any electrical information. Instead, the WISHBONE specification is expressed as logical signals along with high and low voltage signals. In this report, it showcases on how to verify the functionality of the WISHBONE SoC UART interconnection interface using UVM and express the functional coverage rate in ModelSim.

2.3 UART Registers

The UART protocol contains a receiver and a transmitter. The receiver converts a serial data stream to a parallel data stream on the asynchronous data frame received from external devices. The transmitter performs parallel-to-serial conversion on the multiple bit data received from the CPU. The following is the description of the registers used in WISHBONE UART protocol [2].

UART Register	Value
AMBER_UART_PID0	16’h0fe0
AMBER_UART_PID1	16’h0fe4
AMBER_UART_PID2	16’h0fe8
AMBER_UART_PID3	16’h0fec
AMBER_UART_CID0	16’h0ff0
AMBER_UART_CID1	16’h0ff4
AMBER_UART_CID2	16’h0ff8
AMBER_UART_CID3	16’h0ffc
AMBER_UART_DR	16’h0000
AMBER_UART_RSR	16’h0004
AMBER_UART_LCRH	16’h0008
AMBER_UART_LCRM	16’h000c
AMBER_UART_LCRL	16’h0010
AMBER_UART_CR	16’h0014
AMBER_UART_FR	16’h0018
AMBER_UART_IIR	16’h001c
AMBER_UART_ICR	16’h001c

Figure 2.1: Registers used in UART

- Peripheral identification register (AMBER_UART_PIDX):

The peripheral identification register provides standard information required for all UART components and the configuration of the UART peripheral. There are four different peripheral identification registers listed in the UART module.

- Component identification register (AMBER_UART_CIDX):

The component identification registers are eight-bit wide registers, that can theoretically be treated as a single register that holds a 32-bit component identification

value. There are four different component identification registers listed in the UART module.

- Data register (AMBER_UART_DR):

The data register is used to contain a second operand for the operations, and it can be assigned to a variety of functions by the users. It can be utilized with any computer instruction to perform operations on data level computation.

- Receiver Shift Register (AMBER_UART_SRS):

In most essential form, the shift register is a bidirectional first-in-first-out circuit. It is commonly used in digital converters where it can translate serial data to parallel data, or vice-versa.

- Line Control Register High_Byte (AMBER_UART_LCRH):

The AMBER_UART_LCRH register is the line control register. This register can access high bytes of the UART line control register.

- Line Control Register Medium_Byte (AMBER_UART_LCRM):

The AMBER_UART_LCRM register is the line control register. This register can access medium bytes of the UART line control register.

- Line Control Register Low_Byte (AMBER_UART_LCRL):

The AMBER_UART_LCRL register is the line control register. This register can access low bytes of the UART line control register.

- Line Control Register (AMBER_UART_LCRX):

The user programmer has the control over regulating the format of the asynchronous data stream exchange by using its line control register. It is an 8-bit wide register. Additionally, the user can modify the internal content of the line control register

which excludes the need for separate storage capacity of the line features in the overall system.

- Control Register (AMBER_UART_CR):

The control register is used to provide configuration and control for memory management unit, interrupts, and location for exception vectors. It is a 32-bit write/read register which is only accessible in privileged modes.

- First In, First Out (FIFO) Register (AMBER_UART_FR):

The FIFO register is a write-only 8-bit register at the same address as the interrupt identification register which has read-only access. The internal UART FIFOs can be enabled and cleared by the FIFO register and it can choose the receiver FIFO activate level.

- Interrupt Identification Register (IIR) (AMBER_UART_IIR):

The interrupt identification register is a read-only access register at the same address as the FIFO register which has write-only access. The IIR shows that an interrupt is expected when an interrupt is enabled in the Interrupt Enable Register. There is an on-chip interrupt generation and prioritization which permits flexible data exchange with the central processing unit. The UART offers three different interrupt prioritizations: receive line state which has the highest priority (priority 1), receiver timeout or receiver data ready (priority 2), and transmitter holding register empty (priority 3).

- Interrupt control register (AMBER_UART_ICR):

The interrupt control register is a hardware register which used to configure the chips to generate interrupts in order to raise a signal on an interrupt line in response to some event occurring within the chip. An interrupt control is used in various microprocessor

controllers to generate interrupt signals which tells the central processing unit to stop its current task and start executing another set of predefined instructions.

Chapter 3

UVM Structure Guide

3.1 Challenges of Verifying Systems using UVM

Today, typical microprocessor development from scratch is a challenging task which could many years to complete, and the challenges are numerous during the development. It requires parallel developments across multiple teams, and it takes large amount of human effort to verify the functional correctness of a microprocessor. The classic way is to partition the whole CPU into smaller modules and each team or multiple teams are responsible for verifying that module. Therefore, reuse of previous code becomes an essential key to avoid duplication of work, and it is critical to have the ability to integrate an exterior IP block as well as the checkers and test stimuli can be reused at a higher hierarchy level. Verifying the complex systems requires meticulous verification planning, code structure, and fault-tolerant development. The standardization turns into a key consideration during the development. Fortunately, the UVM methodology can help resolve those problems.

3.2 Why Use UVM Methodology?

UVM stands for Universal Verification Methodology, and it supports a framework for layered and modular components in verification. It enables

configuration of components to be used in a variety of modules, and each functional definition for the components is clear and understandable [5]. The UVM methodology is released and maintained by the Accellera organization and all the source code is available online [3]. It is a mature methodology and there are signification amounts of resources and support available.

3.3 Key Components of a UVM Testbench

The verification components in UVM is derived from the uvm_component class which has features like hierarchy searching, configuration, factory, various phasing, transaction recording and reporting [3]. The following are a few of the uvm_component classes.

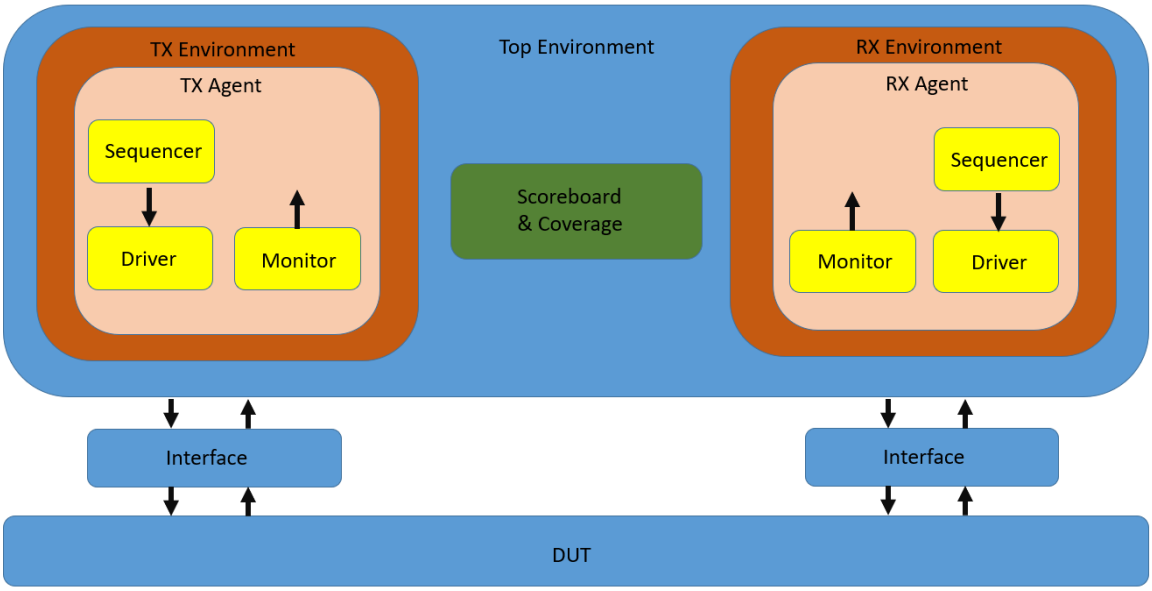

Figure 3.1: UVM Testbench Structure

- Agent:

An agent is an encapsulation of a monitor, a sequencer, and a driver. A user-defined agent is extended from uvm_agent where is inherited by uvm_component. Because of

the structure of agent, the testbench of UVM can be re-usable. An UVM structure can contain one or more agents depending on the environment setting. An agent can be configured as an active agent or passive agent by using a set_config method; the default agent will be an active agent. If the agent is in active mode, then the agent will have the monitor(s), sequencer(s), and driver(s). But if the agent is passive mode, then the agent will only have a monitor.

