UVM Verification Plan

1. Introduction

The goal of this verification plan is to ensure that the UART design (Universal Asynchronous Receiver/Transmitter) operates correctly across various conditions and edge cases. The verification will be accomplished using UVM (Universal Verification Methodology), which is a standardized framework for verifying complex designs in SystemVerilog.

2. Testbenches and Components

The testbench consists of several UVM components that interact with the DUT (Design Under Test) to verify its functionality. These components include the **driver**, **monitor**, **scoreboard**, **sequencer**, and **coverage collector**.

3. Test Strategy

- **Functional Coverage**: Ensure that the UART functionality is exercised under different conditions and scenarios.
- **Code Coverage**: Track the exercised lines, branches, and FSMs to ensure thorough testing.
- **Transaction-level verification**: Focus on ensuring that the correct data is transmitted and received, and that no errors occur during the transmission.

4. Functional Coverage Goals

- Coverage for various data input scenarios (0-255) for the UART transmitter (tx_in).
- Ensure correct reception of data and matching of expected data through tx_out.

5. Code Coverage Goals

- Ensure that all lines of the UART driver, monitor, and sequencer are exercised.

6. Testbenches

Tests will involve generating sequences of transactions (data packets) and verifying the transmitted data with the expected data. The **simple_test** class will serve as the primary test to verify functionality.

UVM Architecture

The **UVM architecture** is based on a hierarchical structure where the testbench is broken down into different components, each responsible for a specific part of the verification process. Here's an overview of the key elements of UVM architecture:

1. UVM Test

A UVM test is the highest-level component that orchestrates the overall test flow. The **simple_test** class in the provided code represents the test and contains the entire environment setup, execution flow, and objection handling.

2. UVM Environment

The environment is the component that contains the agents, monitors, and scoreboards. The **uart_env** class encapsulates the UART agent and scoreboard.

3. UVM Agent

An agent represents the DUT interface. It connects to the DUT and contains the driver, sequencer, and monitor. The **uart_agent** class represents the agent for UART.

4. UVM Driver

The driver sends transactions to the DUT. The **uart_driver** class handles the sending of transactions like **tx_in** (data packets to be transmitted).

5. UVM Monitor

The monitor observes the signals of the DUT and sends data to the scoreboard for comparison. The **uart_monitor** class captures the transmitted and received data.

6. UVM Scoreboard

The scoreboard compares expected and actual outputs from the DUT to verify if the data is transmitted and received correctly. The **uart_scoreboard** class ensures that the received data matches the expected data.

7. UVM Sequencer

The sequencer controls the sequence of transactions being sent by the driver. The **uart_agent** class contains the sequencer that manages the sequence generation.

8. UVM Coverage Collector

The coverage collector records how much of the design's functionality has been exercised. The **uart_coverage** class collects functional coverage for the input data.

UVM Hierarchy and Component Descriptions

The UVM testbench uses the following hierarchy and components:

1. Top Level (top_tb_test)

- **Description**: The top-level testbench module, responsible for configuring the test environment and running the test.
- **Key Actions**: Sets the virtual interface and starts the test with run_test().

2. Interface (tb_ifc)

- **Description**: Provides the interface for communication between the UVM testbench and the DUT.
- **Key Actions**: Contains methods for transferring data and receiving inputs/outputs from the DUT.

3. Test Class (simple_test)

- **Description**: The main UVM test class that manages the entire test, including environment configuration, agent creation, and sequence execution.
- **Key Actions**: Configures the environment and agents, runs the test sequences, and handles objections.

4. Environment Class (uart_env)

- **Description**: Represents the test environment. It holds the UART agent, scoreboard, and other verification components.
- **Key Actions**: Creates and configures agents and scoreboards, manages connections between components.

5. Agent Class (uart_agent)

- **Description**: Contains the components interacting directly with the DUT: the driver, sequencer, and monitor.
- **Key Actions**: Instantiates and configures the driver, sequencer, monitor, and coverage components.

6. Driver Class (uart_driver)

- **Description**: Sends transactions to the DUT.

- **Key Actions**: Retrieves transactions from the sequencer and sends them to the DUT using the virtual interface.

7. Monitor Class (uart_monitor)

- **Description**: Observes the DUT and reports transactions.
- **Key Actions**: Captures the transmitted and received data, and sends it to the scoreboard for comparison.

8. Scoreboard Class (uart_scoreboard)

- **Description**: Compares expected and actual outputs to verify the correctness of the DUT.
- **Key Actions**: Receives data from the monitor, compares it with expected values, and reports mismatches.

9. Coverage Collector Class (uart_coverage)

- **Description**: Collects functional coverage for the transmitted data.
- **Key Actions**: Monitors and records of the different data values transmitted, helping ensure complete functional verification.

10. Sequencer Class (write_tx_sequence)

- **Description**: Controls the generation and flow of transactions.
- **Key Actions**: Generates and sends transactions to the sequencer for processing by the driver.

Comments:

We are currently enhancing the testbench to boost performance and aim to complete these tasks as soon as possible.

Github link: Defuse-cfg/UART-PROTOCOL