RAL PROJECT

TEAM 1

CONTRIBUTOR:

AMAN SASIDHARAN KP

GOVINDA SRIKALA REDDY

NIKITHA

OMAR FARHAN KHALIFA

PRASTHUTHI AMIN

RAKSHITHA R NAYAK

## Contents

[**Contents**](#_heading=h.gjdgxs)

**Chapter 1**

[Introduction](#_heading=h.30j0zll)

**Chapter 2**

[Architecture](#_heading=h.1fob9te)

* 1. [Testbench Architecture](#_heading=h.3znysh7)

[**Chapter 3**](#_heading=h.2et92p0)

[Implementation](#_heading=h.tyjcwt)

* 1. [Pin Interface](#_heading=h.3dy6vkm)
  2. [Testbench Components](#_heading=h.1t3h5sf) 
     1. Interface
     2. [Environment](#_heading=h.35nkun2)
     3. [Scoreboard](#_heading=h.1ksv4uv)
     4. [Register Agent](#_heading=h.44sinio)
     5. [Register Driver](#_heading=h.z337ya)
     6. [Register Monitor](#_heading=h.3j2qqm3)
     7. [Write Agent](#_heading=h.1y810tw)
     8. [Write Sequencer](#_heading=h.4i7ojhp)
     9. [Write Driver](#_heading=h.2xcytpi)
     10. [Write Monitor](#_heading=h.1ci93xb)
     11. [Read Agent](#_heading=h.3whwml4)
     12. [Read Monitor](#_heading=h.2bn6wsx)

[**Chapter 4**](#_heading=h.2s8eyo1)

[Verification Plan](#_heading=h.17dp8vu)

* 1. [Verification Plan](#_heading=h.qsh70q)

[**Chapter 5**](#_heading=h.3rdcrjn)

[Test Cases](#_heading=h.26in1rg)

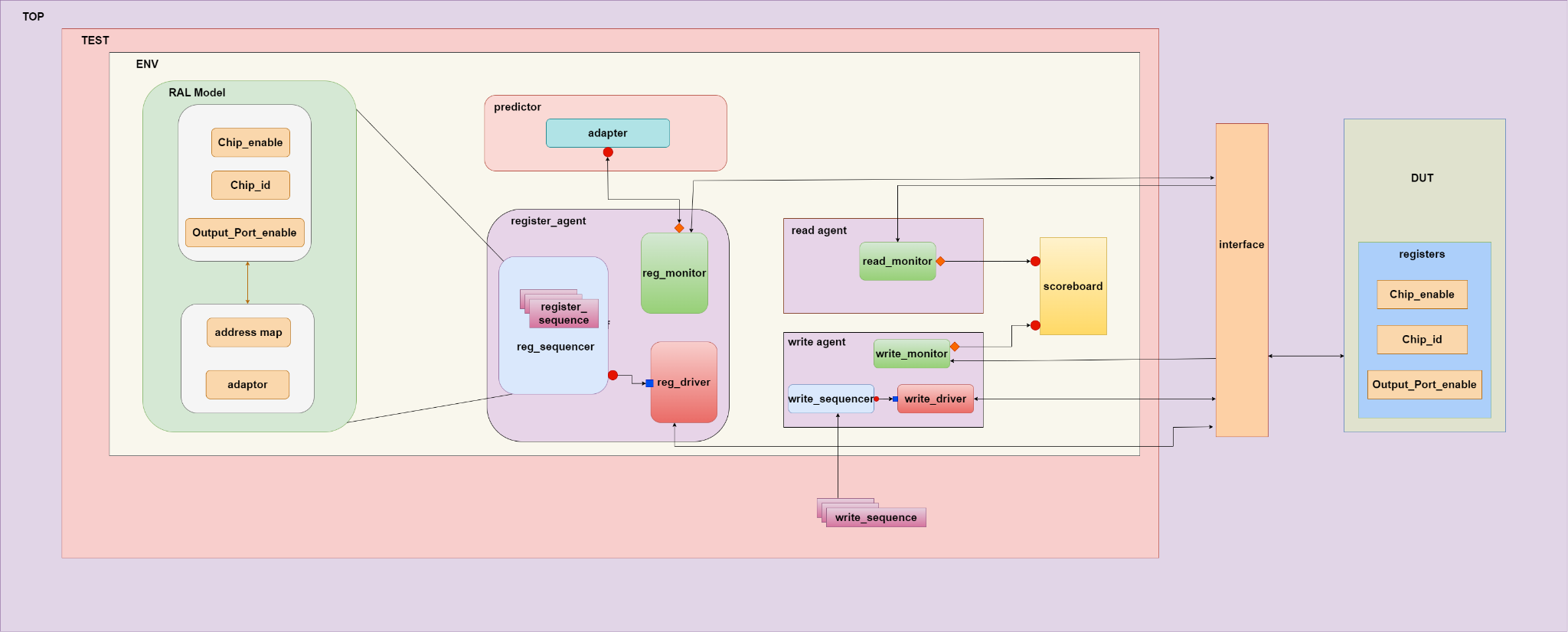
* 1. [Test Flow](#_heading=h.3as4poj)
  2. [Register Sequence item](#_heading=h.1pxezwc)
  3. [Write Sequence item](#_heading=h.49x2ik5)
  4. [Test Cases](#_heading=h.2p2csry)