- Environment

The environment is the most important component of UVM testbench and the object of this class is created and instantiated in test. The environment contains components like virtual sequencer, agents, scoreboard and so on. A UVM environment contains various reusable verification modules and defines their default configurations which are required by the verification products. The environment has the connection between scoreboard and monitors. If there is a virtual sequencer in the environment, we need to create an instance of the virtual sequencer and connect physical sequencers with their handles in the virtual sequencer.

- Interface

SystemVerilog includes the interface construct which encapsulates the communication between various blocks. An interface is a package of signals or nets through which a testbench can communicate with the design. We can view virtual interface as a handle pointing to the interface instance which aims to capture details of the communication between the blocks.

- Scoreboard

The scoreboard in UVM is a verification component that has checkers that verify the functional correctness of a design. It usually receives transaction level objects which obtained from the interfaces of a design under test via transaction level modeling analysis ports. It compares the data sent from one side to the other side. For instance,

it compares two results gathered from the write monitor and read monitor as well as it sampling the signal values which are included in the cover groups in order to analyze the functional coverage.

- Sequence

In the sequence, the input driven data is randomized. The randomized input data is given to the driver through the sequencer. Hence, a sequence can be viewed as a pattern of sequence items which can be send to the driver for insertion into the design.

- Sequencer

The sequencer operates the request flow and give a response for sequence items between the driver and sequences by using a handshake protocol. The transaction level modeling interface is driven by the driver and sequencer in order to communicate transactions. uvm_driver and uvm_sequencer base classes contain seq_item_port and seq_item_port for transaction respectively.

- Test

We begin the sequences in the test. In the base test, we will set all the parameters of configuration database class based on the specification. We will set up the interface on the top and connect all the interface handles in the local base classes. At the same time, the base test can also initialize the environment. Additionally, all those tasks are completed during the build phase, and the child tests will be produced from this parent test class. The handle for virtual sequences is created inside the child test and will begin executing on virtual sequencer. The objection will be raised before starting the sequence and the objection will be dropped after starting the sequence for the knowledge purpose. If we do not raise the objection, the simulator will consider there does not exist a run phase to execute, therefore, simulator can bypass this phase and jump directly to the extract phase. Also, the entire number of raised objections should be matched to the number of dropped objections to keep objection consistency.

- Driver

The driver is an active entity and it mimics the logic that drives the design under test. It aims to fetch the data stream from the sequencer via the sequencer repeatedly. Then, the driver packages the data and transfers it to the design under test using the virtual interface. All driver classes should be extended from the uvm_driver parent class. In short, a driver is a component that transforms a sequence item or a transaction into a set of pin level toggling signals and delivers them to the DUT through the virtual interface.

- Monitor

A UVM monitor can sample the DUT signals, it is called a passive entity because it does not drive them. The monitor is responsible for obtaining signal activity from the interface and converting it into transaction level data objects which can be sent to other components. A virtual interface handle should connect to actual physical interface that the monitor aims to monitor, and transaction level modeling analysis port declarations should transmit the captured data to other needed components. The collected data can be used for coverage collection and protocol checking. High level functional checking should not be done within the monitor, in a scoreboard component.

- Design under Verification

The design is ready to be verified which is called as a design under verification (DUV). The DUV comes with a set of inputs and outputs which are going to be compared with the reference model in the scoreboard in order to ensure the functional correctness of the design. And functional coverage is used to measure how much of the design specification has been exercised.

- Subscriber

The subscribers receive the information from the analysis port. They endorse to a broadcaster and obtain objects when an item is endorsed via the analysis port. A uvm_component does not come with an analysis port, however a uvm_subscriber has an analysis implementation object which is called analysis_export. Therefore, we do not need to create additional implementation for our purpose.

Chapter 4

Experiments and Results

4.1 Application Guidelines

Before looking into the examples that we used for verification, we will first focus on the situations where the application of UART would be more advantageous and appropriate.

The UVM flow proposed in this technical report aims to largely improve the reusability and modularity of verification in large designs, where classic verification easily runs into the issues where development teams need to spend additional time writing documentation for their added code. The major advantage in the UVM flow is to utilize the configure mechanisms to simplify configuration of objects with deep hierarchy and apply factory mechanisms to make easier modification of UVM components separately. Another case, also for larger design verification, UVM introduces the concepts of transaction level modelling where is a methodology to model the design at a higher level of abstraction. Even though the actual physical interface to the design under test is embodied in actual signal level activity, the majority of verification tasks such as gathering coverage rates, producing stimulus, functional checking, and so on, are done at the transaction level by keeping them away from actual signal details. This is beneficial because it facilitates the reuse of those components and the code can be better maintained within or across various projects.

In this Chapter, experiments on a WISHBONE UART using UVM will be

conducted and the corresponding results and coverage will be evaluated to illustrate the functional correctness and coverage rates.

4.2 WISHBONE UART UVM Verification Setup

This technical report is designed as verification for WISHBONE UART IP Core using Universal Verification Methodology (UVM). UVM is a formidable standardized verification methodology that was architecture to be able to verify a broad range of verification designs. UVM is implemented as a SystemVerilog based class library. For this technical report, the reader will be exposed to a UVM testbench with half-duplex mode to verify a WISHBONE UART IP Core[4].

Here are the steps to get UVM verification running in ModelSim[3].

- Type module load mentor/modelsim/2016 in terminal

UVM verification for the receiver side:

- There is test.sv file inside "sim" folder, uncomment out line 22 where it says seq_rx.start and comment out line 23 where it says seq_tx.start.
- There is scoreboard.sv inside "sim" folder, uncomment out line 58, 59, 60 where it mentions fifo_in_1.get(tx_in_1), fifo_out_1.(tx_out_1) and compare_1(). And you should comment out line 62, 63, 64, 65.
- Type make in the "sim"folder in the terminal.

UVM verification for the transmitter side:

- There is test.sv file inside "sim" folder, uncomment out line 23 where it says seq_tx.start and comment out line 22 where it says seq_rx.start.
- There is scoreboard.sv inside "sim" folder, uncomment out line 62, 63, 64, 65 where mentions fifo_in_2.get(tx_in_2), plyload=tx_in_2.payload, fifo_out_2.get(tx_out_2), and compare_2(). And you should comment out line 58, 59, 60.
- Type make in the "sim"folder in the terminal.

4.3 Verification Results and Coverage

4.3.1 Verification Structure

In order to verify the WISHBONE UART protocol, it is better to build a reusable environment with modularity. During this research project, I explored the Amber25 Core UART protocol and exploited functional verification for its UART module. Also, I built a model of the design-under-test and observed how the model behaved with given stimulus input data steam. Lastly, I proved and disproved the correctness of proposed UART protocol based on the results alone with its coverage rates.

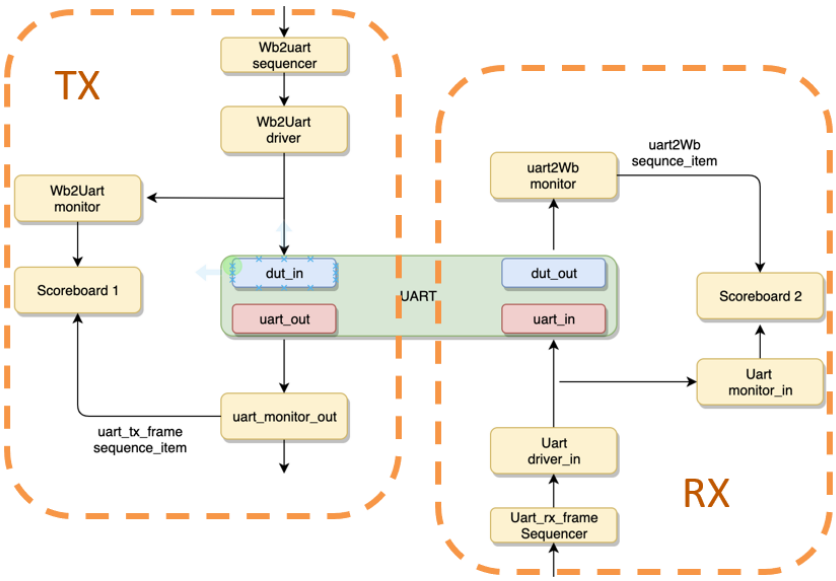

Figure 4.1: UART UVM Structure

Figure 4.3.1.1, shows the UVM structure used to verify the UART protocol inside the AMBER25 Core. There are transmitter side and receiver side for UART. Each side has its own sequence driver, sequencer, driver, and monitor.

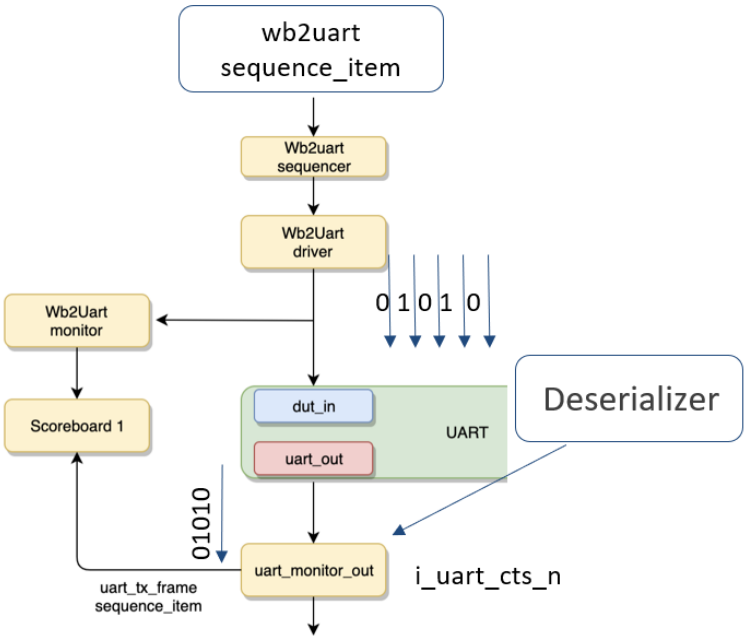

Figure 4.2: UART Transmitter

Figure 4.3.1.2, the stimulus generates random inputs (wb2uart sequence_item) and the wb2uart sequencer carries the sequence item and sends it to the wb2uart driver. Then the driver can translate the sequence item to the pin level signal protocol and send it to design under test through virtual interface. The scoreboard compares the results from the output of the design under test with the reference results to ensure its correctness. The input WISHBONE data (i_wb_dat) to the design under test is 32-bits and the WISHBONE UART output data in the transmitter side has 32 1-bit data items when one full packet is transmitted. There is a deserializer inside uart_monitor_out in order to transform the serial data to parallel 32-bits data.

The communication protocol between sequencer and driver is called a handshake. The UVM sequence-driver advanced programing language primarily

uses blocking methods on driver and sequence side for sending a sequence item to driver and receiving the response back from driver once it is acknowledged.

On the sequence side, there are two functions to achieve the handshake communication protocol as follows.

1) start_item(<item>): This method commands the sequencer to gain the access to the driver for the sequence item and returns when driver grants access to the sequencer.
2) Finish_item(<item>): This approach results in the driver obtaining the sequence item and is a blocking statement which returns only after driver sends back item_done() function.

On the driver side, there are two functions to achieve the handshake communication protocol as followed:

1) get_next_item(req): The driver asked a sequence item from the sequencer by using get_next_item method through the seq_item_port which is a transaction level modeling handle. Subsequently, the implementation of this port is declared in the sequencer, the function call makes the sequencer to pop out the sequence item from the internal FIFO and send it back to the driver. In driver, get_next_item is a blocking method that blocks until an item is received. This method returns the sequence item and the driver can translate the sequence item to the pin level signal protocol.
2) item_done(req): This is a non-blocking method and the driver can use this method to issue the completion of driver-sequencer handshake and it should be called after a try_next_item() method call or a get_next_item() method call.

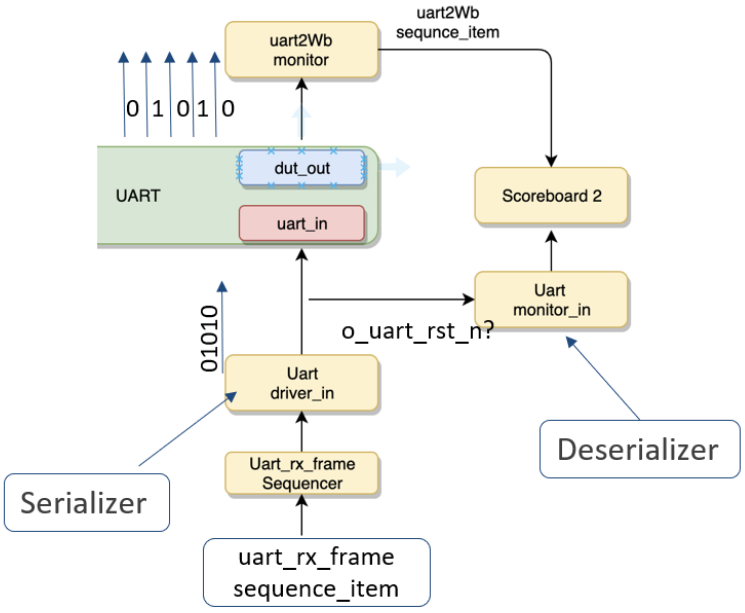

Figure 4.3: UART Receiver

Figure 4.3.1.3 shows how the stimulus generates random input (uart_rx_frame sequence_item) and uart_rx_frame sequencer carries the sequence item and sends it to uart_tx_frame driver. Then the driver can translate the sequence item to the pin level signal protocol and send it to design under test through the virtual interface. The scoreboard compares the results from the output of design under test with the reference results to ensure its correctness.

4.3.2 Verification Coverage

Coverage is a generic metric commonly used in verification in order to measure the productivity and completeness of the process. It is declared as the percentage of the verification process goals that have been met. Coverage metric forms a critical portion of measuring progress in constrained random testbenches and it aims to provide excellent response to the effectiveness of constrained random testbenches. The coverage tool aims to obtain the information during the simulation

procedure to produce a development report. We can utilize the report to check for coverage rates for bins we set it up. And we can modify the current test cases until we are satisfied with the coverage rates for the bins. Generally, there are two types of verification coverage metrics which are code coverage and functional coverage.

1) Code coverage: The code coverage is also called structural coverage and it is spontaneously created by the simulators. It measures the execution of the actual RTL code which is tested by given test cases in the simulation and it can provide a quantitative measure of the testing environment. Based on the various program exercise, code coverage can break down into the following types.
    a. Line coverage: Line coverage measures how many lines are observed during the tests and this is generally targeted to be 100% before the verification procedure sign off.
    b. Branch coverage: Branch coverage assesses conditions like case statement, if-else statement, and ternary operator statement and checks whether both true and false case are covered.
    c. Block coverage: A group of statement with while loop, for loop, case statement, if-else statement, and begin-end statement is called a block. Thus, block coverage provides the coverage measurement for those blocks during a simulation.
    d. Expression coverage: Expression coverage examines the right-hand side of an assignment in order to obtain all the possible cases that can occur and determine how well those cases are covered during a simulation.
    e. Conditional coverage: Conditional coverage is like Boolean expression coverage and conditional coverage counts coverage rates of the expression was true or false.
    f. FSM coverage: FSM coverage computes the numbers of states, transitions, or arcs in the given state machines that are covered during a simulation.

g. Toggle coverage: Toggle coverage reflects the changes in signal values, and it can help in detecting any unused signals or ports that do not change their values.

2) Functional coverage: Function coverage is a user-defined (not automatically inferred) verification metric to quantify how much of the design specification has been exercised during a simulation. It is an important metric for measuring whether some corner cases or any relevant design scenarios have been tested or evaluated during a test. It does not depend on the design code as it is employed based on design specification. Thus, the functional coverage is code that examines the execution of a test plan based on the design specification.

4.3.3 Test Cases with Half-Duplex Mode

Constrained random generation test cases with half-duplex mode enabled for both transmitter side and receiver side are verified successfully.