# Introduction

This project aims to transmit packets containing source address, destination address, data, and ID to a DUT module. The DUT is configured to route incoming packets to specific output ports based on register settings. Register configuration is facilitated through an APB interface to tailor the DUT's behaviour. Upon processing, the DUT swaps the source and destination addresses in the output packet. The output format features the destination address preceding the source address, followed by data and ID.

# Architecture

#### Testbench Architecture



# Implementation

#### Pin Interface

Table 3.1 shows the APB pins used to interface to external devices.

|  |  |  |
| --- | --- | --- |
| Signals | Source | Description |
| paddr | APB bridge | Address. This is the APB address bus.It can be up to 32 bits wide and is a data access or an instruction access. |
| pselx | APB bridge | Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a pselx signal for each slave. |
| penable | APB bridge | Enable. This signal indicates the second and subsequent cycle of an APB transfer. |
| pwrite | APB bridge | Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW. |
| pwdata | APB bridge | Write data. This bus is driven by the peripheral bus bridge unit during the write cycle when pwrite is HIGH. This bus can be up to 32 bits wide. |
| prdata | Slave interface | Read Data.The selected slave drives this bus during read cycles when pwrite is LOW. This bus can be up to 32-bits wide. |

#### Testbench Components

In this section, testbench components of the apb-avip are discussed

##### Interface

Passing Signals: clk, rst

Declaration of signals: paddr, pwrite, pwdata, penable, prdata, valid\_in,valid\_out, outport\_port1, outport\_port2, outport\_port3, outport\_port4,are declared as logic type

##### Environment

The env class initializes and connects testbench components such as register, write, and read agents, along with the register environment and scoreboard. It configures the read agent as passive and establishes connections between agents, monitors, and predictors for efficient communication and coordination during verification.

##### Scoreboard

The `scoreboard` class serves as a crucial component for verifying transactions between the write and read agents in the UVM environment. It interfaces with read and write agent monitors through analysis ports (`r\_agt\_mon2scb` and `w\_agt\_mon2scb`) to receive transaction items for analysis. The class maintains queues (`sent\_pkt` and `received\_pkt`) to store expected and observed packets, respectively, enabling comparison during the check phase.

During the check phase, the scoreboard iteratively compares sent and received packets, controlled by parameter `N`. For each iteration, it extracts a pair of packets from the queues for comparison. The `compare\_pkt()` function meticulously examines various packet attributes, including source and destination addresses, packet ID, and data fields. It then reports any discrepancies between expected and observed packets, aiding in the identification of potential issues within the design or verification environment. Overall, the `scoreboard` class ensures that transactions happen correctly and align with the expected behavior.

##### Register Agent

The register\_agent class serves as a register agent in the environment.

During the build\_phase, the agent initializes its components, including the register driver (m\_drvr), register monitor (m\_mon), and register sequencer (m\_seqr), if the agent is in active mode (is\_active == UVM\_ACTIVE).

It also retrieves and sets up the virtual interface (vif) for communication with the DUT's APB interface.

In the connect\_phase, the agent establishes connections between the register driver and sequencer, ensuring that transactions generated by the driver are sequenced properly for the DUT.