The design is done using Verilog hardware description language and verified using UVM structure with testbench. The coverage is completed using ModelSim tool virtualize the coverage rates.

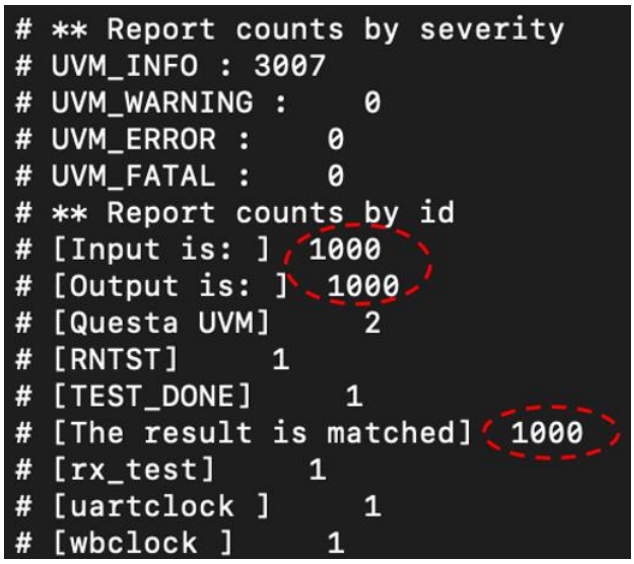

Figure 4.4: UVM Results Receiver Side

Figure 4.3.3.1, I randomly generated 1000 input streams for the receiver side and sent it to design under test (WISHBONE UART) to see whether it can successfully get those data streams transmitted. The experiment results shown that all input data streams matched the corresponding output result data stream.

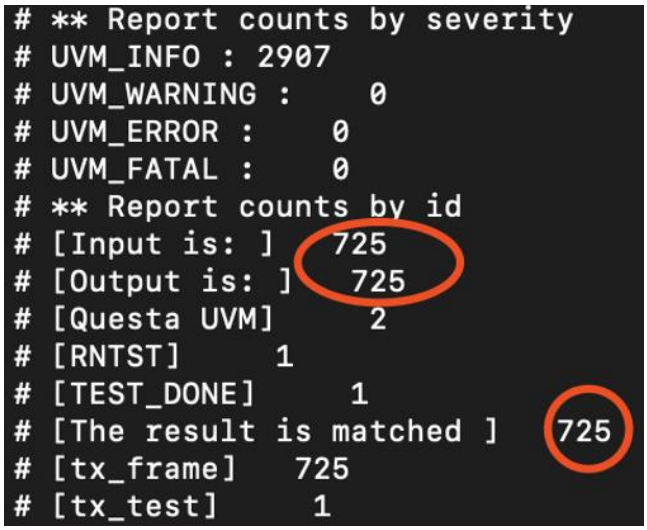

Figure 4.5: UVM Results Transmitter Side

Figure 4.3.3.2 shows I randomly generated 725 input streams for the transmitter side and sent to the design under test (WISHBONE UART) to see whether it can successfully get those data streams transmitted. The experiment results shown that all input data streams matched with output result data streams.

The constraining I set it up for the input data stream randomization is as shown in table 4.3.3.1.

Constraint Register	Value
i_wb_addr_lo	0
i_wb_addr_hi	0
i_wb_stb	dist{1:=80, 0:=15}
i_wb_we	dist{1:=80, 0:=15}

Table 4.1: Constraints for Input Data Stream

The cover points I set it up for wb2uart and uart_tx_frame module is as followed in table 4.3.3.2.

Cover point
i_wb_adr
i_wb_we
i_wb_stb
i_wb_dat
uart_addr
uart_we
uart_stb
tx_frame

Table 4.2: Cover point for Inputs and Outputs

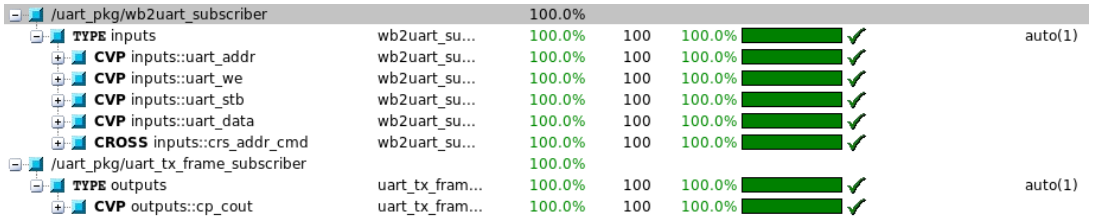

Figure 4.6: UART UVM Coverage

Figure 4.3.3.3 shows that all the cover points are covered. The functional coverage obtained is 100% for both wb2uart and uart_tx_frame module.

The functionality of WISHBINE UART IP Core has been verified successfully and various constrained randomization input data streams have been exercised. The functional coverage acquired for this verification plan of UART IP Core has reached to 100.

Chapter 5

Conclusions and Future Work

This technical report has proposed a verification for the WISHBONE UART IP Core to extend the capacity of UVM tools, which proves to be extremely effective because of its modularity and reusability. Although powerful, the verification methodology proposed in this technical report still has many aspects that could be enhanced. For example, a dynamic simulation can only boost the confidence regarding the design accuracy, and it would never achieve 100% completion. In contrast, formal verification can systematically prove or disprove the correctness of a design against specifications. Formal verification is a process where we can utilize mathematical modelling to verify a design to check whether it meets a requirement. In formal verification, all the states and input data are covered implicitly by the tool without the need for developing any expected output results or stimulus generators as UVM. A formal description of the specification in terms of elevated level model is needed by the tool for comprehensively covering all input combinations to disprove or prove functional precision with given the assumption statement, properties, and coverage range. Also, the assertions in formal verification can act as a tool to find a counterexample for any corner cases. If we find any counterexample, then we can trace back and find the potential bugs in the RTL code regarding to its state transition or input data. Therefore, I would like to explore formal verification for the WISHBONE UART IP Core using formal verification to see what the verification outcomes will be compared to the one using UVM verification.

Appendices

Appendix A

WISHBONE UART IP Core UVM Code

The following 18 different SystemVerilog files are inside a folder called "sim".

A.1 agent_in.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
typedef uvm_sequencer #(wb2uart) tx_sequencer_in;
typedef uvm_sequencer #(uart_rx_frame) rx_sequencer_in;

class agent_in extends uvm_agent;
    `uvm_component_utils(agent_in)

    uvm_analysis_port #(wb2uart) aport;
    uvm_analysis_port #(uart_rx_frame) bport;

    tx_sequencer_in wb2uart_sequencer_in_h;
    rx_sequencer_in rx_frame_sequencer_in_h;

    wb2uart_driver wb2uart_driver_h;
```

```
    wb2uart_monitor wb2uart_monitor_h;

    uart_driver_in uart_driver_in_h;
    uart_monitor_in uart_monitor_in_h;

    function new(string name, uvm_component parent);
        super.new(name,parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
        aport=new("aport",this);
        bport=new("bport",this);

        wb2uart_sequencer_in_h=tx_sequencer_in::type_id::create("wb2uart_sequencer_in_h",this);
        rx_frame_sequencer_in_h=rx_sequencer_in::type_id::create("rx_frame_sequencer_in_h",this);

        wb2uart_driver_h=wb2uart_driver::type_id::create("wb2uart_driver_h",this);
        wb2uart_monitor_h=wb2uart_monitor::type_id::create("wb2uart_monitor_h",this);

        uart_driver_in_h=uart_driver_in::type_id::create("uart_driver_in_h",this);
        uart_monitor_in_h=uart_monitor_in::type_id::create("uart_monitor_in_h",this);
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        wb2uart_driver_h.seq_item_port.connect(wb2uart_sequencer_in_h.seq_item_export);
        uart_driver_in_h.seq_item_port.connect(rx_frame_sequencer_in_h.seq_item_export);
        wb2uart_monitor_h.aport.connect(aport);
        uart_monitor_in_h.aport.connect(bport);
    endfunction: connect_phase

endclass: agent_in
```

A.2 agent_out.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
```

```
class agent_out extends uvm_agent;
   `uvm_component_utils(agent_out)

   uvm_analysis_port #(uart2wb) aport;
   uvm_analysis_port #(uart_tx_frame) bport;

   uart2wb_monitor uart2wb_monitor_h;
   uart_monitor_out uart_monitor_out_h;

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      aport=new("aport",this);
      bport=new("bport",this);
      uart2wb_monitor_h=uart2wb_monitor::type_id::create("uart2wb_monitor_h",this);
      uart_monitor_out_h=uart_monitor_out::type_id::create("uart_monitor_out_h",this);
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      uart_monitor_out_h.aport.connect(bport);
      uart2wb_monitor_h.aport.connect(aport);

   endfunction: connect_phase

endclass: agent_out
```

A.3 config.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class uart_dut_config extends uvm_object;
   `uvm_object_utils(uart_dut_config)
```

```
    virtual uart_in uart_vi_in;
    virtual uart_out uart_vi_out;

endclass: uart_dut_config


class amber_dut_config extends uvm_object;
    `uvm_object_utils(amber_dut_config)

    virtual dut_in dut_vi_in;
    virtual dut_out dut_vi_out;

endclass: amber_dut_config
```

A.4 cov.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;

class wb2uart_subscriber extends uvm_subscriber #(wb2uart);
    `uvm_component_utils(wb2uart_subscriber)

    logic [15:0]              i_wb_adr;
    logic       i_wb_we;
    logic       i_wb_stb;
    logic [31:0]    i_wb_dat;

    covergroup inputs;

                uart_addr: coverpoint i_wb_adr {
                    bins amber_uart_dr   = {AMBER_UART_DR};
                }

                uart_we: coverpoint i_wb_we {
                    bins one   = {1};
                }

                uart_stb: coverpoint i_wb_stb {
                    bins one   = {1};
                }
```

```
            uart_data: coverpoint i_wb_dat;

            crs_addr_cmd: cross uart_addr, uart_we, uart_stb;

    endgroup: inputs

    function new(string name, uvm_component parent);
        super.new(name,parent);
        inputs=new;
    endfunction: new

    function void write(wb2uart t);
        i_wb_adr = t.i_wb_addr_lo;
        i_wb_we  = t.i_wb_we;
        i_wb_stb = t.i_wb_stb;
        i_wb_dat = t.i_wb_dat;
        inputs.sample();

    endfunction: write

endclass: wb2uart_subscriber

class uart_tx_frame_subscriber extends uvm_subscriber #(uart_tx_frame);
    `uvm_component_utils(uart_tx_frame_subscriber)

    logic tx_frame;

    covergroup outputs;

    cp_cout : coverpoint tx_frame{
            bins zero = {0};
    }
    endgroup: outputs


function new(string name, uvm_component parent);
    super.new(name,parent);
    outputs=new;
endfunction: new

function void write(uart_tx_frame t);
    tx_frame = t.tx_frame;
    outputs.sample();
```

```
endfunction: write
endclass: uart_tx_frame_subscriber
```

A.5 env.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class env extends uvm_env;
   `uvm_component_utils(env)

   agent_in agent_in_h;
   agent_out agent_out_h;
   wb2uart_subscriber wb2uart_subscriber_h;
   uart_tx_frame_subscriber uart_tx_frame_subscriber_h;
   UART_scoreboard uart_scoreboard_h;

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      agent_in_h = agent_in::type_id::create("agent_in_h",this);
      agent_out_h = agent_out::type_id::create("agent_out_h",this);
      wb2uart_subscriber_h = wb2uart_subscriber::type_id::create("wb2uart_subscriber_h",this);

      uart_tx_frame_subscriber_h = uart_tx_frame_subscriber::type_id::create("uart_tx_frame_subscriber_h",this);
      uart_scoreboard_h = UART_scoreboard::type_id::create("uart_scoreboard_h",this);
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      agent_in_h.aport.connect(wb2uart_subscriber_h.analysis_export);

      agent_out_h.bport.connect(uart_tx_frame_subscriber_h.analysis_export);

      agent_in_h.aport.connect(uart_scoreboard_h.sb_in_1);
      agent_out_h.bport.connect(uart_scoreboard_h.sb_out_1);


      agent_in_h.bport.connect(uart_scoreboard_h.sb_in_2);
      agent_out_h.aport.connect(uart_scoreboard_h.sb_out_2);

   endfunction: connect_phase
```

```
    function void start_of_simulation_phase(uvm_phase phase);
        uvm_top.set_report_verbosity_level_hier(UVM_LOW);
    endfunction: start_of_simulation_phase

endclass: env
```

A.6 interfaces.sv

```
`include "uvm_macros.svh"

interface dut_in;
    logic                   i_clk;

    logic   [31:0]          i_wb_adr;
    logic       i_wb_we;
    logic   [31:0]          i_wb_dat;
    logic       i_wb_stb;

endinterface: dut_in

interface dut_out;
    logic       i_clk;
    logic [31:0]            o_wb_dat;
    logic       o_wb_ack;
    logic       o_wb_err;

endinterface: dut_out

interface uart_in;
    logic       i_uart_clk;
    logic       i_uart_rxd;
    logic      o_uart_rts_n;
endinterface: uart_in

interface uart_out;
    logic       i_uart_clk;
    logic       o_uart_txd;
    logic      i_uart_cts_n;

endinterface: uart_out
```

A.7 scoreboard.sv

```
`include "uvm_macros.svh"
import uvm_pkg::*;

class UART_scoreboard extends uvm_scoreboard;
   `uvm_component_utils(UART_scoreboard)

   uvm_analysis_export #(wb2uart) sb_in_1;
   uvm_analysis_export #(uart_tx_frame) sb_out_1;

   uvm_tlm_analysis_fifo #(wb2uart) fifo_in_1;
   uvm_tlm_analysis_fifo #(uart_tx_frame) fifo_out_1;

   wb2uart tx_in_1;
   uart_tx_frame tx_out_1;

   uvm_analysis_export #(uart_rx_frame) sb_in_2;
   uvm_analysis_export #(uart2wb) sb_out_2;

   uvm_tlm_analysis_fifo #(uart_rx_frame) fifo_in_2;
   uvm_tlm_analysis_fifo #(uart2wb) fifo_out_2;

   uart_rx_frame tx_in_2;
   uart2wb tx_out_2;

   logic [7:0] payload;

   function new(string name, uvm_component parent);
      super.new(name,parent);
      tx_in_1=new("tx_in_1");
      tx_out_1=new("tx_out_1");

      tx_in_2=new("tx_in_2");
      tx_out_2=new("tx_out_2");
   endfunction: new

   function void build_phase(uvm_phase phase);
      sb_in_1=new("sb_in_1",this);
      sb_out_1=new("sb_out_1",this);
      fifo_in_1=new("fifo_in_1",this);
```

```
        fifo_out_1=new("fifo_out_1",this);

        sb_in_2=new("sb_in_2",this);
        sb_out_2=new("sb_out_2",this);
        fifo_in_2=new("fifo_in_2",this);
        fifo_out_2=new("fifo_out_2",this);
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        sb_in_1.connect(fifo_in_1.analysis_export);
        sb_out_1.connect(fifo_out_1.analysis_export);

        sb_in_2.connect(fifo_in_2.analysis_export);
        sb_out_2.connect(fifo_out_2.analysis_export);
    endfunction: connect_phase

    task run();
        forever begin
            fifo_in_1.get(tx_in_1);
            fifo_out_1.get(tx_out_1);
            compare_1();

            //fifo_in_2.get(tx_in_2);
            //payload  = tx_in_2.payload;
            //fifo_out_2.get(tx_out_2);
            //compare_2();
        end
    endtask: run

    extern virtual function void compare_1;
    extern virtual function void compare_2;

endclass: UART_scoreboard

function void UART_scoreboard::compare_1;

    if (tx_in_1.i_wb_dat[7:0] == tx_out_1.tx_frame[8:1]) begin
        `uvm_info("Input is: ", tx_in_1.convert2string(), UVM_LOW);
        `uvm_info("Output is: ", tx_out_1.convert2string(), UVM_LOW);
        `uvm_info("The result is matched ", "\n", UVM_LOW);
    end
    else begin
        `uvm_info("Input is: ", tx_in_1.convert2string(), UVM_LOW);
        `uvm_info("Output is: ", tx_out_1.convert2string(), UVM_LOW);
```