##### Register Driver

The register\_driver class is responsible for driving APB transactions (both read and write) on the DUT interface (apb\_if). Here's a brief explanation of its logic:

* Transaction Handling:
  + The run\_phase task continuously waits for transactions from the sequence item port (seq\_item\_port.get\_next\_item(pkt)).
  + Upon receiving a transaction, it checks if it's a write or read transaction.
  + If it's a write transaction, it calls the write task to drive the address and data onto the interface.
  + If it's a read transaction, it calls the read task to drive the address and read data onto the interface.
* Write Task (write):
  + Sets the address and data on the interface (vif.paddr, vif.p\_wdata) and asserts control signals (vif.p\_write, vif.psel, vif.pen) to initiate the write operation.
  + Waits for two clock cycles to complete the write operation and then de-asserts control signals to end the transaction.
* Read Task (read):
  + Sets the address on the interface and de-asserts the write control signal to indicate a read operation.
  + Waits for two clock cycles for the read data to be available.
  + Retrieves the read data from the interface (vif.prdata) and de-asserts control signals to end the transaction.

This driver class provides a mechanism to drive transactions onto the DUT interface, handling both read and write operations in a synchronized manner with the DUT clock.

##### Register Monitor

The register\_monitor class in UVM is designed to continuously observe and report APB transactions between the testbench and the device under test (DUT). Here's a concise breakdown of its logic:

Initialization: It sets up an analysis port to broadcast transactions to other UVM components.

Configuration: Retrieves a virtual interface to the APB bus from the UVM configuration database.

Monitoring: Runs continuously in the background (using fork ... join\_none), watching for APB transactions when both the select (psel) and enable (pen) signals are active.

Data Capture:

* + Captures the address from the APB interface.
  + Depending on the write signal (p\_write), captures data from either the write data bus (p\_wdata) for writes or the read data bus (prdata) for reads.

Transaction Reporting: Constructs a transaction packet (pkt) with the captured data and sends it through the analysis port for consumption by other UVM components like scoreboards or coverage monitors.

This setup allows the monitor to efficiently track and report all relevant data transfers over the APB interface, facilitating effective verification of the DUT’s communication activities.

##### Write Agent

The write\_agent class is a component that coordinates the activities related to writing transactions. It manages three main components:

* Driver (dri): Responsible for driving write transactions onto the interface.
* Sequencer (seqr): Generates sequences of transactions to be driven by the driver.
* Monitor (write\_mon): Observes and checks the transactions on the interface to ensure correctness.

During the build phase, if the agent is active, it creates instances of the driver and sequencer components. It also creates an instance of the monitor component. In the connect phase, it connects the driver's sequence item port to the sequencer's sequence item export to facilitate the flow of transactions.

##### Write Sequencer

The write\_sequencer class is responsible for managing the generation and coordination of sequences of "write" transactions, specifically those of the write\_seq\_item type. It facilitates the flow of these transactions from higher-level sequences down to the driver for execution.

##### Write Driver

* The write\_driver class is responsible for driving transactions (write\_seq\_item) onto an APB interface (apb\_if).
* write\_seq\_item bus\_tx represents a transaction item of type write\_seq\_item, which contains fields like src\_addr, dest\_addr, id, and data.
* Virtual apb\_if vif declares a virtual interface handle for connecting to the APB interface of the DUT.
* In run phase asserts and de-asserts the reset signal (vif.rst) based on clock edges.Then enters a forever loop to continuously get the next transaction from the sequence item port, drive the packet to the DUT (drive\_packet\_dut()), and signal completion of the transaction.
* Drives the transaction packet to the DUT in the drive\_packet\_dut task. Forms the packet by concatenating the transaction fields into bus\_tx.pkt. Drives the packet's data and control signals to the DUT based on clock edges.
* Resets control signals (vif.Valid\_in) and clears the packet (bus\_tx.pkt) at the end of each transaction.

##### Write Monitor

The write\_monitor class continuously monitors the interface signals, particularly the clock signal (vif.clk), to synchronize its operations with the clock cycles of the design under test (DUT).

* Valid Transaction Detection:

When the Valid\_in signal goes high (vif.Valid\_in), indicating the presence of a valid

transaction on the interface, the monitor captures the transaction data.

* Transaction Capture:

For each valid transaction detected, the monitor captures the data from the Data\_in signal

(vif.Data\_in) for a duration of 64 clock cycles.

* The captured data is then stored in a write\_seq\_item transaction item named bus\_tx. This transaction item is designed to encapsulate the relevant information of a write transaction, such as the data being written.
* After creating the bus\_tx transaction item, the monitor writes it to an analysis port (wr\_mon\_ap). This analysis port serves as an interface for transmitting the captured transactions to the scoreboard.