```
      `uvm_info("The result is not matched!!!" ,"\n", UVM_LOW);
   end
endfunction

function void UART_scoreboard::compare_2;

   if (payload == tx_out_2.o_wb_dat[7:0]) begin
      `uvm_info("Input is: ", $sformatf("%b",payload), UVM_LOW);
      `uvm_info("Output is: ", tx_out_2.convert2string(), UVM_LOW);
      `uvm_info("The result is matched", "\n", UVM_LOW);
   end
   else begin
      `uvm_info("Input is: ", $sformatf("%b",payload), UVM_LOW);
      `uvm_info("Output is: ", tx_out_2.convert2string(), UVM_LOW);
      `uvm_info("The result is not matched!!!", "\n", UVM_LOW);
   end
endfunction
```

A.8 sequences.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
`include "../dut/system_config_defines.sv"
`include "../dut/register_addresses.sv"

   class wb2uart extends uvm_sequence_item;
            `uvm_object_utils(wb2uart);

      rand logic [15:0]   i_wb_addr_hi;
      rand logic [15:0]   i_wb_addr_lo;
               rand logic [31:0]      i_wb_dat;
      rand logic                      i_wb_we;
      rand logic                      i_wb_stb;

      constraint uart_addr {
         i_wb_addr_lo == 0;
         i_wb_addr_hi == 0;
               }
      constraint stb {
         i_wb_stb dist {1:= 80, 0 := 15};
      }
      constraint wb {
```

```
    i_wb_we dist {1:= 80, 0 := 15};
  }

  function new(string name = "");
    super.new(name);
  endfunction: new

  function string convert2string;
    convert2string={$sformatf("wb_dat: %b\n",i_wb_dat[7:0])};
  endfunction: convert2string

endclass: wb2uart


class uart_rx_frame extends uvm_sequence_item;

        rand bit start_bit;
        rand logic [7:0] payload;
        rand bit stop_bits;

        constraint default_start_bit        { start_bit == 1'b0;}
        constraint default_stop_bits        { stop_bits == 1'b1;}

        `uvm_object_utils_begin(uart_rx_frame)
          `uvm_field_int(start_bit, UVM_DEFAULT)
          `uvm_field_int(payload, UVM_DEFAULT)
          `uvm_field_int(stop_bits, UVM_DEFAULT)
        `uvm_object_utils_end

        function new(string name = "");
          super.new(name);
        endfunction

function string convert2string;
    convert2string={$sformatf("start_bit: %b, payload: %b\n",start_bit,payload)};
  endfunction: convert2string

endclass: uart_rx_frame

class uart2wb extends uvm_sequence_item;
        `uvm_object_utils(uart2wb);

          logic                 i_clk;
  logic [31:0] o_wb_dat;
```

```
    logic    o_wb_ack;
    logic    o_wb_err;

    function new(string name = "");
        super.new(name);
    endfunction: new;

    function string convert2string;
        convert2string={$sformatf("o_wb_ack: %b\no_wb_dat: %b",o_wb_ack,o_wb_dat)};
    endfunction: convert2string

endclass: uart2wb

class uart_tx_frame extends uvm_sequence_item;
            `uvm_object_utils(uart_tx_frame);

    logic[11:0]            tx_frame;

    function new(string name = "");
        super.new(name);
    endfunction: new;

    function string convert2string;
        convert2string=$sformatf("tx_frame: %b ",tx_frame[8:1]);
    endfunction: convert2string

endclass: uart_tx_frame

class simple_rx extends uvm_sequence #(uart_rx_frame);
    `uvm_object_utils(simple_rx)

    function new(string name = "");
        super.new(name);
    endfunction: new

    task body;
            uart_rx_frame rx;
            rx=uart_rx_frame::type_id::create("rx");
            start_item(rx);
            assert(rx.randomize());
            finish_item(rx);
    endtask: body

endclass: simple_rx
```

```
class simple_tx extends uvm_sequence #(wb2uart);
   `uvm_object_utils(simple_tx)

   function new(string name = "");
      super.new(name);
   endfunction: new

   task body;
         wb2uart tx;
         tx=wb2uart::type_id::create("tx");
         start_item(tx);
         tx.uart_addr.constraint_mode(1);
         assert(tx.randomize());
         finish_item(tx);
   endtask: body

endclass: simple_tx

   class simple_fifo_enable extends uvm_sequence #(wb2uart);
   `uvm_object_utils(simple_fifo_enable)

   function new(string name = "");
      super.new(name);
   endfunction: new

   task body;
         wb2uart tx;
         tx=wb2uart::type_id::create("tx");
         start_item(tx);
         tx.uart_addr.constraint_mode(0);
         assert(tx.randomize());
         finish_item(tx);
   endtask: body

endclass: simple_fifo_enable

class rx_seq extends uvm_sequence #(uart_rx_frame);
   `uvm_object_utils(rx_seq)
   `uvm_declare_p_sequencer(uvm_sequencer#(uart_rx_frame))

   function new(string name = "");
      super.new(name);
   endfunction: new
```

```
    task body;

            `uvm_info("rx_test", "\n--------------------------start------------------------------\n", UVM_LOW);
            repeat(10000)
            begin
      simple_rx seq;
      seq = simple_rx::type_id::create("seq");
      assert( seq.randomize());
      seq.start(p_sequencer);
            end
    endtask: body
  endclass: rx_seq


class tx_seq extends uvm_sequence #(wb2uart);
    `uvm_object_utils(tx_seq)
    `uvm_declare_p_sequencer(uvm_sequencer#(wb2uart))

    function new(string name = "");
      super.new(name);
    endfunction: new

    task body;

    `uvm_info("tx_test", "\n--------------------------start------------------------------\n", UVM_LOW);
    repeat(10000)
    begin
      simple_tx seq;
      seq = simple_tx::type_id::create("seq");
      assert( seq.randomize());
      seq.start(p_sequencer);
    end
    endtask: body
  endclass: tx_seq
```

A.9 tb.sv

```
`timescale 1ns / 100ps
`include "uvm_macros.svh"
import uart_pkg::*;
import uvm_pkg::*;
```

```
module dut(dut_in _in, dut_out _out, uart_in _uart_in, uart_out _uart_out);
uart uart_dut(
        .i_clk              (_in.i_clk),
        .i_wb_adr   (_in.i_wb_adr),
        .i_wb_we    (_in.i_wb_we),
        .o_wb_dat   (_out.o_wb_dat),
        .i_wb_dat   (_in.i_wb_dat),
        .i_wb_stb   (_in.i_wb_stb),
        .o_wb_ack   (_out.o_wb_ack),
        .o_wb_err   (_out.o_wb_err),
        .i_uart_cts_n       (_uart_out.i_uart_cts_n),
        .o_uart_txd  (_uart_out.o_uart_txd),
        .o_uart_rts_n       (_uart_in.o_uart_rts_n),
        .i_uart_rxd  (_uart_in.i_uart_rxd)

);
endmodule: dut

module top;
`define AMBER_UART_BAUD 921600
`define AMBER_CLK_DIVIDER 20
`ifndef Veritak
localparam real UART_BAUD       = `AMBER_UART_BAUD;
localparam real CLK_FREQ        = 800.0 / `AMBER_CLK_DIVIDER ;
localparam real UART_BIT_PERIOD   = 1000000000 / UART_BAUD;
localparam real UART_BIT_PERIOD_HALF   = UART_BIT_PERIOD / 2 - 5;
localparam real UART_WORD_PERIOD  = ( UART_BIT_PERIOD * 12 );
localparam real CLK_PERIOD      = 1000 / CLK_FREQ;
localparam real CLK_PERIOD_HALF   = CLK_PERIOD / 2;
localparam real CLKS_PER_WORD     = UART_WORD_PERIOD / CLK_PERIOD;
localparam real CLKS_PER_BIT      = CLKS_PER_WORD / 12;
localparam [9:0] TX_BITPULSE_COUNT        = CLKS_PER_BIT;
localparam [9:0] TX_CLKS_PER_WORD         = CLKS_PER_WORD;
`endif
localparam [9:0] TX_BITADJUST_COUNT       = TX_CLKS_PER_WORD - 11*TX_BITPULSE_COUNT;
localparam [9:0] RX_BITPULSE_COUNT        = TX_BITPULSE_COUNT-2;
localparam [9:0] RX_HALFPULSE_COUNT       = TX_BITPULSE_COUNT/2 - 4;

dut_in dut_in1();
dut_out dut_out1();
uart_in uart_in1();
uart_out uart_out1();
```

```
initial begin
   dut_in1.i_clk<=0;
   forever #CLK_PERIOD_HALF dut_in1.i_clk<=~dut_in1.i_clk;
end

initial begin
   dut_out1.i_clk<=0;
   forever #CLK_PERIOD_HALF dut_out1.i_clk<=~dut_out1.i_clk;
end

initial begin
   uart_in1.i_uart_clk<=0;
   forever #UART_BIT_PERIOD_HALF uart_in1.i_uart_clk<=~uart_in1.i_uart_clk;
end

initial begin
   uart_out1.i_uart_clk<=0;
   forever #UART_BIT_PERIOD_HALF uart_out1.i_uart_clk<=~uart_in1.i_uart_clk;
end

dut dut1(dut_in1,dut_out1,uart_in1,uart_out1);

initial begin
   dut1.uart_dut.tx_fifo_wp = 5'd0;
   dut1.uart_dut.wb_start_read_d1 = 1'd0;
end

initial begin
   uvm_config_db #(virtual dut_in)::set(null,"uvm_test_top","dut_vi_in",dut_in1);
   uvm_config_db #(virtual dut_out)::set(null,"uvm_test_top","dut_vi_out",dut_out1);
   uvm_config_db #(virtual uart_in)::set(null,"uvm_test_top","uart_vi_in",uart_in1);
   uvm_config_db #(virtual uart_out)::set(null,"uvm_test_top","uart_vi_out",uart_out1);
   uvm_top.finish_on_completion=1;
   uvm_report_info("wbclock ", $sformatf("wbclock = : %d\n",CLK_PERIOD_HALF * 2), UVM_LOW);
   uvm_report_info("uartclock ", $sformatf("uartclock = : %d\n",UART_BIT_PERIOD_HALF * 2), UVM_LOW);
   run_test("test1");
end

endmodule: top
```

A.10 tests.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;

class test1 extends uart_test;
   `uvm_component_utils(test1)

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   task run_phase(uvm_phase phase);

               tx_seq seq_tx;
               rx_seq seq_rx;

               seq_tx = tx_seq::type_id::create("seq_tx");
               seq_rx = rx_seq::type_id::create("seq_rx");

               phase.raise_objection(this);

               fork
               //seq_rx.start(env_h.agent_in_h.rx_frame_sequencer_in_h);
               seq_tx.start(env_h.agent_in_h.wb2uart_sequencer_in_h);
               join
               phase.drop_objection(this);
   endtask: run_phase
endclass: test1

class test2 extends uart_test;
   `uvm_component_utils(test2)

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   task run_phase(uvm_phase phase);

               rx_seq seq;
               seq = rx_seq::type_id::create("seq");
               assert( seq.randomize() );
               phase.raise_objection(this);

               seq.start(env_h.agent_in_h.rx_frame_sequencer_in_h);
```

```
            phase.drop_objection(this);
    endtask: run_phase
endclass: test2
```

A.11 uart_driver_in.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class uart_driver_in extends uvm_driver#(uart_rx_frame);

    `uvm_component_utils(uart_driver_in)

    uart_dut_config dut_config_0;
    virtual uart_in uart_vi_in;

    function new(string name, uvm_component parent);
        super.new(name,parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
      assert( uvm_config_db #(uart_dut_config)::get(this, "", "uart_dut_config", dut_config_0));
      uart_vi_in = dut_config_0.uart_vi_in;
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        get_and_drive();
    endtask : run_phase

    task get_and_drive();
        while (1) begin
            begin
                forever begin
                    uart_vi_in.i_uart_rxd = 1;
                    @(posedge $root.top.dut1.uart_dut.i_clk)
                    if(!uart_vi_in.o_uart_rts_n) begin
                        seq_item_port.get_next_item(req);
                        sent_uart_frame(req);
                        seq_item_port.item_done();
                    end

                end
            end
```

```
        end
    endtask : get_and_drive

    task sent_uart_frame(input uart_rx_frame req);
        int bit_counter = 0;
        wait(!uart_vi_in.o_uart_rts_n)
        uart_vi_in.i_uart_rxd = req.start_bit;

        while(bit_counter < 10) begin
            @(negedge uart_vi_in.i_uart_clk)
                if ((bit_counter >= 0) && (bit_counter < 8)) begin
                    uart_vi_in.i_uart_rxd = req.payload[bit_counter];
                end else begin
                    uart_vi_in.i_uart_rxd = req.stop_bits;
                end
                bit_counter++;
        end
    endtask : sent_uart_frame
endclass: uart_driver_in
```

A.12 uart_monitor_in.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class uart_monitor_in extends uvm_monitor;
    `uvm_component_utils(uart_monitor_in)

    uvm_analysis_port #(uart_rx_frame) aport;

    uart_dut_config dut_config_0;

    virtual uart_in uart_vi_in;

    function new(string name, uvm_component parent);
        super.new(name,parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
        dut_config_0=uart_dut_config::type_id::create("config");

        aport=new("aport",this);
```

```
        assert( uvm_config_db #(uart_dut_config)::get(this, "", "uart_dut_config", dut_config_0) );

        uart_vi_in=dut_config_0.uart_vi_in;

    endfunction: build_phase

    task run_phase(uvm_phase phase);
        uart_rx_frame frame;
        frame = uart_rx_frame::type_id::create("frame");
    @(posedge $root.top.dut1.uart_dut.i_clk)
        forever
        begin

        int monitor_counter = 0;
        wait(!uart_vi_in.o_uart_rts_n && !uart_vi_in.i_uart_rxd);
        repeat(1) @(posedge uart_vi_in.i_uart_clk);
        while(monitor_counter < 9) begin
            @(posedge uart_vi_in.i_uart_clk)
                if ((monitor_counter >= 0) && (monitor_counter < 8)) begin
                    frame.payload[monitor_counter] = uart_vi_in.i_uart_rxd;
                end
                monitor_counter++;
        end
        aport.write(frame);
       end
    endtask: run_phase
endclass: uart_monitor_in
```

A.13 uart_monitor_out.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class uart_monitor_out extends uvm_monitor;
  `uvm_component_utils(uart_monitor_out)

  uvm_analysis_port #(uart_tx_frame) aport;

  uart_dut_config uart_config_0;

  logic [3:0]  tx_state;
```

```
logic [11:0] tx_frame;

virtual uart_out uart_vi_out;

function new(string name, uvm_component parent);
  super.new(name,parent);
endfunction: new

function void build_phase(uvm_phase phase);
  uart_config_0=uart_dut_config::type_id::create("config");
  aport=new("aport",this);
  assert( uvm_config_db #(uart_dut_config)::get(this, "", "uart_dut_config", uart_config_0) );
  uart_vi_out=uart_config_0.uart_vi_out;
endfunction: build_phase

task run_phase(uvm_phase phase);
  repeat(1) @(posedge uart_vi_out.i_uart_clk);
  tx_state = 4'd0;
  tx_frame = 12'd0;
  uart_vi_out.i_uart_cts_n = 0;
  forever
  begin
    uart_tx_frame tx;
    wait(!$root.top.dut1.uart_dut.tx_fifo_empty);
    @(posedge uart_vi_out.i_uart_clk);
      tx = uart_tx_frame::type_id::create("tx");
      tx_state = (tx_state == 4'd12) ? 4'd0 :
      (tx_state != 4'd0) ? tx_state + 4'd1 :
      (uart_vi_out.o_uart_txd == 1'd0) ? 4'd1 : 4'd0;

      if(tx_state != 4'd0)begin
         tx_frame  = {uart_vi_out.o_uart_txd,tx_frame[11:1]};
         uart_vi_out.i_uart_cts_n = 1;
      end

                  if (tx_state == 4'd12) begin
       `uvm_info("tx_frame", $sformatf("data  %b",tx_frame), UVM_LOW);
       tx.tx_frame = tx_frame;
       tx_frame = 12'd0;
       aport.write(tx);
       uart_vi_out.i_uart_cts_n = 0;
    end
  end
endtask: run_phase
```

```
endclass: uart_monitor_out
```