##### Read Agent

The read\_agent class, derived from uvm\_agent, is designed for use in a UVM testbench to monitor read transactions in a passive mode. It features:

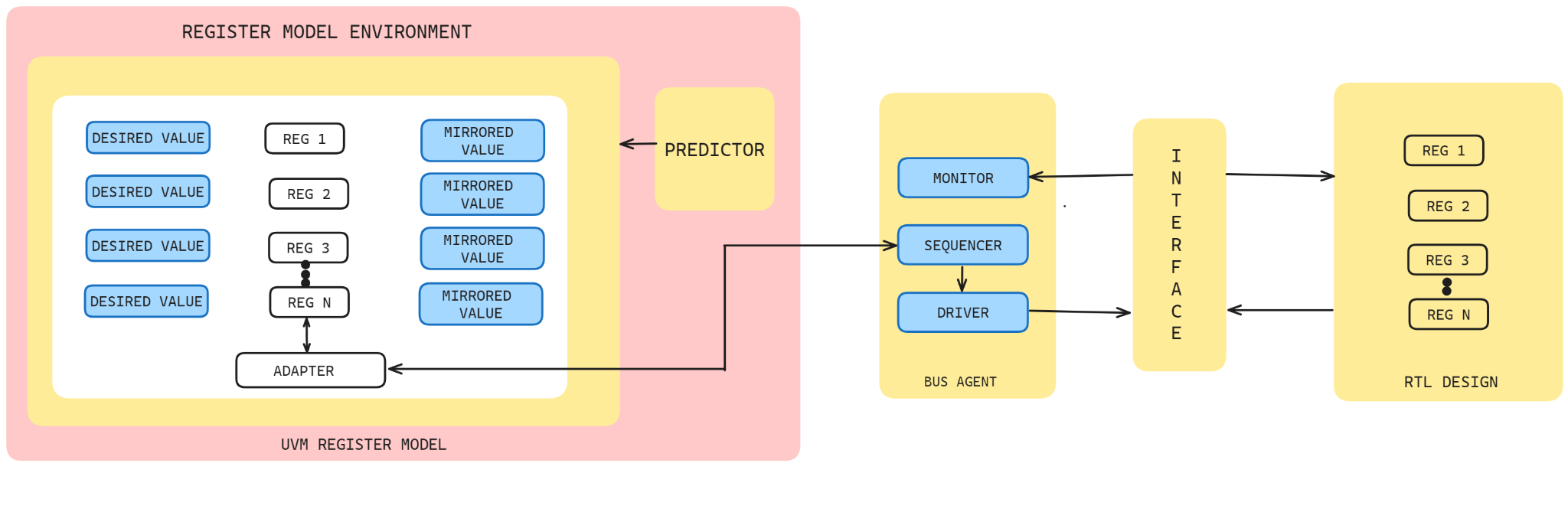
* Registration Macro: Utilizes uvm\_component\_utils for UVM factory compatibility.
* Monitor: Contains a read\_monitor for observing read transactions.
* Constructor: Initializes the agent with a specified name and parent.
* Build Phase: Constructs a read\_monitor and logs its role if set to passive mode

##### Read Monitor

The read\_monitor class observes read transactions on an interface in a UVM testbench:

* It continuously checks for valid read transactions.
* Upon detection, it collects data from the appropriate output port over 64 clock cycles.
* Collected data is stored in a transaction item (bus\_tx).
* Once a complete packet of data is collected, it is sent through an analysis port (rd\_mon\_ap) for further processing.