A.14 uart_pkg.sv

```
package uart_pkg;
import uvm_pkg::*;
        `include "config.sv"
        `include "sequences.sv"
        `include "uart2wb_monitor.sv"
        `include "uart_driver_in.sv"
        `include "uart_monitor_in.sv"
        `include "uart_monitor_out.sv"
        `include "wb2uart_driver.sv"
        `include "wb2uart_monitor.sv"
        `include "agent_in.sv"
        `include "agent_out.sv"
        `include "scoreboard.sv"
        `include "cov.sv"
        `include "env.sv"
        `include "uart_test.sv"
        `include "tests.sv"
endpackage:uart_pkg
```

A.15 uart_test.sv

```
class uart_test extends uvm_test;
   `uvm_component_utils(uart_test)

   uart_dut_config dut_config_0;
   amber_dut_config dut_config_1;
   env env_h;

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
```

```
      dut_config_0 = new();
      dut_config_1 = new();
      if(!uvm_config_db #(virtual dut_in)::get( this, "", "dut_vi_in", dut_config_1.dut_vi_in))
        `uvm_fatal("NOVIF", "No virtual interface set for dut_in")

      if(!uvm_config_db #(virtual dut_out)::get( this, "", "dut_vi_out", dut_config_1.dut_vi_out))
        `uvm_fatal("NOVIF", "No virtual interface set for dut_out")

      if(!uvm_config_db #(virtual uart_in)::get( this, "", "uart_vi_in", dut_config_0.uart_vi_in))
        `uvm_fatal("NOVIF", "No virtual interface set for uart_in")

      if(!uvm_config_db #(virtual uart_out)::get( this, "", "uart_vi_out", dut_config_0.uart_vi_out))
        `uvm_fatal("NOVIF", "No virtual interface set for uart_out")

      uvm_config_db #(uart_dut_config)::set(this, "*", "uart_dut_config", dut_config_0);
      uvm_config_db #(amber_dut_config)::set(this, "*", "amber_dut_config", dut_config_1);

      env_h = env::type_id::create("env_h", this);
   endfunction: build_phase
endclass:uart_test
```

A.16 uart2wb_monitor.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class uart2wb_monitor extends uvm_monitor;
   `uvm_component_utils(uart2wb_monitor)

   uvm_analysis_port #(uart2wb) aport;

   amber_dut_config dut_config_0;

   virtual dut_out dut_vi_out;
   virtual dut_in dut_vi_in;

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      dut_config_0=amber_dut_config::type_id::create("config");
      aport=new("aport",this);
```

```
      assert( uvm_config_db #(amber_dut_config)::get(this, "", "amber_dut_config", dut_config_0) );
      dut_vi_out=dut_config_0.dut_vi_out;
      dut_vi_in = dut_config_0.dut_vi_in;

   endfunction: build_phase

   task run_phase(uvm_phase phase);
    @(posedge dut_vi_out.i_clk);
     forever
     begin
      uart2wb tx;
     @(negedge dut_vi_out.i_clk);
      if(~$root.top.dut1.uart_dut.rx_fifo_empty  ) begin
      dut_vi_in.i_wb_we = 0;
      dut_vi_in.i_wb_stb = 1;
      dut_vi_in.i_wb_adr = 0;
      tx = uart2wb::type_id::create("tx");
      repeat(1) @(posedge dut_vi_out.i_clk);
     @(negedge dut_vi_out.i_clk);
      dut_vi_in.i_wb_we = 0;
      dut_vi_in.i_wb_stb = 0;
      repeat(3) @(posedge dut_vi_out.i_clk);
      tx.i_clk    = dut_vi_out.i_clk;
      tx.o_wb_dat = dut_vi_out.o_wb_dat;
      tx.o_wb_ack = dut_vi_out.o_wb_ack;
      tx.o_wb_err = dut_vi_out.o_wb_err;
      aport.write(tx);
      repeat(100) @(posedge dut_vi_out.i_clk);
     end
     end
   endtask: run_phase
endclass: uart2wb_monitor
```

A.17 wb2uart_driver.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;

class wb2uart_driver extends uvm_driver#(wb2uart);
   `uvm_component_utils(wb2uart_driver)
```

```
  amber_dut_config dut_config_0;
  virtual dut_in dut_vi_in;

  function new(string name, uvm_component parent);
     super.new(name,parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    assert( uvm_config_db #(amber_dut_config)::get(this, "", "amber_dut_config", dut_config_0));
    dut_vi_in = dut_config_0.dut_vi_in;
  endfunction : build_phase

  task run_phase(uvm_phase phase);
   dut_vi_in.i_wb_stb  = 0;

   repeat(200)@(posedge dut_vi_in.i_clk);
   forever
   begin
    wb2uart tx;
    @(negedge dut_vi_in.i_clk);
     seq_item_port.get_next_item(tx);
     seq_item_port.item_done();

     if(((tx.i_wb_we && (~$root.top.dut1.uart_dut.tx_fifo_full)) || (~tx.i_wb_we &&
          (~$root.top.dut1.uart_dut.rx_fifo_empty))) && (tx.i_wb_stb)) begin
       dut_vi_in.i_wb_adr = {tx.i_wb_addr_hi,tx.i_wb_addr_lo};
       dut_vi_in.i_wb_we  = tx.i_wb_we;
       dut_vi_in.i_wb_stb = tx.i_wb_stb;
       dut_vi_in.i_wb_dat = tx.i_wb_dat;
       @(posedge dut_vi_in.i_clk);
       repeat(1)@(negedge dut_vi_in.i_clk);
       dut_vi_in.i_wb_stb = 1'b0;
      end
     if((~$root.top.dut1.uart_dut.tx_fifo_full) && tx.i_wb_we && tx.i_wb_stb)begin
      end
     repeat(2000) @(posedge dut_vi_in.i_clk);
    end
  endtask: run_phase
endclass: wb2uart_driver
```

A.18 wb2uart_monitor.sv

```
`include "uvm_macros.svh"
import uart_pkg::*;
class wb2uart_monitor extends uvm_monitor;
   `uvm_component_utils(wb2uart_monitor)

   uvm_analysis_port #(wb2uart) aport;

   amber_dut_config dut_config_0;

   virtual dut_in dut_vi_in;

   function new(string name, uvm_component parent);
      super.new(name,parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      dut_config_0=amber_dut_config::type_id::create("config");
      aport=new("aport",this);
      assert( uvm_config_db #(amber_dut_config)::get(this, "", "amber_dut_config", dut_config_0) );
      dut_vi_in=dut_config_0.dut_vi_in;
   endfunction: build_phase

   task run_phase(uvm_phase phase);
   @(posedge dut_vi_in.i_clk);
    forever
    begin
      wb2uart tx;
      @(posedge dut_vi_in.i_clk);
      if((dut_vi_in.i_wb_we && (~$root.top.dut1.uart_dut.tx_fifo_full)) && (dut_vi_in.i_wb_stb)) begin
         tx = wb2uart::type_id::create("tx");
         tx.i_wb_addr_hi = dut_vi_in.i_wb_adr[31:16];
         tx.i_wb_addr_lo   = dut_vi_in.i_wb_adr[15:0];
         tx.i_wb_we  = dut_vi_in.i_wb_we;
         tx.i_wb_dat = dut_vi_in.i_wb_dat;
         tx.i_wb_stb = dut_vi_in.i_wb_stb;
                aport.write(tx);
      end
    end
   endtask: run_phase
endclass: wb2uart_monitor
```

Bibliography

[1] Ramdas Mozhikunnath, and Robin Garg. Cracking Digital VLSI Verification Interview Interview Success, March 2016.

[2] 1M.Tech Student in DMS SVH College of Engineering, Machilipatnam, Krishna District, A.P., India 2 Professor and HOD, ECE Department, DMS SVH College of Engineering, Machilipatnam, Krishna District, A.P., India. UART IP Core Verification by using UVM, May 2016.

[3] Accellera Systems Initiative, and Elk Grove, Calif. Universal Verification Methodology, October 2015.

[4] Wei Ni and Xiaotian Wang. Functional Coverage-driven UVM-based UART IP Verification, November 2015.

[5] Rajesh Sarkar. Verification of UART, December 2019.