##### Register Block



# Verification Plan

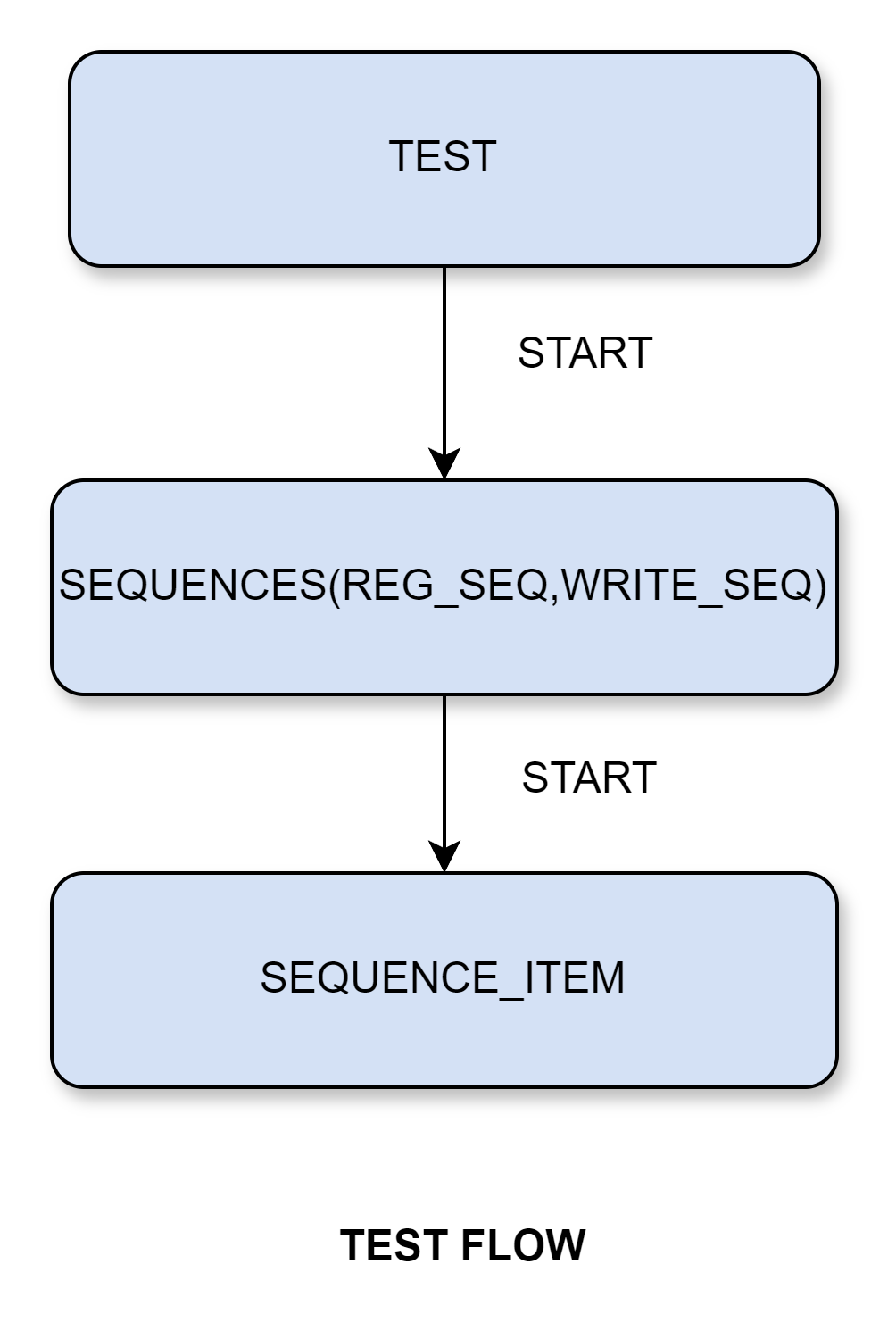
#### Verification Plan

We adhered to this verification plan throughout our testing process.

<https://docs.google.com/spreadsheets/d/1hTYTMFTQP9leOGpDyf0npQtX56zC4yLyLu88MB0OqJk/edit#gid=838750096>

# Test Cases

#### Test Flow



#### Register Sequence item

The apb\_tr(Register Sequence item) class models transactions in an APB interface. It contains variables for address, data, and write indication. The constraint ensures valid address selections. It's equipped with serialization macros and a constructor for initialization. Overall, it facilitates the simulation of APB transactions during verification.

#### Write Sequence item

The write\_seq\_item class defines data structures for representing write transactions in a verification environment. It contains variables for source and destination addresses, an identifier, data, and flags for transaction validity. Additionally, it includes fields for expected and actual packet data, facilitating comparison during verification. The constructor initializes the object with a provided name.

#### Test

The test class is a UVM test that orchestrates the verification process. It instantiates the test environment (env) and sequences for register (m\_seq) and write (w\_seq) operations. During the build phase, it creates instances of these components. In the run phase, it starts both register and write sequences concurrently using fork-join construct, then waits for a specified duration before dropping objections, signifying the end of the test.